Physik auf dem Computer I & II

PD Dr. S. Luding, Prof. Dr. H. J. Herrmann,
Dr. S. Schwarzer, Dipl. Phys. Matthias Müller,
Dr. H.-G. Matuttis, Dipl. Phys. M. Brunner,
Dipl. Phys. M. Lätzel, Dipl. Phys. C. Manwart
T. Karle und S. Manmana
Uni–Stuttgart

8. Juli 1999

partielle Differentialgleichungen, sowie die Möglichkeiten der linearen Algebra und nicht zuletzt auch der 3D Grafik.

Vieles läßt sich durch die bereits mehrfach empfohlene online Hilfe finden, für weitere Interna müssen wir auf die Literatur verweisen.

```
> writedata ( ‘data1‘, lst );
```

wieder in eine Datei `data1` zurückgeschrieben werden. Man beachte, daß Maple Zeichenketten wie Dateinamen durch die Benutzung von Rechtsakzenten ‘ identifiziert. Maple sucht dann nicht nach Variablen, die es auswerten könnte.

Um die manchmal komplexen Ausdrücke, die als Ergebnis einer Sitzung mit Maple entstehen können auch in externen numerischen Rechnungen nutzen zu können, gibt es Routinen, die Ausdrücke in syntaktisch korrekte Formen externer Programmiersprachen bringen können, hier am Beispiel einer im Hinblick auf numerische Auswertungen optimierten Lösung eines Polynoms dritten Grades:

```
> restart:
> polyeqn := x^3 - a*x =1;
```

$$polyeqn := x^3 - a\,x = 1$$

```
> sols := solve ( polyeqn, x );
```

$$sols := \frac{1}{6}\,\%1^{1/3} + 2\,\frac{a}{\%1^{1/3}},\ -\frac{1}{12}\,\%1^{1/3} - \frac{a}{\%1^{1/3}} + \frac{1}{2}\,I\,\sqrt{3}\,(\frac{1}{6}\,\%1^{1/3} - 2\,\frac{a}{\%1^{1/3}}),$$
$$-\frac{1}{12}\,\%1^{1/3} - \frac{a}{\%1^{1/3}} - \frac{1}{2}\,I\,\sqrt{3}\,(\frac{1}{6}\,\%1^{1/3} - 2\,\frac{a}{\%1^{1/3}})$$
$$\%1 := 108 + 12\,\sqrt{-12\,a^3 + 81}$$

```
> # erste Loesung nehmen
> sol1 := sols[1]:
> # Funktion zur Umwandlung in C nachladen
> readlib( C);
> # und Programmcode erzeugen
> C( sol1, ‘optimized‘);
```

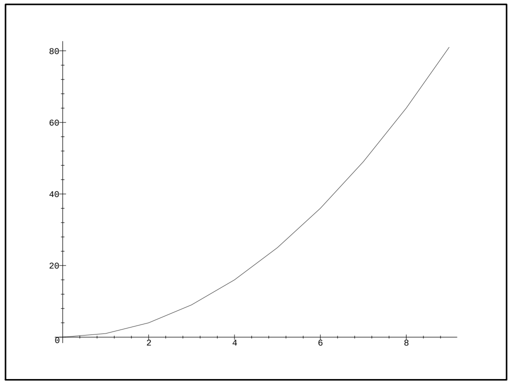
$$\mathbf{proc}()\ \ldots\ \mathbf{end}$$

```
      t1 = a*a;
      t4 = sqrt(-12.0*t1*a+81.0);
      t6 = pow(108.0+12.0*t4,1.0/3.0);
      t9 = t6/6+2.0*a/t6;
> #   bzw gleich in eine Datei schrieben
> C( sol1, ‘optimized‘, filename = ‘file.c‘);
```

14.12 Weiteres

Damit haben wir unseren Schnelldurchgang durch Maple beendet und haben eine grosse Menge von Funktionalität noch nicht besprochen. Hierzu gehören viele zahlentheoretische Funktionen, die Interna der Integrations- und Differntiationsroutinen, Auswertung von Reihen und Grenzwertbildung, Maples verschiedene Möglichkeiten, Ausdrücke zu vereinfachen, zu faktorisieren, nach Vorfaktoren zu ordnen, die Löser für gewöhnliche und

Inhaltsverzeichnis



Im letzten Kommando hätten wir mit dem optionalen Argument `style=point` Einzelpunkte ausgeben können. Auch dreidimensionale Darstellungen sind in grosser Vielfalt möglich. Wir verweisen dazu auf Maple's online Hilfefunktion.

14.11 Ein- und Ausgabe, Zusammenarbeit mit anderen Programmen

Neben der Möglichkeit, die Arbeitsblätter extern abzulegen und bei Bedarf wieder zu laden, kann Maple auch Daten aus externen Daten in interne Datenstrukturen einlesen und dann bearbeiten. Für geeignet formatierte Daten gibt es dazu die Funktion `readdata`, die als erstes Argument den Namen einer Datei, als zweites die Anzahl der zu lesenden Datenspalten erwartet. Die Funktion erzeugt eine Liste von Listen, die sich zur Anzeige mit `plot` eignet.

Nehmen wir an, die Datei `data` enthalte den folgenden Inhalt,

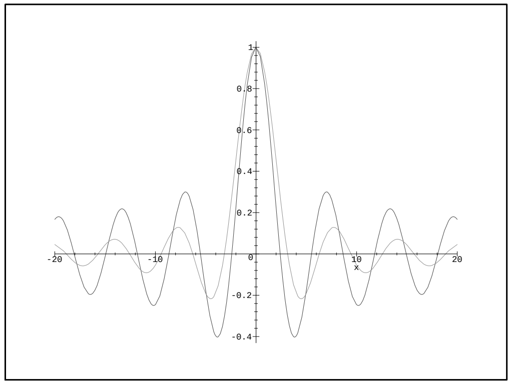
```
1 2   3
2 5   7
3 5.2 9
```

dann erzeugt `readdata( ‘data‘, 3 )` die Liste

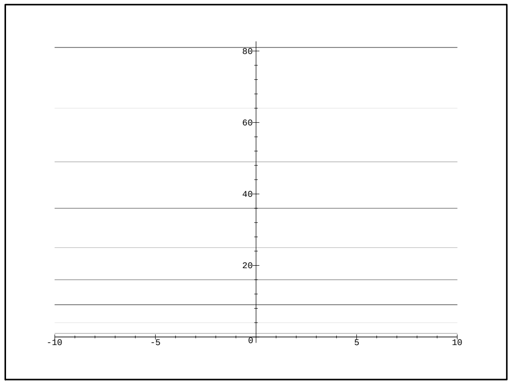
```
> lst := readdata( ‘data‘, 3 );
```

$$lst := [[1., 2., 3.], [2., 5., 7.], [3., 5.2, 9.]]$$

Diese kann mit




```
> # eine Liste von einzelnen Datenpunktenl, die plot leider nur
> # zu parallelen Linien ueberredet
> lst:= [seq( i^2, i=0..9 ) ]: plot ( lst );
```




```
> # um Punktmengen darzustellen, muessen wir
> # (x,y) Paare erzeugen
> pairs := \{ seq( [ i, lst[i+1] ], i=0..9 )\};
```

$$pairs := \{[0, 0], [1, 1], [2, 4], [3, 9], [4, 16], [5, 25], [6, 36], [7, 49], [8, 64], [9, 81]\}$$

```
> plot ( pairs );
```

```
> # matrix und eine anonyme Funktion zur Erzeugung von b
> b:=matrix( 2, 2, (i,j) -> 2*i +j ):
> # Maple merkt, dass a nicht den passenden Indexbereich besitzt
> evalm( a &* b );
```

$$a \,\&*\, \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix}$$

```
> # wir wandeln a also mittels Listenschreibweise in eine Matrix um
> c := matrix( 2,2, [ seq( seq( a[i,j],i=0..1),j=0..1) ] );
```

$$c := \begin{bmatrix} 7 & e^5 \\ e^5 & a_{1,1} \end{bmatrix}$$

```
> # das Ergebnis von evalm ist eine Matrix
> mproduct:= evalm ( c &* b );
```

$$mproduct := \begin{bmatrix} 21 + 5\,e^5 & 28 + 6\,e^5 \\ 3\,e^5 + 5\,a_{1,1} & 4\,e^5 + 6\,a_{1,1} \end{bmatrix}$$

```
> # die wir als solche weiterverarbeiten k\"onnen
> eigenvals( mproduct );
```

$$\frac{21}{2} + \frac{9}{2}\,e^5 + 3\,a_{1,1} + \frac{1}{2}\sqrt{441 + 378\,e^5 + 308\,a_{1,1} + 73\,(e^5)^2 + 108\,e^5\,a_{1,1} + 36\,{a_{1,1}}^2},$$
$$\frac{21}{2} + \frac{9}{2}\,e^5 + 3\,a_{1,1} - \frac{1}{2}\sqrt{441 + 378\,e^5 + 308\,a_{1,1} + 73\,(e^5)^2 + 108\,e^5\,a_{1,1} + 36\,{a_{1,1}}^2}$$

14.10 Grafik

Ein Beispiel für Grafik mit Maple haben wir bereits in Abschnitt 14.5 gesehen, als wir einen Funktionsverlauf grafisch dargestellt haben. `plot` ist in der Lage zweidimensionale Graphen, die entweder in expliziter Form (y-Wert als Formel oder Funktion, x variiert) oder in parametrischer Form (x und y als Funktionen einer weiteren Variable, des Kurvenparameters) vorgegeben sind. Über optionale Parameter wie `color, style, thickness` läßt sich die Ausgabe beeinflussen. Im Beispiel zeigen wir den Vergleich einer Bessel-Funktion mit einem abklingenden Sinus, sowie die Manipulationen, die erforderlich sind, um eine Punktwolke darzustellen.

```
> restart:
> plot( \{BesselJ(0,x), sin(x)/x\}, x=-20..20);
```

14.9 Lineare Algebra

Der Einsatz von Feldern liegt insbesondere in der linearen Algebra, in der mit Matrizen umgegangen werden muß. Matrizen sind spezielle Felder, deren Indexbereiche immer mit 1 beginnen. Das Paket zu Aufgabenstellungen aus der linearen Algebra stellt die Funktion `matrix` zur Verfügung, mit deren Hilfe sich Matrizen leicht erzeugen lassen, wenn man eine Funktion angeben kann, die die Einträge berechnet. Ansonsten kann die Angabe der Werte auch über geschachtelte Listen erfolgen.

Die Auswertung von Ausdrücken die Matrizenaddition oder -multiplikation verwenden, geschieht mit `evalm`. Bei der Matrizenmultiplikation muß beachtet werden, daß das Symbol `*` in Maple für eine *kommutative* Multiplikationsoperation reserviert ist. Um die Nichtkommutativität von Matrizenmultiplikationen anzudeuten, wird der Operator `&*` verwendet.

Die Vielfalt der möglichen Operationen sieht man der Ausgabe von Maple beim Laden der Bibliothek durch die Anweisung `with(linalg);` an.

```
> restart:
> a:= array( symmetric, 0..1, 0..1, [ (0,0)=7,
> (0,1) = exp(5) ] );
```

$$a := \mathrm{array}(symmetric, 0..1, 0..1, [\\ (0, 0) = 7\\ (0, 1) = e^5\\ (1, 0) = e^5\\ (1, 1) = a_{1,1}\\ ])$$

```
> with ( linalg );
Warning, new definition for norm
Warning, new definition for trace
```

[*BlockDiagonal*, *GramSchmidt*, *JordanBlock*, *LUdecomp*, *QRdecomp*, *Wronskian*, *addcol*, *addrow*, *adj*, *adjoint*, *angle*, *augment*, *backsub*, *band*, *basis*, *bezout*, *blockmatrix*, *charmat*, *charpoly*, *cholesky*, *col*, *coldim*, *colspace*, *colspan*, *companion*, *concat*, *cond*, *copyinto*, *crossprod*, *curl*, *definite*, *delcols*, *delrows*, *det*, *diag*, *diverge*, *dotprod*, *eigenvals*, *eigenvalues*, *eigenvectors*, *eigenvects*, *entermatrix*, *equal*, *exponential*, *extend*, *ffgausselim*, *fibonacci*, *forwardsub*, *frobenius*, *gausselim*, *gaussjord*, *geneqns*, *genmatrix*, *grad*, *hadamard*, *hermite*, *hessian*, *hilbert*, *htranspose*, *ihermite*, *indexfunc*, *innerprod*, *intbasis*, *inverse*, *ismith*, *issimilar*, *iszero*, *jacobian*, *jordan*, *kernel*, *laplacian*, *leastsqrs*, *linsolve*, *matadd*, *matrix*, *minor*, *minpoly*, *mulcol*, *mulrow*, *multiply*, *norm*, *normalize*, *nullspace*, *orthog*, *permanent*, *pivot*, *potential*, *randmatrix*, *randvector*, *rank*, *ratform*, *row*, *rowdim*, *rowspace*, *rowspan*, *rref*, *scalarmul*, *singularvals*, *smith*, *stack*, *submatrix*, *subvector*, *sumbasis*, *swapcol*, *swaprow*, *sylvester*, *toeplitz*, *trace*, *transpose*, *vandermonde*, *vecpotent*, *vectdim*, *vector*, *wronskian*]

Tabellen und Felder

Tabellen sind Zuordnungsvorschriften, die einem Element eines Schlüsseldatentyps einen "Wert" zuordnen. Wir haben Tabellen bei der Diskussion der `remember` Option von Prozeduren kennengelernt. Intern sind Tabellen als *Hash tables* organisiert, d.h. daß die Position des gesuchten Elementes i.w. durch eine Indexfunktion ermittelt wird und so ein sehr schneller Zugriff unabhängig von der Elementposition erfolgen kann.

```
> table( [seq(i=2*i,i=0..1) ] );
```

table ([
0 = 0
1 = 2
])

```
> a[1];
```

2

Felder verhalten sich wie mehrdimensionale Tabellen mit speziellen Vereinbarungen für die zulässigen Indizes, die frei wählbare, aber lückenlose Bereiche von ganzen Zahlen sein müssen. Nicht alle Feldelemente müssen definiert werden. Das Feld kann zusätzliche Eigenschaften aufweisen, wie `sparse, tridiagonal, identity, diagonal, symmetric, antisymmetric`, die Maple erlauben, die Werte speichereffizient abzulegen. Diese Eigenschaften sind letztlich Indizierungsfunktionen, die man als Benutzer auch selbst definieren kann.

```
> a:= array( symmetric, 0..1, 0..1, [ (0,0) = 7, (0,1) = exp(3) ] );
```

$a :=$ array($symmetric$, 0..1, 0..1, [
$(0, 0) = 7$
$(0, 1) = e^3$
$(1, 0) = e^3$
$(1, 1) = a_{1,1}$
])

```
> a[1,0];
```

e^3

Listen

Im unteren Beispiel berechen wir mit `allvalues` eine Sequenz von Nullstellen einer Funktion, die wir mit eckigen Klammern `[]` umgeben, wodurch eine geordnete Liste entsteht, die wir dann mit `map` weiterverarbeiten. `map` entnimmt jedes Element aus der Liste, und formt mit den Ergebnissen der Anwendung der als erstes Argument spezifizierten Funktion eine neue Liste, unter Beibehaltung der Reihenfolge.

```
> # Maple scheint zu dumm, um Nullstellen zu ermitteln
> f:= x^4-x-1: solveI f=0, x);
```

$$\mathrm{RootOf}(\_Z^4 - \_Z - 1)$$

```
> # wir koennen aber noch hartnaeckiger nachfragen,
> # gleichzeitig verpacken wir die Nullstellen in eine Liste []
> zeroes := [ evalf( allvalues( ") , 20) ];
```

$$\begin{aligned}zeroes := [&-.72449195900051561159,\\ &-.24812606280262193189 - 1.0339820609759677567\,I,\\ &-.24812606280262193189 + 1.0339820609759677567\,I,\\ &1.2207440846057594754]\end{aligned}$$

```
> # Zur Probe Einsetzen in f, Berechnen der Funktionswerte
> map( y-> evalf( subs( x=y, f) ) , zeroes );
```

$$[0, 0, 0, 0]$$

Listen können beliebig ineinander verschachtelt werden und beliebige Datentypen als Elemente enthalten. Maple-intern sind Listen lineare Felder von Zeigern auf ihre Elemente. Die Anzahl der Elemente wird durch `nops` ermittelt, der direkte Zugriff auf Elemente erfolgt durch einen in eckigen Klammern angegebenen Index. Das erste Element der Liste trägt im Gegensatz zu C aber in Übereinstimmung mit z.B. FORTRAN den Index 1.

Mengen

Mengen (`sets`) werden in Maple durch geschweifte Klammern angedeutet. Sie unterscheiden sich in ihrem Verhalten von Listen dadurch, daß Maple automatisch gleiche Elemente eliminiert und gleichzeitig die Erhaltung der Reihenfolge nicht garantiert ist. Mengen werden häufig als Eingabeparameter von Prozeduren verwendet, wenn mehrere Variablennamen oder Gleichungen übergeben werden müssen, bei denen es ja auf die Reihenfolge nicht ankommt. Eine solche Anwendung haben wir bereits am Ende des Beispieles in Abschnitt 14.5 gesehen, als wir eine Grafik mit mehreren Kurven generiert haben.

14.7 Objektattribute

Maple scheint bei bestimmten einfachen Umformungen zu versagen. So weigert es sich etwa, `sqrt(a*a)` zu `a` zu vereinfachen. Nun ist diese Vereinfachung tatsächlich nur dann gültig, wenn a eine positive reelle Zahl ist. In allen anderen Fällen muß die Mehrdeutigkeit der Wurzelfunktion beachtet werden. Dieses Problem tritt ganz allgemein auf bei analytisch fortsetzbaren Funktionen, die ggfs. mehrdeutige Wertebereiche aufweisen. Solche Funktionen führen Maples Integrationsroutinen häufig bei komplexen Kurventintegralen in die Irre.

Als eine teilweise Lösung des Problems erlaubt Maple es daher, mit Hilfe der `assume` Prozedur eine Variable mit einer Eigenschaft *(property)* auszustatten. Dabei sind bestimmte Schlüsselworte wie `real, positive, negative, nonneg, integer, imaginary` aber auch Intervallangaben zulässig: z.B. sind `assume( a, nonneg )` und `assume( a>= 0)` gleichbedeutend.

```
> sqrt(a*a);
```

$$\sqrt{a^2}$$

```
> # Eine mit speziellen Eigenschaften versehene Zahl ist durch
> # ein nachgestelltes ~ gekennzeichnet
> assume( a>= 0); sqrt(a^2);
```

$$a\tilde{}$$

```
> # Manchmal ist es eine gute Idee, Maple nach seinen Vorurteilen
> # zu fragen:
> about (a);
Originally a, renamed a\symbol{126}:
  is assumed to be: RealRange(0,infinity)
```

Maple ist in der Lage, mit diesen Eigenschaften zu "rechnen." Schränkt man etwa `a` darauf ein, eine reelle Zahl zu sein, dann kann Maple deduzieren, daß dann `exp(a)` eine positive reelle Zahl ist. Damit kann es den Ausdruck `sqrt( (exp(a))^2 )` zu `exp(a)` vereinfachen.

14.8 Weitere Datenstrukturen

Maple gibt in Fällen, in denen mehrere Antworten richtig sind, häufig als Ergebnis eine *Folge* zurück. Dies ist eine durch Kommata getrennte Aufzählung von Ausdrücken, die weder in eckigen noch geschwungenen Klammern steht. Z.B. bekommen wir eine Folge, wenn wir nach den Nullstellen eines Polynoms fragen. Folgen können auch explizit generiert werden. Z.B. erzeugt `seq(i^2, i=0..9)` die Sequenz der ersten 10 Quadratzahlen $0, 1, 4 \ldots 81$.

$$f := x \to x^2 + bx + c$$

14.6 Verzögerung der Auswertung, Numerik

Eine Möglichkeit, die Auswertung von Ausdrücken zu unterdrücken, haben wir bereits kennengelernt. Selbst wenn der Name `x` auf ein anderes Objekt verweist, kann man sich mit linksgerichteten Akzenten auf `'x'` selbst beziehen, `x` wird nicht ausgewertet.

Eine weitere Möglichkeit, die Auswertung zu verhindern, ist die Verwendung *inerter* Prozeduren. Diese beginnen mit Großbuchstaben, schreiben sich aber sonst wie ihre fleißigeren Geschwister: `Int, Limit, Sum, Diff,` etc. So ist beispielsweise die inerte Funktion `Int` notwendig, um eine numerische *Integration* der Originalfunktion zu bewirken. Ansonsten versucht Maple das Integral zunächst analytisch auszuwerten und erst dann den entstehenden Ausdruck numerisch zu berechnen. Im folgenden Beispiel ist eine Funktion gezeigt, deren uneigentliches Integral Maple in Release 4 falsch zu 0 auswertet. Man muß daher die analytische Auswertung unterdrücken, um noch ein (richtiges) numerisches Resultat zu bekommen.

```
> # diese Funktion ist > 0 fuer alle x, daher ist das
> # Integral ueber R auch groesser als 0
> f := x-> 1/(x^4+x+1);
```

$$f := x \to \frac{1}{x^4 + x + 1}$$

```
> # maple ist leider anderer Meinung
> int(f(x), x=-infinity..infinity);
```

$$0$$

```
> # daher wollen wir die Integration numerisch durchfuehren
> Int(f(x), x=-infinity..infinity);
```

$$\int_{-\infty}^{\infty} \frac{1}{x^4 + x + 1}\, dx$$

```
> # mit 30 Stellen Genauigkeit
> evalf ( ", 30 );
```

$$2.68354824425647647957447074409$$

```
> # Zeilenumbruch innerhalb von Prozeduren: SHIFT+RETURN
> # Typueberpuefung von Argumenten optional, auch Liste moeglich
> fibo := proc(n::nonneg)
> Zeilenumbruch SHIFT+ENTER
>   if n > 1 then fibo(n-1) + fibo(n-2)
>   else 1
>   fi;
> end;
```

$$fibo := \mathbf{proc}(n::nonneg)\ \mathbf{if}\ 1 < n\ \mathbf{then}\ \mathrm{fibo}(n-1) + \mathrm{fibo}(n-2)\ \mathbf{else}\ 1\ \mathbf{fi}\ \mathbf{end}$$

```
> fibo(3), time(fibo(20)); op( 4, eval(fibo));
```

$$3, 4.339$$

```
> # option remember:
> #  Funktion erinnert nun automatisch bereits berechnete Werte
> fibo := proc(n::nonneg) option remember;
>   if n > 1 then fibo(n-1) + fibo(n-2)
>   else 1
>   fi;
> end;
```

$$fibo := \mathbf{proc}(n::nonneg)\ \mathbf{option}\, remember;\ \mathbf{if}\ 1 < n\ \mathbf{then}\ \mathrm{fibo}(n-1) + \mathrm{fibo}(n-2)\ \mathbf{else}\ 1\ \mathbf{fi}\ \mathbf{end}$$

```
> fibo(3), time(fibo(3000));
```

$$3, 1.329$$

Die Tiefen von Maple lassen sich durch Studium des Maple Programmcodes der Maple-Prozeduren ergründen. Diese erhält man nach dem Aufruf `interface( verboseproc=3 )` mit `print`*(Prozedurname)*. Unter anderem läßt sich so auch der Differentiationsoperator `D` entweihen ...

Wo Maple sonst einen Ausdruck erwartet, kann auch eine Prozedur mit entsprechenden Argumentvariablen stehen. Die Wahl des Namens dieser Variable kann jetzt aber natürlich eine andere sein als bei der Definition der Prozedur. Wie im Beispiel gesehen, muß man darauf achten, *niemals* einen Audruck der Form `f(x) := x^2` zur Funktionsvereinbarung zu verwenden. Wie wir oben gesehen haben, merkt sich Maple dann nur, was f zurückliefern soll, wenn das Argument gerade `'x'` ist.

Mit Hilfe der Funktion `unapply` ist es möglich, Variablen eines Ausdrucks als formale Argumentparameter zu kennzeichen und somit einen Ausdruck in eine echte Funktion umzuwandeln.

```
> expression := x^2 + b *x + c:
> f := unapply(expression, x);
```

$$f := x^2 + a\,x + 1$$

gesehen. Hier wurde `f` ein Ausdruck mit Variablen zugewiesen. Maple zeichnet keine Variable in besonderer Form aus, so daß vielen Operationen noch mitgeteilt werden muß, bezüglich welcher Variablen sie arbeiten sollen, so wir wir das oben am Beispiel der Integration und Differentiation gesehen haben. Ein Problem ist, daß die als Unbekannte benutzte Variable (sagen wir x) nachträglich an einen Wert oder einen anderen Variablennamen gebunden werden kann (vgl. Abschnitt *Zuweisungen*), was interessante Effekte hervorruft:

```
> diff(f,x);
```

$$2\,x + a$$

```
> x:=2: diff(f, x);
Error, wrong number (or type) of parameters in function diff
> x:= a: diff(f,x);
```

$$4a$$

Das liegt letztlich natürlich daran, daß Maple zunächst auswertet und dann erst die gewünschte Operation durchführt. Mit einer "echten" Funktion erhalten wir die Freiheit, den für das Argument verwendeten Parameter frei zu wählen. Maple nutzt dabei zwei äquivalente Schreibweisen, eine mittels eines Pfeiloperators, die andere mittels Schlüsselwort `proc`.

```
> f:= x->x^2;
```

$$f := x \to x^2$$

```
> f(y), f(4);
```

$$y^2,\ 16$$

```
> # ein Fehler, der haeufig auch passiert, bevor f definiert ist
> # Bedeutung: ein spezieller Wert von f wird errinnert
> f(4):=17: f(4);
> }
```

$$17$$

```
> # op erlaubt, Komponenten eines Ausdrucks zu selektieren
> # op( 0, eval(f) ) 0=Funktionsname; ..;  4=remember table
> op( 4 , eval(f) );
```

table([
    4 = 17
    ])

Kapitel 1

Einführung

1.1 Was heißt "Physik auf dem Computer"

In dieser Vorlesung soll gezeigt werden, wie man physikalische Fragestellungen mit dem Computer angehen kann. Dies kann auf vielfältige Weise geschehen:

- *Schreibe die Diplomarbeit mit dem Computer*
  Alleine die Textverarbeitung ist nicht mehr aus dem wissenschaftlichen Leben wegzudenken – handgeschriebene Texte sind heutzutage nicht mehr akzeptabel (wie in Prä-Computer Zeiten).

- *Benütze den Computer als besseren Taschenrechner*
  Man macht Physik auf althergebrachte Weise und schaltet den Computer nur im Notfall ein. Je länger man mit dem Computer arbeitet, umso mehr Anwendungsmöglichkeiten wird man finden: z.B. **Grafik**, **Internet-Zugang** oder schnelle Kommunikation via **e-mail**.

- *Steuere Experimente mit dem Computer*
  Der Computer arbeitet auch nachts und ist viel ausdauernder als der fleissigste Student oder Wissenschaftler. Die riesige Datenmenge, die bei Großexperimenten anfällt ist von Menschen überhaupt nicht mehr zu bewältigen.

- ***Werte experimentelle Daten aus***
  Hat man einmal einen Datensatz (z.B. Flugbahn eines UFOs), so kann man unzählige Auswertungs-Operationen durchführen: Aus der Flugbahn kann man extrapolieren, wo das Flugobjekt herkam oder hinfliegt; Interpolation oder Glättung macht aus einer Punktewolke ein glatte Funktion, von der man dann Ableitungen berechnen kann, um die Beschleunigung zu bestimmen.

- ***Löse physikalische Probleme mit dem Computer***
  Nur die wenigsten Fragestellungen in der Physik sind analytisch “mit Bleistift und Papier” lösbar. Es werden jedoch viele Prozesse in der Natur durch komplizierte, nicht-lösbare Differentialgleichungen beschrieben. Der Computer hilft z.B. solche Gleichungen numerisch zu lösen (reales, nichtlineares Pendel) oder hilft bei der analytischen Lösung mit sog. “symbolischen” Programmiersprachen (z.B. MAPLE).

- ***Benutze den Computer für numerische “Experimente”***
  Zum Verständnis vieler physikalischer Probleme, kann man – im Gegensatz zum realen Experiment – das System stufenweise vereinfachen, bis nur noch die essentielle Physik übrigbleibt. Verhält sich das vereinfachte System ebenso wie das reale System, ist man dem Verständniss einen Schritt näher gekommen, da man unwichtige Effekte eliminiert hat. Man kann mit dem Computer sogar “unmögliche” Experimente durchführen. Wissenschaftler untersuchen z.B. Quanten-Phänomene, Sonnen oder sogar schwarze Löcher u.v.m., also Systeme, die experimentell nicht zugänglich sind.

In dieser Vorlesung werden wir uns vor allem auf die letzten drei Punkte konzentrieren. Zuvor jedoch noch einige grundlegende Bemerkungen und Erklärungen.

1.1.1 Vom Menschen zum Computer und zurück

Am Anfang steht eine Idee oder ein zu lösendes Problem. Der Forscher muß dieses nun in “Worte” fassen, die der Computer verstehen muß, um bei der Lösung helfen zu können. Eine geeignete “Sprache” nennt man **Programmiersprache** und die Befehle, die der Computer ausführen soll, **Programm**. Die Art und Weise, wie man das zu lösende Problem formuliert, nennt man **Algorithmus**. Dieser hängt a-priori nicht von der Programmiersprache ab, auch wenn bestimmte Sprachen für manche Algorithmen besser geeignet sein können als andere. Es gibt allerdings Abstufungen, was die Maschinennähe, bzw. Benutzerorientierung angeht. Besonders maschinennah sind Assemblersprachen, ein flexibler und allgemeinerer Kompromiss sind höhere Programmiersprachen (C/C++, PASCAL, FORTRAN) und besonders benutzernah (weil problemorientiert) sind Sprachen wie LISP und Interpreter wie MATHLAB oder MAPLE. Mit letzteren kann man bestimmte Aufgaben ganz einfach lösen, andere dagegen nur schwer.

Hat man dem Computer einmal beigebracht was man von ihm erwartet, so wird er das entweder tun oder auch nicht. Er wird sich häufig beschweren, daß er die Sprache (**Syntax**) nicht oder falsch versteht. Neben solchen formalen Fehlern gibt es natürlich noch die logischen Fehler, die schon bei der Formulierung des Problems auftreten. Diese müssen natürlich vom Programmierer selbst aufgespürt werden; sie können häufig dadurch vermieden werden, daß man vor dem Programmieren sorgfältig nachdenkt. Mit den Beschwerden (und anderen Hilfsmitteln) kann man den Fehler im Programm lokalisieren und ausbügeln (**debugging**). Tut der Computer schließlich was man von ihm will, so gibt er z.B. Zahlen


```
> # Berechnung der Minima und Maxima (allg. unter Nebenbedingungen)
> readlib(extrema):
> extrema( f, {}, x, 's' ), s;
```

$$\{-\frac{1}{4}\pi\}, \{\{x=0\}\}$$

```
> zeroes:=solve( f=0, x );
```

$$zeroes := \frac{1}{2}\sqrt{2},\ -\frac{1}{2}\sqrt{2}$$

```
> # Probe: Einsetzen der Nullstellen, map wendet Funktion
> #  auf Elemente einer Liste an
> map( y->subs(x=y,f), [zeroes] );
```

$$[\arctan(0), \arctan(0)]$$

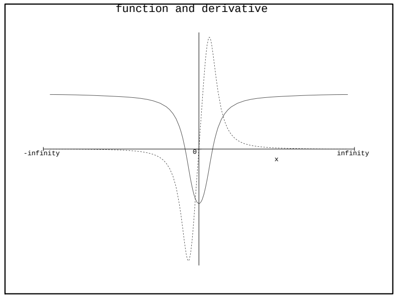
```
> # Taylorreihe um x=0 und infinity
> series( f, x=0, 15);
```

$$-\frac{1}{4}\pi + 2x^2 - \frac{8}{3}x^6 + \frac{32}{5}x^{10} - \frac{128}{7}x^{14} + \mathrm{O}(x^{15})$$

```
> series(f,x=infinity);
```

$$\frac{1}{4}\pi - \frac{1}{2}\frac{1}{x^2} + \mathrm{O}(\frac{1}{x^6})$$

```
> # Liste aus 2 Graphen die dann gemeinsam mit verschiedenen Parametern
> # dargestellt werden
> f_plot := plot(f, x=-infinity..infinity,linestyle=0,thickness=2 ):
> df_plot:= plot(df,x=-infinity..infinity,linestyle=4 ):
> plots[display](\{f_plot, df_plot\},title=‘function and derivative‘);
```



Ausdrücke, Funktionen und Prozeduren

Wir haben oben Beispiele von Anweisungszeilen der Form

```
> f := x^2 + a*x + 1;
```

```
> evalc(");
```

$$\frac{1}{2}\pi - I\ln\left(2+\sqrt{3}\right)$$

14.5 Eingebaute Funktionen und Prozeduren

An einem Beispiel sollen der Umfang und die Leistungsfähigkeit der Routinen demonstriert werden, die Maple bereitstellt. Wir werden nicht auf die Syntax aller folgenden Funktionen eingehen; Einzelheiten lassen sich am besten der online Hilfe entnehmen.

Maple kennt die Eigenschaften (Differenzierbarkeit, Wertebereich, Verzweigunsschnitte, spezielle Werte) einer riesigen Menge eingebauter Funktionen, die von den üblichen algebraischen und trigonometrischen Funktionen über unstetige Vertreter wie der Heaviside- oder der δ-"Funktion" bis hin zu Exoten wie Lamberts W-Funktion mit $W(x)*exp(W(x)) = x$ reichen.

Darüberhinaus gibt es viele Funktionen für die Manipulation von Ausdrücken, von denen wir oben bereits `subs` kennengelernt haben. Maple unterscheidet die Typen `procedure` als dem Namen einer Funktion/Prozedur und `function`, den man erhält, wenn ein bekannter oder nicht bekannter Name mit einer Argumentliste in runden Klammern versehen wird.

Hier gehen wir durch eine kurze Kurvendiskussion, um einen kurzen Eindruck von der Breite möglicher Operationen zu gewinnen.

```
> f:= arctan( (2*x^2-1) / (2*x^2+1));
```

$$f := \arctan(\frac{2\,x^2-1}{2\,x^2+1})$$

```
> raw_df := diff(f,x);
```

$$raw\_df := \frac{4\,\frac{x}{2\,x^2+1} - 4\,\frac{(2\,x^2-1)\,x}{(2\,x^2+1)^2}}{1+\frac{(2\,x^2-1)^2}{(2\,x^2+1)^2}}$$

```
> df:=simplify( raw_df );
```

$$df := 4\,\frac{x}{4\,x^4+1}$$

```
> F:= int( df, x);
```

$$F := \arctan(2\,x^2)$$

auf dem Bildschirm aus, die man dann als Grafik darstellen kann. Das Gesamtschema ist in Abb. 1.1 dargestellt.

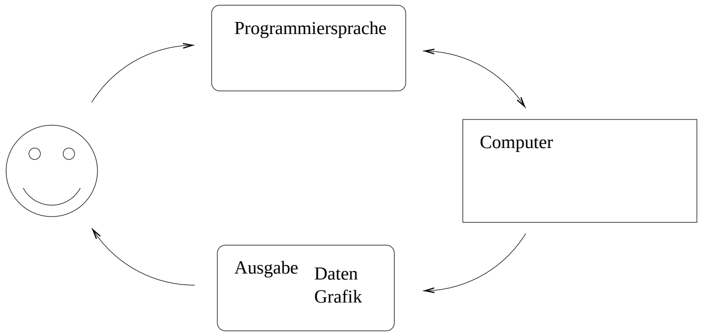


Abbildung 1.1: Schematische Darstellung der Interaktion zwischen Mensch und Computer

1.1.2 Das erste Beispiel

Wie oben erwähnt steht am Anfang das physikalische Problem. Der Einfachheit halber wählen wir einen harmonischen Oszillator, also eine Masse die an einer Feder schwingt. Ziel ist es – mit Hilfe des Computers – die Bewegung der Masse auf dem Bildschirm darzustellen.

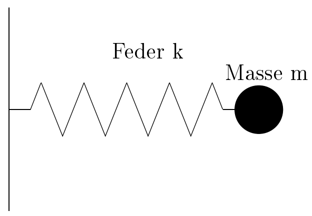


Abbildung 1.2: Harmonischer Oszillator

Zuerst muß man das physikalische Problem genau spezifizieren. Wir gehen davon aus, daß die Masse des schwingenden Körpers viel schwerer ist als die Feder (die Federmasse

kann damit vernachlässigt werden). Weiterhin sollen keine externen Kräfte, wie Gravitation und Reibung wirken. Dann folgt aus Newtons zweitem Gesetz und der Hookschen Beziehung für die Federkraft:

$$m\ddot{x} = -kx \ . \tag{1.1}$$

Dabei ist k die Federkonstante, m die Masse, x die Auslenkung der Feder und $\ddot{x}$ die zweite Zeitableitung derselben. Die Lösung dieser Gleichung ist analytisch möglich, und enthält zwei frei verfügbare Konstanten (hier die Phase δ und die Amplitude A), da es sich um eine Differentialgleichung (DGL) zweiter Ordnung handelt,

$$x(t) = A\sin(\omega t + \delta) \ . \tag{1.2}$$

Die Schwingungsfrequenz $\omega^2 = \frac{k}{m}$ ist durch die physikalischen Parameter festgelegt. Setzt man die Anfangsbedingungen $x(0) = x_0$ und $\dot{x}(0) = v_0$ in die Gleichung mit $t = 0$ ein, ergeben sich die Amplitude

$$A = \frac{x_0}{\sin\delta} \ , \tag{1.3}$$

mit der Phase

$$\tan\delta = \frac{x_0\omega}{v_0} \ . \tag{1.4}$$

Alternativ kann man auch Anfangsort und -geschwindigkeit anstelle von Amplitude und Phase verwenden.

Bis jetzt wurde die Hilfe des Computers noch nicht benötigt. Außerdem wäre es jederzeit möglich die Lösung mit Hilfe eines Taschenrechners auf Millimeterpapier zu übertragen. Trotzdem wollen wir dieses Beispiel weiterführen um schrittweise die Vorgehensweise zu verstehen.

Um eine Funktion $x(t)$, Gleichung (1.2), in einem Programm verwenden zu können muß sie diskretisiert werden, d.h. man berechnet sie zu verschiedenen Zeitpunkten t_i, genauso, wie man es machen würde, um sie auf Millimeterpapier zu zeichnen. Der Algorithmus ist in diesem Fall denkbar trivial, er lautet nämlich: *Berechne die Funktion $x(t)$ an den Stützstellen t_0, t_1, t_2, ..., t_n.* Ziel ist es nun, ein Computerprogramm zu schreiben, das genau dies tut.

```
// Binde noetige Bibliotheken ein
#include<iostream>
#include<fstream>

// Definiere das Hauptprogramm
int main()
{
```


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

```
> # alternativ haetten wir auch y den Wert Pi zuweisen koennen
> y := Pi; expression;
```

$$y := \pi$$

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

```
> # Beispiel f"ur eine fehlerhafte, rueckbezuegliche Def.
> a := b:  b := 'a':
> # Fehler in revision 3: Stack overflow: pid 26029, proc mapleV, ...
> # Fehler in revision 4: too many levels of recursion
> b;
```

Selbstkontrolle:
Warum müssen wir nach der Zuweisung `x := y` im obigen Beispiel im Ausdruck `expression` die Variable `y` substituieren und nicht `x`?
Wie hätte Maple's Antwort auf die Eingabe `a := b:  b := a: b;` gelautet?
Zu welchem Ausdruck wird `subs( 1=3, x^1 + y );` ausgewertet? Berücksichtigen Sie dabei die interne Darstellung!

Wir haben oben gesehen, daß Maple einige Namen bereits von sich aus belegt: dazu gehören die Namen interner Funktionen und Prozeduren wie `eval()` und `subs()` sowie die Konstanten `Pi` und `E`. Vorsicht ist geboten bei Namen, die griechischen Buchstaben entsprechen; der Name `pi` wird von Maple zwar als π wiedergegeben, aber intern existiert keinerlei Verknüpfung zur Konstanten `Pi`. Die Konstante `gamma` ist die Euler-Mascheroni-Konstante 0.57721 . . . .

Maple verwendet `I`, um die komplexe Einheit $\sqrt{-1}$ zu bezeichnen. Maple wertet Ausdrücke normalerweise nur dann aus, wenn die nötigen Umformungen auch über den komplexen Zahlen gültig sind. Das Ergebnis wird aber, auch wenn es komplex ist, nicht immer als komplexe Zahl dargestellt, sondern bleibt häufig als Ausdruck stehen. Um eine Zerlegung in Real- und Imaginärteil zu erreichen, benutzt man `evalc`.

```
> restart;
> exp( I * Pi +2 );
```

$$e^{(I\,\pi+2)}$$

```
> evalc(");
```

$$-e^2$$

```
> arcsin(2);
```

$$\arcsin(\,2\,)$$

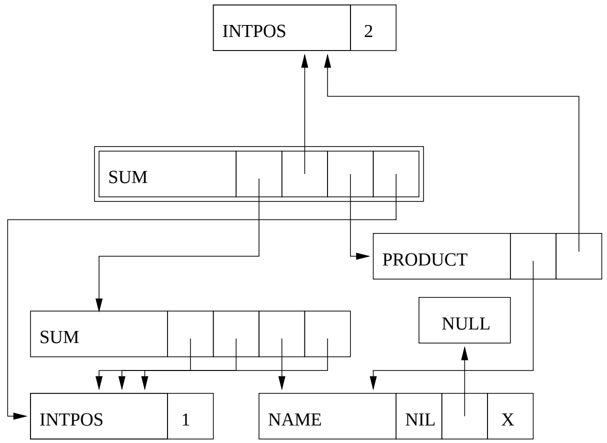


Abbildung 14.1: Interne Repräsentation des Ausdrucks $2(1+x)+x^2$ durch einen gerichteten, nichtzyklischen Graph (DAG) im Maple Kern. Summen werden aufgelöst in Summanden und die dazugehörigen Koeffizienten, sofern diese Konstanten sind. So steht beispielsweise `SUM ->(1+x) ->2 ->x^2 ->1` (doppelt umrahmt) für eine Summe mit den Bestandteilen $(1+x)$ und x^2 mit den Koeffizienten 2 und 1. Ein Produkt wird wiederum in Ausdrücke und Wiederholungsfaktoren zerlegt, die jetzt als Exponenten zu interpretieren sind. Nur wenn diese Exponenten keine Konstanten sind (was in der Abbildung nicht der Fall ist), ist es nötig, auf eine Struktur "Potenz" zurückzugreifen.

Abhängigkeiten ergeben, so wird dies in neueren Versionen als Fehler erkannt und führt in älteren Systemen zum Absturz.

```
> restart;
> # Ein Ausdruck wird mit einem Namen belegt
> expression:= sin(x) + 3*x^3 +1/x;
```

$$expression := \sin(x) + 3\,x^3 + \frac{1}{x}$$

```
> # wir ersetzen x durch y (keine Ausgabe) und werten neu aus
> x := y: expression;
```

$$\sin(y) + 3\,y^3 + \frac{1}{y}$$

```
> # subs() erlaubt das (temporaere) Ersetzen, hier von y
> subs( y=Pi, expression ) ;
```

$$\sin(\pi) + 3\,\pi^3 + \frac{1}{\pi}$$

```
> # eval erzwingt die Auswertung des neuen Ausdrucks
> eval( subs(y=Pi, expression ));
```



```
// Definiere das Feld x(t)
   double x[1000], t;   // Ausgabevariablen

// Anfangsbedingungen und Zustaende
   double A, delta;     // A=Amplitude, delta=Phasenwinkel
   double mass, kfeder; // mass=Masse, kfeder=Federkonstante
   double t_max, dt;    // t_max=Gesamtzeit, dt=Zeitintervall

// Parameter abfragen und einlesen
   cout << "Amplitude     ";  cin >> A;
   cout << "Phasenwinkel  ";  cin >> delta;
   cout << "Masse         ";  cin >> mass;
   cout << "Federkonstante";  cin >> kfeder;
   cout << "Gesamtzeit    ";  cin >> t_max;
   cout << "Zeitintervall ";  cin >> dt;
   double omega=sqrt(kfeder/mass);
// Nicht ANSI-Konform - besser waere:
// std::cout und std::cin anstelle von cout und cin

// Schleife von t=0 bis t=t_max
   t=0.0;
   for( int i=0; i<=t_max/dt; i++ ) // Schleife
   {
        x[i]=A*sin(omega*t+delta);  // Berechne die Funktion
        t=t+dt;                     // Gehe zur naechsten Zeit
   }

// Ausgabe
   ofstream outfile("harmon.dat");       // Oeffne Datei
   t=0.0;
   for( int i=0; i<=t_max/dt; i++ ) // Schleife
   {
      outfile << t << " " << x[i] << "\n";  // Ausgabezeile
      t=t+dt;                               // Gehe zur naechsten Zeit
   }
// Ende des Hauptprogramms
   return 0;
}
```

Die Bedeutung der vielen Symbole wird in den folgenden Kapiteln ausführlicher erklärt werden. Zum Schreiben des Programms benützt man einen Editor (Textverarbeitung) wie z.B. `xedit`, `emacs` oder `xemacs`. Das geschriebene Programm kann man dann als Datei `harmon.cc` abspeichern, wobei die Endung `.cc` anzeigt daß es sich um ein C++ Programm handelt. Der Computer kann mit dem Programm alleine noch nichts anfangen, man muß es erst in Maschinensprache übersetzen, d.h. man *compiliert* es. Dies geschieht mit dem

Kommando `g++ harmon.cc -o harmon`, das die *ausführbare Datei* `harmon` erzeugt. Diese wird durch das Kommando `harmon` aktiviert und fragt dann nacheinander die Amplitude, den Phasenwinkel, die Masse, die Federkonstante, die Gesamtzeit und das Zeitintervall der Stützstellen ab. Hat man diese eingegeben, so erzeugt das Programm die Datei `harmon.dat`, mit z.B. folgendem Inhalt (bei Verwendung von $A = 1$, $\delta = 0$, $m = 1$, $k = 10$, Ausgabedauer $t_{max} = 1.5$ und Zeitschritt $dt = 0.1$):

```
0    0
0.1  0.310984
0.2  0.591127
0.3  0.812649
0.4  0.953581
0.5  0.999947
0.6  0.947148
0.7  0.800422
0.8  0.574318
0.9  0.291259
1   -0.0206835
1.1 -0.330575
1.2 -0.607684
1.3 -0.824528
1.4 -0.959605
1.5 -0.999519
```

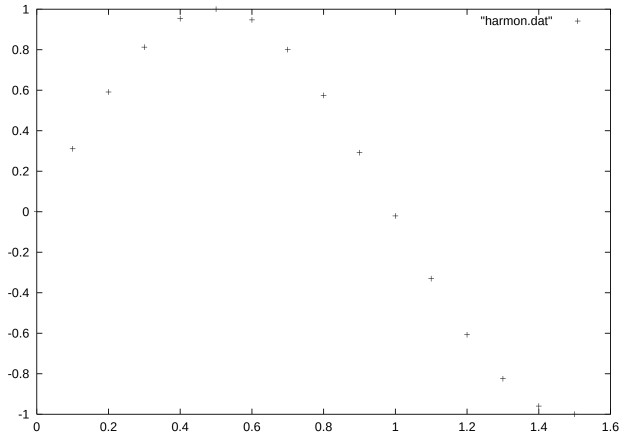


Abbildung 1.3: Grafische Darstellung der zweispaltigen Zahlenreihe auf dem Bildschirm. Die erste Spalte definiert die x-, die zweite die y-Koordinate der Punkte.

Diese Zahlenkolonne (2 Spalten) kann man nun mit dem Programm `gnuplot` auf den Bildschirm zeichnen. Nach Eingabe des Kommandos `gnuplot` erscheint eine Meldung des Programms und der sog. *Prompt*: '`>`'. Dann gibt man den Befehl
`plot "harmon.dat"` ein und wird die Grafik, wie in Abb. 1.3, auf dem Bildschirm erscheinen sehen. Dieses Beispiel kann zusammen mit anderen in der ersten Übung ausprobiert werden.

1.2 Programmier-Philosophie

Man kann verschiedene Programmiersprachen für ein Problem verwenden. Je nach Entwicklungsstufe und Spezialisierungsgrad sind die Sprachen mehr oder weniger benutzerfreundlich. Manche Sprachen eignen sich besonders für spezielle Aufgaben. Es ist im Prinzip egal, welche Sprache man verwendet, jedoch erspart die richtige Wahl unter Umständen Zeit und Mühe.


```
> evalf("",25);
```

1.053571428571428571428571

14.3 Hilfe, Dokumentation online

Eines der wichtigsten Hilfsmittel zum Umgang mit Maple ist die interaktiv verfügbare Hilfe. Man erreicht sie im Kommandozeilenmodus durch Voranstellen eines Fragezeichens vor die Funktion, für die Hilfe benötigt wird, z.B. `?exp`. Ein Index läßt sich über `?index` erreichen.

In Fenstersystemen gibt es zusätzlich leistungsfähige Suchmechanismen, die in der Regel erlauben, die benötigte Funktionalität in kurzer Zeit aufzufinden (*Topic Search, Full Text Search*).

14.4 Variablen, Zuweisung und Auswertung

Jedes sinnvolle Objekt kann in Maple mit einen Namen verbunden werden Die Repräsentation eines solchen Objektes im Speicher beinhaltet den Typ, Speicherbereich für die interne Darstellung der Zeichenkette sowie einen Zeiger, der auf einen eventuellen "Wert" zeigt. In Abb. (14.1) ist als Beipiel die interne Darstellung eines polynomialen Ausdrucks gezeigt. Ein bisher unbekannter Name in einem Ausdruck dient als Variable in dem Sinne, daß Maple versucht, Operationen so durchzuführen, daß für alle möglichen Einsetzungen korrekte Ergebnisse resultieren.

Dieser Wert kann jetzt ein beliebiger anderer Name, eine Zahl, eine Funktionsvorschrift oder ein anderer Maple-Datentyp sein. Ein Wert wird in einer Zuweisung festgelegt, die entweder durch den Operator `:=` oder funktional durch `assign()` durchgeführt wird. Zur internen Repräsentation wird ein Objekt angelegt, daß der linken Seite entspricht und dann im Namensobjekt ein Zeiger auf diese Struktur eingetragen.

Achtung: Das Symbol `=` alleine trennt linke und rechte Seite einer mathematisch aufzufassenden *Gleichung*, die selbst wieder ein Objekt darstellt, das zugewiesen werden kann. Der auf der linken Seite einer Zuweisung stehende Ausdruck wird dabei von Maple vor der Zuweisung zunächst soweit möglich ausgewertet. Um eine Auswertung zu verhindern, benutzt den linksgerichteten Akzent `'` oder `evaln()`; um eine Auswertung zu erzwingen, stellt Maple `eval()` bereit.

Bei der Auswertung eines Ausdruckes folgt Maple dabei soweit wie möglich den Zeigern, die die verschiedenen Objekte miteinander verknüpfen. Falls sich dabei schleifenförmige

keine Information verlorengeht. Falls Arithmetik erforderlich ist, so wird diese mit beliebiger, bzw. bei numerischen Rechnungen mit einer vorher festgelegten (fast) beliebigen Genaugkeit ausgeführt. Der Hauptnachteil besteht in dem gegenüber rein numerischen, hardwareorientierten Ansätzen sehr großen Bedarf an Speicherplatz und Rechenzeit.

Der folgende Text beabsichtigt nur, in die Grundideen von Maple einzuführen, die häufig auch Entsprechungen in den Konzepten der anderen Programme haben, die sich dann ebenfalls leicht erlernen lassen sollten. Komplexere Aufgabenstellungen sollten mit einem der eingangs erwähnten Büchern und der im Programmpaket enthaltenen interaktiven Hilfefunktionalität erschließbar werden.

14.2 Eingabe

Maple arbeitet zeilenorientiert. Es gibt daher sowohl eine kommandoorientierte Version, die sich durch `maple` aufrufen läßt, als auch eine Version mit grafischer Benutzeroberfläche, die unter UNIX als `xmaple` startet. Deren wesentlicher Vorteil ist eine bessere Editierbarkeit der Kommandos und natürlich die viel besseren Möglichkeiten, Grafiken darzustellen. Der rein algorithmische Kern ist bei beiden Versionen gleich. Die aktuelle Maple Version ist V, Release 5. Die folgenden Beispiele sind mit Release 3 und 4 gerechnet worden.

Die Eingabesyntax ist ähnelt denen vieler prozeduraler Programmiersprachen mit den üblichen Operatoren, Assoziativitäts-, Kommutativitäts und Präzedenzregeln.

Im interaktiven Betrieb wertet Maple alle Eingaben hinter dem Prompt `>` aus, sobald sie durch ein Semikolon `;` oder einen Doppelpunkt `:` beendet werden. Der Doppelpunkt unterdrückt dabei die Ausgabe des berechneten Ergebnisses. Maple führt Rechnungen soweit möglich, symbolisch aus, und geht erst dann zu numerischen Rechnungen über, wenn (i) es mit `evalf` (**eval**uate by **f**loating point computation) dazu aufgefordert wird, oder (ii) der zu berechnende Ausdruck eine bereits numerisch ermittelte Zahl enthält, wozu ein Punkt in der Zahldarstellung ausreicht. Kommentare werden durch `#` eingeleitet und enden am Zeilenende.

```
> restart;
>  3 /7 + 5/8;
```

$$\frac{59}{56}$$

```
> # mit  "  bezieht man sich auf das jeweils letzte Ergebnis,
> # zwei "" bezeichnen das vorletzte Ergebnis, usw.
> evalf(");
```

$$1.053571429$$

Eine Programmiersprache wie C++ hat die denkbar größte Komplexität und Flexibilität, und findet heutzutage die breiteste Anwendung bei Programmierern. Bei Stellenangeboten werden sehr häufig Kenntnisse in C++ gefordert. Außerdem sind viele immer wieder vorkommende Aufgaben mit existierenden *Tools* leicht zu lösen. Aus diesen Gründen wird im folgenden die Syntax von C++ verwendet.

Unabhängig von der Syntax (dem Vokabular) der Programmiersprache C++ kann man jedoch Aussagen zum Programmierstil machen. Wenn das Ziel ist, ein Problem schnellstmöglich zu lösen, wird man zwar kurzfristige Erfolge erzielen, langfristig aber feststellen, daß viele Nachteile gegen eine “quick and dirty” Methode sprechen. Der Programmcode ist schon nach wenigen Tagen unverständlich geworden, weil man nicht von der Möglichkeit gebraucht macht, **Kommentare** einzubauen. Deshalb ist das Programm nahezu unverständlich für andere und ist dadurch nicht wiederverwendbar. Man kann die schnelle Methode also als “write-only” bezeichnen.

Ziel soll es jedoch sein “write-and-read” Programme zu schreiben, die man auch im nächsten Monat noch verstehen und benutzen kann (wenn man es für eine neue Übungsaufgabe benötigt). Dazu gehört eine konsistente Schreibweise, die man sich leider erst durch Erfahrung angewöhnt. Deshalb bietet es sich an, sich ein System auszusuchen und dieses beizubehalten. Ein optisch übersichtliches Programm hilft bei Änderungen und beim Verstehen ebenso wie häufige kurze Kommentare. Man sollte auch Dinge kommentieren, die im Moment klar erscheinen, denn schon bald kann das nicht mehr der Fall sein.

1.3 Geschichte der Programmierung mit C++

C wurde in den 70er Jahren von K. Thompson, B. Kernighan und D. Ritchie zusammen mit dem Betriebssystem UNIX eingeführt. Es war als typisierte Sprache zur strukturierten Programmierung konzipiert. Der ANSI Standard wurde 1988 als ANSI-C festgelegt.

C++ ist als Erweiterung von C zu sehen, die zwischen 1983 und 1985 von B. Stroustrup propagiert und eingeführt wurde. Neben dem Ziel, eine objektorientierte Sprache zu erschaffen, war auch eine Abwärtskompatibilität zu C erwünscht. Deshalb ist C als Teilmenge von C++ anzusehen. Der ANSI Standard für C++ wurde erst im Juni 1998 festgelegt. Dieses Skript hält sich noch nicht ganz daran, da auch die meissten benutzten Computer noch nicht ANSI-C++ “sprechen”. Im Rahmen dieser Vorlesung werden wir C++ als ein “besseres” C verwenden und einige dubiose Eigenschaften von C vermeiden.

Gründe für die Verwendung von C++ sind Systemunabhängigkeit, Erweiterbarkeit, Wiederverwendbarkeit und Robustheit. Die Bedeutung dieser Begriffe wird aber erst klar, wenn man länger mit der Sprache arbeitet.

Ziel dieser Vorlesung ist es nicht, perfekt C++ zu programmieren! Dazu gibt es Spezialkurse und Bücher, auf die unten verwiesen wird. Stattdessen soll erlernt und vertieft werden, wie man physikalische Fragestellungen erkennt, in ein Programm formuliert und schließlich mit Hilfe des Computers beantwortet.

1.4 Anwendungsbeispiele

Im Laufe der Vorlesung wird nach einer
• Einführung in die Programmierung (Syntax C++),
• die numerische Differentiation, Integration und Interpolation,
• die Lösung von Differentialgleichungen (Pendel, Planeten),
• der dichtlineare Oszillator, Chaos, dann
• Zufallszahlen, Monte-Carlo Methoden, Gittermodelle und
• Fouriertransformation (FFT)
behandelt.

1.5 Literatur

Als weiterführende Literatur empfehlen wir die *Manuals des Rechenzentrums* für C, C++ und UNIX, die sich durch ein hervorragendes Preis-Leistungs Verhältnis auszeichnen. Ein Nachschlagewerk für numerische Methoden ist das Buch *Numerical Recipes* von William H. Press, Saul A. Teukolsky, William T. Vetterling und Brian P. Flannery, Cambridge University Press. Daneben gibt es mehrere – häufig englischsprachige – Werke, wie z.B.:
○ S. E. Koonin, Physik auf dem Computer, Oldenburg Verlag
○ H. Gould, I. Tobochnik, Introduction to Computer Simulation Methods, Addison Wesley
○ D. W. Heermann, Computer Simulation Methods, Springer
○ W. Kinzel, G. Reents, Physics by Computer, Springer

Kapitel 14

Computeralgebra mit Maple

14.1 Einführung

Gute Einführungen in Maple sind André Heck, *Introduction to Maple*, Springer Verlag sowie Michael Kofler, Maple V Release 4, Einführung und Leitfaden für den Praktiker, Addison Wesley. Weiterhin gibt es einige brauchbare Kurzeinführungen, die auf dem Internet verfügbar sind. Als Startpunkt der Suche empfiehlt sich `www.maplesoft.com`, ein Rechner, der von den Maple-Entwicklern unterhalten wird.

Maple ist eines der am weitesten entwickelten Computeralgebra-Programme. Es bietet einen weiten Vorrat an Symbolmanipulationsroutinen, numerischen Algorithmen, flexiblen Daten- und Kontrollstrukturen für die Programmierung, grafische Ausgabemöglichkeiten und die Möglichkeit der Protokollierung der Sitzungen in sogenannten Arbeitsblättern *(worksheets)*.

Ganz ähnliche Funktionalität wie Maple bieten Mathematica und Macsyma, andere weitverbreitete Computeralgebra-Programme, die nicht auf bestimmte Problemklassen spezialisiert sind. Es sollte noch REDUCE erwähnt werden, das weiterhin gepflegt wird und als LISP Quellcode verfügbar ist. REDUCE ist wie Macsyma ein "Veteran" und stammt bereits aus den 60'er Jahren. Die auf LISP beruhenden Programme haben den Ruf, mit Systemresourcen nicht sehr sparsam umzugehen, was insbesondere bei umfangreichen symbolischen Operationen Probleme machen kann.

Die "modernen" Systeme wie Mathematica und Maple setzen auf einem algorithmischen Kern aus C Routinen auf, in dem die Grundfunktionalität implementiert ist. Diese Systeme sind durch benutzerdefinierte Routinen erweiterbar, die entweder zur Laufzeit interpretiert werden oder in einem kompakten, schnell lesbaren internen Zwischencode vorliegen.

Der Hauptvorteil der Computeralgebra liegt darin, daß bei symbolischer Manipulation

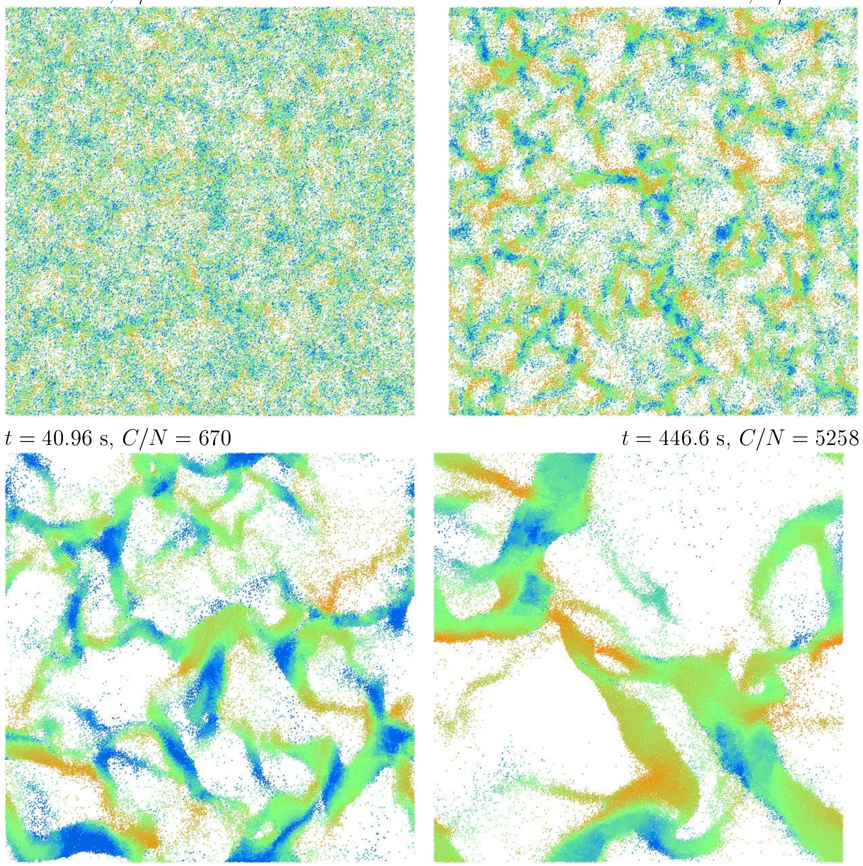


Abbildung 13.5: Momentaufnahmen einer Simulation mit $N = 79524$ Teilchen in einem System der Länge $L = 1000\,a$ mit einer Dichte $\rho = 0.25$. Materialparameter waren $r = 0.8$ und $\beta = -1$. Die Farbskala gibt die maximale (rot), mittlere (grün) oder minmale Energie einzelner Teilchen an.

Kapitel 2

Grundlagen von C++

Wie am Beispielprogramm in der Einführung zu sehen, besteht ein Programm aus den unterschiedlichsten Elementen und Konstrukten. In den folgenden Abschnitten werden nach und nach die grundlegenden Sprachelemente vorgestellt.

2.1 Sprachkonstrukte

Das erste Konstrukt sind *Kommentare*. Sie haben keinen Einfluß auf den Programmablauf (oder den Computer), sondern sollten lediglich zum besseren Verständnis des Programms eingefügt werden. Als Kommentar wird jener Text angesehen, der rechts der Zeichen '`//`' steht.

Das zweite wesentliche Element sind *Daten*, also z.B. die physikalischen Parameter die man dem Computer zur Berechnung gibt, Konstanten wie die Zahl π, oder Indices $i = 1, ..., n$, die zum Zählen verwendet werden. Man muß dem Computer zuerst mitteilen, welche Daten, Variablen und Konstanten man benötigen wird.

Operatoren und *Funktionen* sind nötig, um Daten zu manipulieren. Beispiele für Operatoren sind die Grundrechenarten, Beispiele für Funktionen sind trigonometrische Funktionen wie Sinus und Cosinus. Funktionen sind nicht nur Sprachkomponenten, sondern können auch vom Programmierer dazu verwendet werden, häufig wiederkehrende Manipulationen zusammenzufassen und dadurch übersichtlich ins Programm einzubauen.
Wichtig ist, daß das sog. Hauptprogramm im Prinzip ebenfalls eine Funktion ist, die immer `main` genannt werden muß. Der Beginn und das Ende einer Funktion sind durch '{' und '}' gekennzeichnet.

Die daneben wichtigsten Sprachelemente sind diejenigen für die Kommunikation zwischen Benutzer und Computer, kurz die der *Ein- und Ausgabe*, die häufig auch mit der aus dem

Angelsächsischen abgeleiteten Abkürzung I/O für Input/Output belegt werden. Im Sprachumfang von C/C++ sind genormte Bibliotheksfunktionen enthalten, die die Ausgabe auf den Bildschirm oder in Dateien regeln. Ebenso wird die Eingabe per Tastatur, Maus oder andere angeschlossene Geräte (externe Meßgeräte, Trackball, etc.) behandelt. Dies wird durch spezielle Bibliotheken erledigt, die je nach der vorhandenen Rechnerhardware und -software verschieden sein können und daher auch nicht allgemein behandelt werden können.

Wie man sieht, sind Dateien eine zweigleisige Möglichkeit mit einem Programm zu kommunizieren und Daten sowohl ein- als auch auszugeben.

Zuletzt seien noch *Organisations*-elemente erwähnt, die nicht direkt zur Programmiersprache gehören, sondern von einer Vorstufe des Compilers verarbeitet werden, dem sog. Präprozessor. Diese Organisationselemente erlauben es, unübersichtliche Programmelemente auszulagern oder Programmteile einfach auszublenden.

Das einfachste denkbare Programm gibt den Text “Hello World!” auf den Bildschirm aus:

```
#include <iostream>
using std::cout;

int main()
{
   cout << "Hello World!\n" ;  // print a string to the screen
   return 0;
}
```

- Die erste Zeile `#include <iostream>` holt die Definitionen aus der Standard-Bibliothek. Diese ist nötig für die Ein- und Ausgabe von Daten und Text. Diese Zeile muß am Anfang eines jeden Programms stehen, das I/O Operatoren wie z.B. ‘`cin`’, ‘`cout`’, ‘`<<`’ oder ‘`>>`’ enthält.
- Die Anweisung `using std::cout` importiert das Symbol `cout` aus dem Namensraum `std`. Namensräume werden in C++ benutzt, um Algorithmen in verschiedenen Bibliotheken voneinander zu trennen. Ein `cout` aus anderen Namensräumen (evtl. selbst definierte) kann nun nicht mehr verwendet werden. Wir werden häufig auf die C++-Standardbibliothek zurückgreifen, die den Namensraum `std` benutzt. Zur Benutzung von C-Funktionen ist keine `using`-Direktive nötig.
- `main()` ist eine Funktion vom Typ `int`. Jedes Programm besteht aus zumindest dieser einen Funktion, kann aber auch aus einer beliebigen Anzahl zusammengesetzt sein. Dabei verliert `main()` allerdings nie seine besondere Rolle.
- `return 0` gibt den Wert 0 an das aufrufende Programm, also in diesem Fall an das Betriebssystem zurück. Man kann von Null verschiedene Werte verwenden um Fehler oder Erfolg der Funktion anzuzeigen.

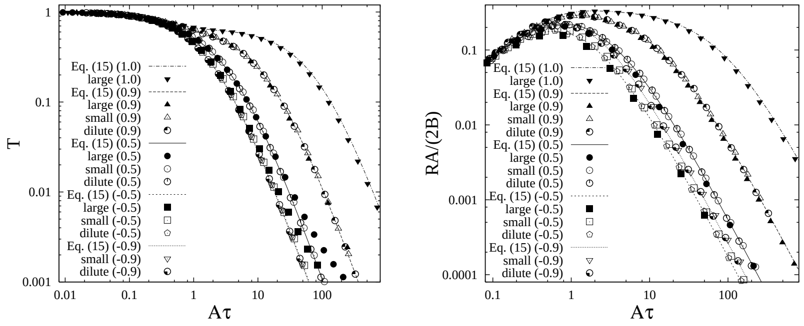


Abbildung 13.4: (Links) Translationsenergie T als Function der skalierten Zeit $A\tau$ mit $\tau \propto t$. Verschiedene Symbole gehören zu unterschiedlichen Simulationen mit $N = 99856$ Teilchen und Dichte $\rho = 0.25$ (large), $N = 198$, $\rho = 0.25$ (small) und $N = 198$, $\rho = 0.01$ (dilute). Als Materialparameter wurden $r = 0.99$ und die in der Figur angegebenen Werte für β verwendet. Die Linien geben die numerische Lösung von T der Gleichungen 13.6 und 13.7 an. (Rechts) Skalierte Rotationsenergie $RA/(2B)$ als Function von $A\tau$. Die Simulationen sind aus Abb. 13.4 entnommen, und die Linien geben die numerische Lösung von $RA/(2B)$ an.

Beobachtet man insbesondere die großen Simulationen genauer, so stellt man für größere Zeiten Abweichungen zwischen Simulation und Theorie fest. Dies kommt daher, daß bei der Herleitung der Gleichungen 13.6 und 13.7 die Homogenität des Systems angenommen wurde. In den Simulationen ergeben sich allerdings Situationen, in denen diese Annahme falsch wird. In Abb. 13.5 sind Momentaufnahmen einer typischen Simulation dargestellt. Es entwickeln sich aus dem homogenen Anfangszustand Fluktuationen der Dichte, sog. Cluster von Teilchen entstehen. In diesem Fall wird die einfache theoretische Beschreibung ungültig und man müßte detailliertere Betrachtungen anstellen.

Mit diesem letzten Beispiel einer Anwendung verschiedener Methoden auf eine Fragestellung wurde nochmals die Bedeutung des Computers für die Physik klar. Interessante, komplexe Probleme sind in verschiedenen Niveaus der Modellierung häufig nur mit Hilfe des Computers zu behandeln.

13.3.2 Exakte, Diskrete Lösung des Problems

Alternativ kann das Problem des dissipativen, getriebenen Gases mit Rotationsfreiheitsgraden natürlich auch direkt mit Hilfe der Molekulardynamik behandelt werden. Unter der Annahme daß Stöße wesentlich kürzer andauern als die Zeit zwischen den Stößen, kann man die Geschwindigkeit der Teilchen nach dem Stoß durch die Materialparameter und die Geschwindigkeiten vor dem Stoß ausdrücken. Mit der Teilchenmasse m, dem Radius a und dem Trägheitsmoment $I = qma^2$ mit $q = 2/5$ sind kugelförmige Teilchen spezifiziert. Die Materialparameter r und β beschreiben dann den Energieverlust beim Stoß. $r = 1$ und $\beta = 1$ bedeutet elastisch mit Ankopplung der Rotation, $\beta = -1$ vermeidet die Ankopplung des Rotationsfreiheitsgrads. Alle Werte $r < 1$ oder $-1 < \beta < 1$ führen zu Energieverlust. Die Geschwindigkeiten $\boldsymbol{v}$ und die Winkelgeschwindigkeiten $\boldsymbol{\omega}$ nach einem Stoß sind

$$\boldsymbol{v}'_\mu = \boldsymbol{v}_\mu - \frac{1+r}{2}\boldsymbol{v}_n - \frac{q(1+\beta)}{2q+2}(\boldsymbol{v}_t + \boldsymbol{v}_r) \ , \tag{13.8}$$

$$\boldsymbol{v}'_\nu = \boldsymbol{v}_\nu + \frac{1+r}{2}\boldsymbol{v}_n + \frac{q(1+\beta)}{2q+2}(\boldsymbol{v}_t + \boldsymbol{v}_r) \ , \tag{13.9}$$

$$\boldsymbol{\omega}'_\mu = \boldsymbol{\omega}_\mu + \frac{1+\beta}{a(2q+2)}[\hat{\boldsymbol{r}} \times (\boldsymbol{v}_t + \boldsymbol{v}_r)] \quad \text{und} \tag{13.10}$$

$$\boldsymbol{\omega}'_\nu = \boldsymbol{\omega}_\nu + \frac{1+\beta}{a(2q+2)}[\hat{\boldsymbol{r}} \times (\boldsymbol{v}_t + \boldsymbol{v}_r)] \ , \tag{13.11}$$

wobei μ und ν die Stoßpartner identifiziert. Die Relativgeschwindigkeit wurde dabei in einen Normal- ($\boldsymbol{v}_n$), einen Tangential- ($\boldsymbol{v}_t$) und einen Drehanteil ($\boldsymbol{v}_r$) aufgeteilt, und $\hat{\boldsymbol{r}}$ bezeichnet den Einheitsnormalenvektor am Teilchenkontakt.

Die Parameter A, B und C aus den Gleichungen 13.6 und 13.7 haben die Bedeutung von Translations- (A) und Rotationsenergieverlust (C) bzw. quantifizieren die Kopplung (B) der beiden Freiheitsgrade. In einem zweidimensionalen System kann man zeigen daß

$$A = \frac{1-r^2}{4} + \frac{\eta}{2}(1-\eta), \tag{13.12}$$

$$B = \frac{\eta^2}{2q}, \tag{13.13}$$

$$C = \frac{\eta}{2q}\left(1 - \frac{\eta}{q}\right) \tag{13.14}$$

$$\eta = q(1+\beta)/(2q+2) \tag{13.15}$$

ist, wie im MAPLE Programm des letzten Abschnitts bereits eingesetzt wurde. Zur Physik sei zu bemerken, daß die Zeit in den Zeitentwicklungsgleichungen mit einer typischen Kollisionsrate skaliert wurde, die von Systemparametern wie z.B. der Dichte abhängt.

Führt man nun unterschiedliche Simulationen mit “beliebigen” Parametern durch, so wird man feststellen, daß die Lösung der diskreten Simulation in vielen Fällen *exakt* mit der theoretischen Vorhersage übereinstimmt, siehe Abb. 13.4.

- Die Klammern `{}` schließen das Hauptprogramm, eine Funktion, oder allgemeiner, Programm-*Blöcke* ein. Zu jeder Beginn-Klammer `{` gehört eine Ende-Klammer `}`.
- Die Klammern `()` umschließen eventuelle Argumente, die an `main` übergeben werden.
- Die Anführungszeichen `"..."` umschliessen Text (Feld von `char`’s).
- Die Zeichenkombination ‘`\n`’ führt bei der Ausgabe zu einer neuen Zeile.
- `cout` bezeichnet das Standard-Ausgabe Medium, im Normalfall also den Bildschirm. Im ANSI Standard würde man an dieser Stelle `std::cout` verwenden.
- Die Zeichenkombination `<<` wird als Ausgabeoperator benutzt und gibt alles rechts von ihr stehende “nach links”, auf `cout`, also auf dem Bildschirm aus.
- *Jede* Anweisung muß mit einem Strichpunkt ‘`;`’ beendet werden, denn eine (optisch) neue Zeile im Programm wird vom Compiler nicht als neues Programmelement erkannt. Man trennt einzelne Befehle also durch ‘`;`’, sollte aber vermeiden, dies innerhalb einer Zeile zu tun. Das Semikolon hat die Funktion der Trennung zweier Anweisungen, daher steht es nicht am Ende von *Anweisungs*blöcken, die durch geschweifte Klammern `{}` gekennzeichnet sind.

2.2 Kommentare

Kommentare werden durch `//` eingeleitet und erstrecken sich bis zum Ende der Zeile. Zeilenumbrüche werden ansonsten wie Leerzeichen behandelt und haben keine syntaktische Relevanz. Die C-Kommentarsyntax `/* comment */` gilt auch weiterhin in C++. `/* */` Kommentare lassen sich nicht schachteln, und sind daher zum “Auskommentieren” größerer Programmbereiche ungeeignet. Falls nämlich im auskommentierten Programmbereich bereits ein C-Kommentar vorhanden gewesen sein sollte, so wird die erste schliessende `*/`-Kombination den gesamten Kommentarbereich beenden und vermutlich dazu führen, daß ein Teil des Programmes wieder eingeblendet wird.

Große Programmbereiche sollten mit Präprozessordirektiven ausgeblendet werden (gekennzeichnet durch `#`). Auch sollte man C++-Kommentare in Zeilen vermeiden, die Präprozessordirektiven enthalten, da der Präprozessor aus C-Zeiten möglicherweise nur C-Kommentarsyntax korrekt behandelt.

```
#if 0      /* get rid of the main program for the moment */
int main()
{           // main is THE keyword, the program starts here
   for ( int i=0 /* loop variable */; i< 10 ; i++ )
   {                                           // for loop
```

```
      ;                                  // do nothing else
   }                                    // end of the for loop
}
#endif                  /* end of getting rid of main... */
```

2.3 Datentypen, Variablen, Konstanten

2.3.1 Grundlegende Datentypen und Deklaration

Die grundlegenden, eingebauten Datentypen von C++ sind `bool, char, int, float, double`. Diese Datentypen werden besonders effizient vom Computer bearbeitet.

Der Typ `bool` wird für die Resultate von Vergleichsoperationen (2.5.3) benutzt. Er kann die Werte `true` oder `false` besitzen. Weiterhin ist er frei in den `int` Datentyp umwandelbar, wobei `true` einer 1 und `false` einer 0 entspricht. Diese Umwandlung stellt die Aufwärtskompatibilität zu alten Programmen her, die statt `bool` einfach `int`'s verwenden. Einige einfache Deklarationen sind:

```
  bool    test;                 // declare test
  test  =(0 < 5);               // comparison with result true
                                // test is set to true
  int     i(test);              // i is declared and set to 1
```

Die Initialisierung/Konstruktion einer Variablen kann in der Form *type variable = type_constant* oder äquivalent durch *type variable(type_constant)* erfolgen. Das Gleichheitszeichen an dieser Stelle ist keine Zuweisung, da die Variable gerade erst ins Leben gerufen wird und nicht bereits im Programm existiert.

Der Typ `int` ist der Datentyp, der auf der jeweiligen Maschine vom Compilerschreiber als der für die Verarbeitung integraler Daten als "geeignetster" empfunden wurde. Über seine Größe ist im Standard nichts festgelegt. Mit Hilfe des `sizeof` Operators kann man seine Größe in Einheiten der Größe eines `char` erfragen, der gleichzeitig der kleinste Datentyp ist. Ein `char` wurde vom Compilerschreiber als der für Einzel(schrift)zeichen geeignete Typ gewählt. Verschiedene Rechner unterstützen noch weitere ganzzahlige Datentypen, z.B. neben 32-bit `int` auch 16 oder 64-bit große Variablen. Diese lassen sich ggf. Deklaration von `short` und `long` oder gar (nicht ANSI konform) `long long` ansprechen. Nach diesen Schlüsselworten darf `int` fehlen. Man beachte, daß eine Integer-Variable, immer einen ganzzahligen Wert hat – auch wenn man ihr eine reelle Zahl zuweist. Beispiele für `int`eger Deklarationen sind:

```
  bool    test;                   // declare test
```


Die Anregung führt zu einer Oszillation der Translationsenergie, die sich zeitverzögert auch auf die Rotationsenergie überträgt. Schaltet man die Anregung ab $F = 0$ und aktiviert einen Energieverlust der Translationsenergie $A = 9.41/49$, mit $B = C = 5/49$, so ergibt sich das in Abb. 13.3 (Rechts) dargestellte Verhalten. Da die Translationsenergie ständig dissipiert wird nimmt die Energie stetig ab, die Rotationsenergie stellt sich allerdings auf einem höheren Niveau ein. Das Verhältnis der beiden Energien ist *nicht* gleich, es gilt also nicht das Equipartitionsprinzip.

Zur Lösung des gekoppelten Gleichungssystems wurde wieder MAPLE verwendet

```
> A:=(1-r_n**2)/4 + eta/2*(1-eta);
> B:=eta**2/2/q;
> C:=eta/2/q*(1-eta/q);
> gamma_1:=A-B**2/C;
> eta:=q*(1+r_t)/(2*q+2);
--------------------------------------------------------------------------------
> r_n:=0.8; r_t:=1; q:=2/5;
  delta:=0; f_D:=0; T_D:=2;
  F:=f_D * T_D ** (1-delta);
--------------------------------------------------------------------------------
> A; B; C;
                                    941/4900
                                      5/49
                                      5/49
--------------------------------------------------------------------------------
> de1:=diff(T(t),t)=(-A*T(t)**(3/2)+B*T(t)**(1/2)*R(t))
                                    +F*T(t)**delta *(sin(t/5));
> de2:=diff(R(t),t)=(B*T(t)**(3/2)-C*T(t)**(1/2)*R(t));
--------------------------------------------------------------------------------
> Tnum2:=dsolve( {de1, de2, T(0)=1, R(0)=0}, {T(t), R(t)}, type=numeric );
--------------------------------------------------------------------------------
> Tnum2(5);
         [t = 5, T(t) = .335720336830467481, R(t) = .283818365163101605]
--------------------------------------------------------------------------------
> writeto(xxx);
  for tt from 0 by 1 to 100 do
  lprint( tt, evalf(rhs(Tnum2(tt)[2]),8) ,
            evalf(rhs(Tnum2(tt)[3]),8) ):
  od;
  writeto(terminal):
```

diesen Zweck und erzeugt die Datei `xxx` mit den Ergebnissen. Mit dieser einfachen Gleichung kann man nun versuchen die Produktion bezüglich des Aufwands $I + R$ zu optimieren und Entscheidungsregeln für die Koeffizienten K und R einführen. In Abb. 13.2 wird für einen Fall negative Produktion erreicht (Zusammenbruch), während sich in den anderen Fällen ein positives Gleichgewicht einstellt.

13.3 Parallele Anwendung verschiedener Methoden

13.3.1 Mean-Field Kontinuumsbeschreibung

Ein häufig auftretendes Problem ist die Beschreibung von zwei (oder mehr) gekoppelten Systemen. In diesem Beispiel wird ein getriebenes Gas beschrieben, bei dem sowohl Rotations- als auch Translationsfreiheitsgrade angeregt sind. Die Translationsfreiheitsgrade gehorchen der Differentialgleichung

$$\frac{\partial T}{\partial t} = -AT^{3/2} + BT^{1/2}R + FT^{\delta}\sin(t/t_0) \ , \tag{13.6}$$

während die Zeitentwicklung der Rotationsfreiheitsgrade durch

$$\frac{\partial R}{\partial t} = BT^{3/2} - CT^{1/2}R \tag{13.7}$$

beschrieben werden. Zur physikalischen Bedeutung der Parameter A, B und C sei auf den nächsten Abschnitt bzw. [SL et al. Phys. Rev. E 58, 3416-3425, 1998] verwiesen. Mit den Parametern $A = B = C = 5/49$, $F = 0.2$, $\delta = 0$, $t_0 = 5$, sowie den Anfangswerten $T(0) = R(0) = 1$ ergibt sich das in Abb. 13.3 (Links) dargestellte Verhalten.

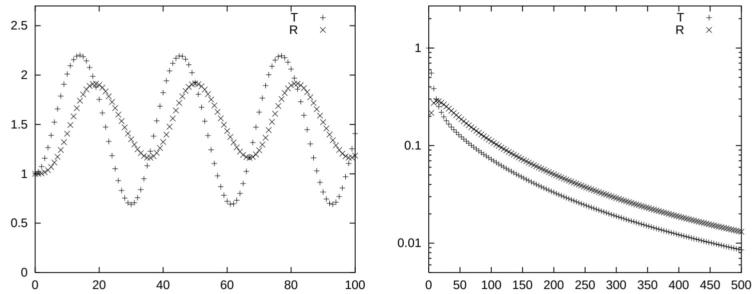


Abbildung 13.3: Translations- und Rotationsenergie in einem periodisch getriebenen, elastischen Gas (Links) und in einem ungetriebenen inelastischen Gas (Rechts).


```
test  =(0 < 5);                  // (see above)
int     i(test);                 // i is declared and set to 1
int     j=0;                     // j is declared and set to 0
long    l, n, m;                 // l, n, and m are declared "long"
short   s1, s2;                  // s1 and s2 are declared "short"
cout << sizeof(test) << " "
     << sizeof(i) << " "
     << sizeof(l) << " "
     << sizeof(s1) << "\n";     // check size of: test, i, l, s1
                                 // Note: this is not guaranteed
                                 // on my machine I get: 8 4 8 2
                                 // ANSI should give:    1 (for bool)
i = 37;
j = 3.5674;                      // ATTENTION
cout << i << " " << j << "\n";  // this leads to: 37 3
```

`float` und `double` sind Gleitkommatypen einfacher und doppelter Genauigkeit, die zu numerischen Zwecken geeignet sind. `long` kann zur Modifikation von `double` verwendet werden, um vierfache Genauigkeit zu erhalten (falls diese unterstützt sein sollte). Die exakte Gleitkommarepräsentation ist abhängig von der Hardware-Unterstützung. Es ist heute (1998) üblich, für doppeltgenaue Gleitkommazahlen zumindest die Grundrechenarten innerhalb der CPU in Rechnerhardware durchzuführen. Einfachgenaue Zahlen müssen dazu erst in doppeltgenaue gewandelt werden. Aus diesem Grund und der allgemein hohen Anforderungen numerischer Rechnungen sollte man `float` nur noch dort verwenden, wo dies durch z.B. Speicherplatzbeschränkungen erfordert wird.

C/C++ legt nichts über die wirkliche Repräsentation der Variablen im Speicher fest. Insbesondere ist der Speicherbedarf einer Variablen nicht definiert, es gelten aber immer die Beziehungen:

`1 = sizeof(char) = sizeof(bool)` $\leq$ `sizeof(short)` $\leq$ `sizeof(int)` $=$ `sizeof(unsigned)` $\leq$ `sizeof(long)`

und

`sizeof(float)` $\leq$ `sizeof(double)` $\leq$ `sizeof(long double)` .

Im allgemeinen sind auf UNIX-Arbeitsplatzrechnern `int`-Variablen 32, `short` 16 und `char` 8 bit lang. Doppeltgenaue `double`s erfordern 64 bit, `float` 32 bit. Die wirklichen Werte können jedoch im Prinzip sogar von Compiler zu Compiler verschieden sein!

Die folgenden Beispiele deklarieren verschiedene Datentypen.

```
#include <string>
using std::string;
```

```
char      c1 = 'x';        // one character: 'x'
char    c2[] = "hello";    // text array with 5 characters
                           //   (note: length is 6 due to added '\0')
string    str("hello");    // the C++ way, arbitrary length strings
                           //   (note: length is 5 - no added '\0')
char      c3 = '\n';       // the new-line character
int       i_var;           // integer variable with name i_var
long int  il_var = 1L;     // long constant
short     is_var;          // short integer variable (int per default)
double    d_var = 34.2;    // real number 34.2 with double precision
```

Bei `char` Variablen ist zu beachten, daß einzelne Zeichen mit einem einfachen `'` eingeschlossen werden können, längere Zeichenketten jedoch mit dem doppelten `"` begrenzt werden müssen. Die Variablen `c1` und `c3` haben beide die Länge Eins, wohingegen die Zeichenkette `"hello"` im “Feld” `c2` abrufbar ist und die Länge Sechs hat, da automatisch ein Zeichenkettenende `'\0'` angefügt wird. (Die Definition `char c2 = 'hello';` führt zu einer Fehlermeldung). Flexibler ist die C++ Methode, `string`s zu benutzen, die den Zeichenkettenterminator `'\0'` nicht benötigen.

Variablendeklaration sind in C++ nicht auf den Anfang eines Blockes beschränkt (wie in C), sondern können überall erfolgen, wo die Variable erstmals an einer Stelle auftaucht und wo auf sie zugewiesen werden kann. Wichtig ist insbesondere die Deklaration innerhalb einer `for(;;)` Schleife

```
double a;
cin >> a;              // input a via keyboard
int    i_trunc = a;    // throw away the fractional part
for ( int i=0; i < i_trunc; i++ )
{
   ...
}
```

Im obigen Programm ist `i` außerhalb der `for(;;)` Schleife (d.h. nach der letzten zugehörigen geschlossenen Klammer, oder der letzten Anweisung des Schleifenkörpers) nicht mehr existent.

Zusammenfassung
(i) Deklarationen können an fast beliebiger Stelle im Programm stehen
(ii) Initialisierung erfolgt in `()` oder `=` Form; es findet *keine Zuweisung* sondern eine Initialisierung statt!
(iii) `sizeof(type)` kann eingesetzt werden, wenn die Größe eines Typs im Programm bekannt sein muß, dies ist für die Portabilität zwischen verschiedenen Rechnern interessant.

13.2 Ökonomie und Wirtschaft

Ein vereinfachtes Beispiel aus der Wirtschaftstheorie betrifft die Berechnung des zukünftigen Gewinns. Die jährliche Produktion sei x mit einem Anfangswert x_0. Interessiert man sich für die jährliche Änderung der Produktion $\dot{x}$, so muß man Investitionen, Rationalisierung und Marktsättigung berücksichtigen. Die Neuinvestitionen I hängen nicht von x ab, da auch, oder vor allem bei niedriger Produktion investiert werden muß. Rationalisierung führt zu einer Produktionssteigerung proportional zur Produktion, ergibt also eine Änderung Rx. Schließlich kann der Markt nur eine bestimmte Menge an Produktion auffangen, so daß die Produktion zurückgenommen werden muß wenn die Marktkapazität erreicht wird, was durch einen Term $-Cx^3$ beschrieben werden kann.

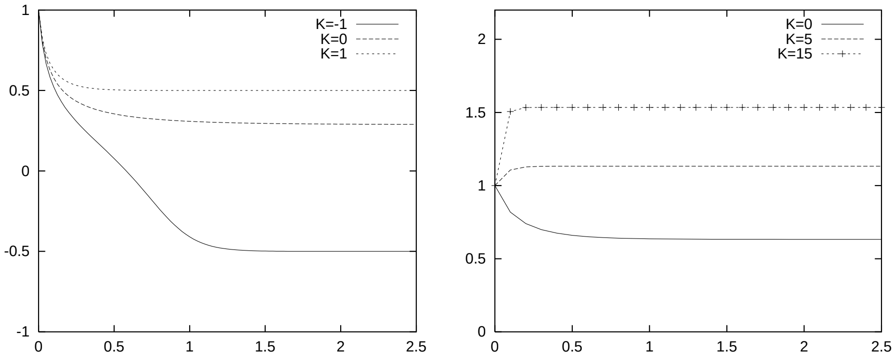


Abbildung 13.2: Lösung einer Produktivitäts-Entwicklungsgleichung für $R = 1$, $C = 12$ und verschiedene K =-1, 0, 1 (Links) und $R = 2$, $C = 5$ und verschiedene K =0, 5, 15 (Rechts).

Insgesamt erhält man so die Differentialgleichung

$$\dot{x} = I + Rx - Cx^3 \; , \tag{13.5}$$

die man mit einem der Standardverfahren aus Kapitel 3 numerisch lösen kann. Ebensogut erfüllt das MAPLE script

```
> restart;
> de1:=diff(T(t),t)=(K+R*T(t)-C*T(t)**3);
> K:=0: R:=1: C:=12: Tnum1:=dsolve( {de1, T(0)=1}, T(t), type=numeric );
> writeto(xxx);
> for tt from 0 by 25 to 2500 do
   lprint( evalf(tt/1000,4), evalf(rhs(Tnum1(tt/1000)[2]),8)  ):
  od;
  writeto(terminal):
```

```
    ipos=nstep +int((double(rand())/p-0.5)*4);
    xdist[i]=ipos;
    xn[ipos]++;
  }

  double time=0.;
  double ww, wp, wm;
  int iww;

  for( int iloop=1; iloop<=nloop; iloop++ ){
    for( int iw=1; iw<=nwalk; iw++ ){
      iww=int(double(rand())/p*nwalk);
      wp= w+sigma*exp((+double(q))/theta
                 - abs(q)*double(xdist[iww]-nstep)/nwalk/nstep);
      wm=-w+sigma*exp((-double(q))/theta
                 + abs(q)*double(xdist[iww]-nstep)/nwalk/nstep);
      ww=wp/(wp+wm);
      time+=1./(wp+wm);
      if( double(rand())/p > ww ){
         if( xdist[iww]>0 ){
            xn[xdist[iww]]-=1;
            xdist[iww]-=1;
            xn[xdist[iww]]+=1;
            q-=1;
         }
      }
      else{
         if( xdist[iww]<2*nstep ){
            xn[xdist[iww]]-=1;
            xdist[iww]+=1;
            xn[xdist[iww]]+=1;
            q+=1;
         }
      }
    }
    cout << time << ' ' << q << ' ' << ' ' << ' ' << '\n';
    std::ofstream outfile1("dens.dat");
    for(int i=0; i<=2*nstep; i++ ){
       outfile1 << iloop << ' ' << i << ' ' << xn[i] << '\n';
    }
    outfile1.close();

  }
  return 0;
}
```


2.3.2 Variablenattribute

Neben den Attributen `short`, `long` können ganzzahlige Variablen noch durch `signed` und `unsigned` modifizert werden, die andeuten, daß entweder ganze ($\in \mathcal{Z}$) oder natürliche Zahlen ($\in \mathcal{N}_0$) repräsentiert werden sollen. Gleichermaßen kann auch hier das `int` der Typdeklaration entfallen.

2.3.3 Lebensdauer und Lage der Variablen im Systemspeicher

`auto` Falls keine weitere Angabe erfolgt, sind Variablen `auto`matic. Dies bedeutet, daß sie angelegt werden, sobald das Programm über die Zeile läuft, in der sie deklarariert sind. Sobald das Programm den Block verlässt, in dem sie deklariert sind, wird der zugehörige Speicher wieder freigegeben. Der Block ist im allgemeinen das innerste die Variable umschließende Paar geschweifter Klammern, oder wie im Fall der in den ()-Bereichen von Anweisungen wie `for(;;)` deklarierten Variablen, das Ende dieser Anweisung. In diesem Fall wird der Speicherplatz freigegeben, sobald diese Anweisung vollständig abgearbeitet ist. Variablen dieser Art werden vom Compiler meist im sogennanten Stack-Bereich des Programmes angelegt.
Der Inhalt einer solchen Variable ist zufällig, wenn sie nicht explizit im Programm initialisiert wurde. Da dies zu subtilen Fehlern führen kann, sollte man solche Variablen immer entweder durch eine explizite Initialisierung auf einen passenden Wert setzen oder möglichst unmittelbar vor der ersten Verwendung deklarieren (so daß sichtbar ist, daß die Variable angemessen – z.B. in einer Eingabe – verwendet wird).

`static` Das Schlüsselwort `static` bewirkt, daß eine Variable während der gesamten Programmlebensdauer existiert. Diese Variablen werden vom Compiler vor Programmbeginn meist auf dem heap-Speicher angelegt und, sofern nicht anders angegeben, auf den Wert 0 initialisiert. Eine `static` Variable innerhalb einer Funktion hat daher den Inhalt 0, wenn die Funktion das erste Mal angesprungen wird und behält zwischen Funktionsaufrufen den jeweils zuletzt zugewiesenen Wert. Eine automatische Variable wird hingegen jedes Mal neu angelegt und ggf. initialisiert. Die folgende Funktion zählt, wie häufig sie aufgerufen wurde, und gibt diesen Wert zurück:

```
int f(){
  static int count;  // 0 at first function call
  return ++count;    // increment before use as return value
                     // (i.e. returns one on the first call)
}
```

`const` Variablen, die `const` erklärt werden, dürfen nach der Initialisierung nicht mehr verändert werden. Dies kann dazu dienen, in einen Namensraum Konstanten einzuführen. Der Compiler kann diese u.U. bereits zur Compilezeit wegoptimieren. Diese Technik ist der unter C weitverbreiteten Gewohnheit, solche Ersetzungen durch den Präprozessor machen zu lassen, vorzuziehen, da der Präprozessor keine Typenüberprüfung vornehmen kann. Die konsequente Benutzung von `const` erlaubt dem Compiler weitere Optimierungen, so daß man Variablen und Funktionsargumente sorgfältig daraufhin überprüfen sollte, ob es möglich, sie als `const` zu erklären.

```
const int    Size = 100;
const double Pi   = 3.1415926;   // When including cmath on can
                                 // use the predifined constant M_PI
```

`volatile` dient dazu, den Compiler darauf hinzuweisen, daß eine Variable ihren Wert ändern kann, auch ohne daß eine Zuweisung oder ein anderweitiger Programmzugriff erfolgt ist. Dies Verhalten ist typisch für hardware-Register, die im Adreßraum des Programmes liegen. Der Compiler muß in diesem Fall von zu agressiver Optimerung abgehalten werden.

2.3.4 Felder

Felder dienen der Zusammenfassung mehrerer Variablen gleichen Typs in einem zusammenhängenden Speicherbereich. Auf die einzelnen Elemente des Feldes kann mit Variablen eines ganzzahligen Typs als Index zugegriffen werden. Sie erlauben einen direkten Zugriff auf große Datenmengen und werden in Programmiersprachen verwendet, um z.B. Matrizen zu implementieren, viele Daten gleichen Typs und gleicher Bedeutung zu speichern und z.B. einfache Such-Operationen zu ermöglichen. Häufig (aber nicht immer) ist das Vorkommen einer indizierten Variable in einer mathematisch-formalen Beschreibung eines Problems ein Hinweis darauf, daß diese in einem Computerprogramm durch ein Feld repräsentiert werden kann.

Felder werden in C/C++ mittels `[]` deklariert. Gleichzeitig dient dieser Operator auch zum Zugriff auf Feldelemente. Felder beginnen in C/C++ immer mit dem Index 0, der letzte gültige Index hat daher den Wert der um Eins verringerten Größe des Feldes. Mehrdimensionale Felder werden durch mehrfache, nicht verschachtelte `[]`-Paare deklariert.

Felder haben in C/C++ *feste* Größen, d.h., daß die in Klammern stehende Feldgröße ein Ausdruck sein muß, der nur Konstanten oder zum Zeitpunkt der Compilation feststehende `const` deklarierte Variablen umfaßt.

```
  #include <string>
```

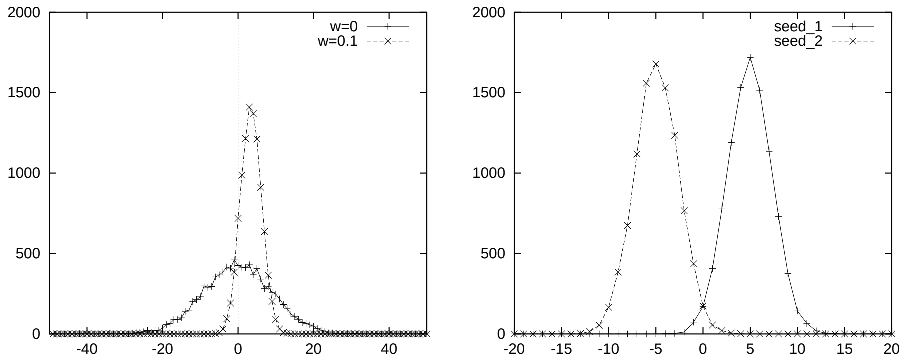



Abbildung 13.1: (Links) Werbung w führt zu einer Polarisierung (und zu einer Fokussierung) der Meinungen. (Rechts) Verschidene, zufällige Faktoren können zu einer entgegengesetzten Polarisierung der Meinungen führen.

Diese Ergebnisse wurden mit folgendem C++ Programm erstellt.

```
#include<iostream>
#include<fstream>
#include<cmath>
using std::cout;

int main()
{
  const int nstep=50, nwalk=9870, nloop=101;
  // double w=0, sigma=.5, theta=1.e5;
  double w=.1, sigma=.5, theta=1.e6;
  int xn[2*nstep+1];
  int xdist[nwalk+1];
  int p=RAND_MAX;

  // srand(129456);
  // srand(123456);
     srand(129456);

  for(int i=0; i<2*nstep+1; i++ ){  // set fields to zero
    xn[i]=0;
  }

  int ipos, q=0;
  for(int i=0; i<nwalk; i++ ){      // set fields to zero
```

Änderung der Größen n_+ und n_- zur Folge hat. Objektive Information, oder subjektive Quasi-Information wie Werbung, werden den Zustand in eine Richtung verschieben, und soziale Interaktion mit Personen der gleichen (oder der anderen) Meinung kann einen ähnlichen Effekt haben.

Um den Prozess zu modellieren kann man die sog. *Master-Gleichung* aus Kapitel 5.5 verwenden:

$$\Delta n(x,t) = -n(x,t)[W_- + W_+] + n(x-1,t)W_+ + n_(x+1,t)W_- \tag{13.1}$$
$$\tag{13.2}$$

wobei W_- (W_+) die Wahrscheinlichkeit ist daß jemand innerhalb einer Zeitspanne Δt seine Meinung von seinem Standpunkt etwas von pro in Richtung contra (contra zu pro) wechselt. Die Summen $n_+ = \sum_{i=1}^{\infty} n(x)$ und $n_- = \sum_{i=-1}^{-\infty}$ quantifizieren dann das Wahlergebnis zur Zeit t, wobei n_0 Stimmenthaltungen vorkommen können. Das eigentliche Problem ist die Wahl der Übergangswahrscheinlichkeiten W_- und W_+.

Nehmen wir an, daß die Lobby der +-Gruppe mehr Werbung betreibt als die Anhänger der Meinung −. Diese Informationen werden zu einer Drift w zu Gunsten von n_+ im Term W_+ führen, und gleichzeitig mit umgekehrtem Vorzeichen im Term W_- auftauchen. Weiterhin sollte soziale Interaktion zu einer Verstärkung der globalen Meinung führen, wenn man viel mehr Anhänger der einen Fraktion trifft kann dies zu einem Meinungswechsel führen, der durch die Größe $q = (n_+ - n_-)$ im Term W_+ quantifiziert wird.

Um der Verschiedenheit der einzelnen Individuen Rechnung zu tragen muß man nun eine Wahrscheinlichkeit einführen, daß jemand überhaupt gewillt ist seine Meinung zu ändern. Bei einer leichtgläubigen oder unentschiedenen Person wird bereits ein schwacher Versuch zu einem Umschwung führen, während eine "sture" Person kaum jemals die Meinung ändert. Um das Problem einfach zu halten nehmen wir unabhängige Individuen, also eine exponentiell abfallende "Sturheit" an, womit

$$W_+ = w + \sigma \exp\left(\frac{q}{\theta} - |q|\frac{x}{x_{max}}\right) \tag{13.3}$$

ist, mit der typischen Toleranz θ und der Rate σ. Die Exponentialfunktion führt dazu, daß die Änderungsrate kleiner wird, wenn n_+ größer wird. Entsprechend ergibt sich

$$W_- = -w + \sigma \exp\left(-\frac{q}{\theta} + |q|\frac{x}{x_{max}}\right) \tag{13.4}$$

mit dem umgekehrten Effekt. Der Term $|q|\frac{x}{x_{max}}$ wurde eingeführt um zu berücksichtigen, daß extreme Meinungen nur bis zu einem gewissen Grad x_{max} auftreten können.

In Abb. 13.1 sind der Einfluß des Parameters w und auch die Zufälligkeit des Meinungsbildungsprozesses dargestellt.


```
using std::string;

// array of 20 int's
int  a[20];

// array of three strings, initialized to some values:
string  stra2[] = { "Matthias", "Stefan", "Oliver" };

// two dimensional array of 20x30 elements of int type
int  ia[20][30];
// initialization in a for loop
for ( int i=0; i < 20; ++i )
  for ( int j=0; j < 30; ++j )
    ia[i][j] = i * j;
```

Wir werden später noch sehen, daß zwischen Feldern und den Zeigervariablen (Pointern) in C/C++ ein sehr enger Zusammenhang besteht, der uns an der Stelle auch erlauben wird, "dynamische" Felder anzulegen, d.h. solche, deren Größe erst während der Laufzeit des Programmes, zum Beispiel durch eine Benutzereingabe, ermittelt wird und ggf. auch dynamisch geändert werden kann.

2.4 Kontrollstrukturen

Für die Umsetzung von Algorithmen in einer bestimmten Programmiersprache ist es essentiell, daß diese Sprache Kontrollstrukuren aufweist, die in der Lage sind, bestimmte Anweisungsblöcke nur unter bestimmten Bedingungen oder mehrfach zu durchlaufen. Die in C (und C++) meistverwendeten Kontrollstrukturen sind das `if`-Statement zur bedingten Ausführung und die `for`-Schleife zur Wiederholdung von geblockten Anweisungen. Daneben existieren noch die abweisenden `while()`- und die nichtabweisenden `do{...}while()` Schleifen, das `switch()...case:`-Statement, das aufgrund des Inhalts einer Variablen einen auszuführenden Block anspringt und die unbedingte Sprunganweisung `goto`, die sich manchmal sinnvoll dazu verwenden läßt, aus tief verschachtelten Schleifen herauszuspringen.

2.4.1 if-Anweisung

Die `if`-Anweisung dient dazu, bestimmte Programmteile nur unter bestimmten Bedingungen zu durchlaufen. Eine solche Anweisung kann, muß aber keinen `else`-Teil enthalten, der durchlaufen wird, wenn der Bedingungsteil `false` ist. Der `else` Teil kann aus

beliebig vielen `else if` und einem `else` bestehen. Sowohl `if`, `else if`, als auch `else`-Zweige können entweder einzelne Anweisungszeilen (abgeschlossen durch Semikolon) oder in Klammerpaare `{}` gesetzte Anweisungsblöcke sein. Ein `else`-Block bezieht sich immer auf die gerade vorstehende, aktuelle `if`-Anweisung. Im folgenden Beispiel ist die zweite `if`-Anweisung ein einzelnes Statement im `else`-Teil der ersten.

```
#include<iostream>
using std::cout;
#include<cmath>
// ...

  // solution of quadratic equations  x^2 + p x + q = 0

  double p= 2., q= 3.;            // or other values
  double sol1, sol2;
  double p= 2., q= 3.;            // or other values
  double sol1, sol2;
  double root_arg = 0.25*p*p - q; // calculate the argument for the root

  if ( root_arg > 0. ) { // begin of if-block
    sol1 = 0.5*p + sqrt(root_arg);
    sol2 = 0.5*p - sqrt(root_arg);
  }                               // end if-block
  else if ( root_arg < 0. )       // 'error' condition, 'else' part
    cout << "no real solution exists";
  else                            // refers to last if(root_arg < 0.)
    sol1 = sol2 = 0.5 * p;        // assignment from right to left

  if ( sol1 > 0. )
    cout << "graph cuts positive x-axis";
                                  // ok, else branch missing
```

2.4.2 Ternärer ?:-Operator

Als Einschub und der Vollständigkeit halber soll hier der ternäre Operator `?:` behandelt werden. Ternär bedeutet, daß der Operator drei Argumente benötigt. Dieser kann dazu dienen, innerhalb von Ausdrücken Teilausdrücke bedingt auszuwerten.

```
inline int abs(int number) {
  // compute absolute value
  return number > 0 ? number : -number;
```

Kapitel 13

Anwendungsbeispiele

Im Rahmen dieser Vorlesung wurden bereits Anwendungsbeispiele aus den Bereichen

- Astronomie (Himmelsmechanik),
- Biologie (Populationsdynamik, Wachstum, neuronale Netze),
- Informationstechnik (Signalverarbeitung, neuronale Netze) und
- Logistik (Handelsreisenden-Problem)

vorgestellt. In diesem letzten Kapitel soll die Vorlesung mit weiteren Fragestellungen aus anderen Fachgebieten abgerundet werden.

13.1 Soziologie

Eine im Zusammenhang mit Menschen meßbare Größe ist die Meinung (so diffus dieser Begriff auch scheinen mag). Meinung kann z.B. durch Meinungsumfragen relativ genau "gemessen" werden und die Qualität einer repräsentativen Meinungsumfrage, beispielsweise bei Wahlen, ist oft erstaunlich gut.

Am Beispiel der einfachen Frage nach pro + und contra − zu einem bestimmten Thema soll die Dynamik der Meinungen genauer untersucht werden. Die Wahrscheinlichkeit eine Person zu finden, die dafür bzw. dagegen ist sei n_+ bzw. n_-, wobei $n_+ + n_- = 1$. Allerdings sind die Meinungen nicht binär: jemand kann nur knapp überzeugt sein, oder ein fanatischer Anhänger der einen Seite. Die Meinung zur Frage kann durch verschiedene Prozesse oder Aktionen positiv oder negativ beeinflußt werden, was entsprechende

(allerdings sind in realistischen Systemen viel mehr verschiedene Spezies vorhanden) und wird neben den in diesem Kapitel vorgestellten Methoden auch direkt mit Monte-Carlo Simulationen aus Kapitel 5 untersucht.

12.3 Hyperbolische partielle Differentialgleichungen

Eine hyperbolische Gleichung ist die Wellengleichung der lineare Kette des letzten Kapitels. Der Unterschied im Vorzeichen der zweiten Ableitungen führt zu komplexen Lösungen bei Ansetzen einer Exponentialfunktion, also zu Schwingungen [13.2]. Die Gleichung

$$\frac{\partial^2 u(x,t)}{\partial t^2} = c^2 \frac{\partial^2 u(x,t)}{\partial x^2} \tag{12.18}$$

kann wie gehabt mit dem Zeitschritt Δt und der Gitterkonstante h diskretisiert werden zu

$$\frac{1}{\Delta t^2}\left[u(x,t+\Delta t) - 2u(x,t) + u(x,t-\Delta t)\right] = \frac{c^2}{h^2}\left[u(x+h,t) - 2u(x,t) + u(x-h,t)\right] \ . \tag{12.19}$$

Aufgelöst nach der gesuchten Größe $u(x, t + \Delta t)$ ergibt sich

$$u(x,t+\Delta t) = 2[1-b]u(x,t) + b[u(x+h,t) + u(x-h,t)] - u(x,t-\Delta t) \ , \tag{12.20}$$

wobei $b = (c\Delta t/h)^2$ dimensionslos ist. Diese Gleichung erinnert stark an das Verlet Integrationsschema aus Kapitel 3.2.1. Mit diesem Verfahren kann man nun die Dynamik einer Kette aneinandergekoppelter Massen untersuchen.

12.4 Weiterführende Literatur

Das Gebiet der partiellen Differentialgleichungen ist bei weitem zu umfangreich um im Rahmen dieser Vorlesung auch nur annähernd erfaßt werden zu können. Die hier vorgestellten Methoden stellen nur die Spitze des Eisbergs dar. Für tiefergehende Studien seien besonders die Bücher [13.1] und [13.2] empfohlen.

[13.1] S. E. Koonin and D. C. Meredith, *Computational Physics*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.
[13.2] H. Gould and J. Tobochnik, *An Introduction to Computer Simulation Methods*, Addison-Wesley Publishing Company, 1996.


```
}
```

Ist der vor dem Fragezeichen stehende Ausdruck `true`, so wird der Teilausdruck vor dem Doppelpunkt, sonst derjenige nach dem Doppelpunkt ausgewertet. Der resultierende Typ der beiden alternativen Ausdrücke muß gleich sein. Aufgrund der geringen Priorität dieses Operators und zur Verdeutlichung der Zusammengehörigkeit der Teilausdrücke sollte man sich angewöhnen, großzügig Klammern zu setzen.

```
int i1   =       3 > 4 ? 0 : 1;    //  i1   is 1
int i2   = 3 * ( 3 > 4 ? 0 : 1 );  //  i1_3 is 3
int i3   = 3 *   3 > 4 ? 0 : 1;    //  i2_3 is 0, since * binds
                                   //     stronger than >
```

Dieser Operator kann immer durch eine `if`-Anweisung vermieden werden, kann aber in vereinzelten Situationen klarer sein. Da er anderenfalls eine der zahlreichen Möglichkeiten in C/C++ ist, seine Programme für andere erfolgreich unlesbar zu gestalten, sollte man ihn sparsam einsetzen.

2.4.3 (do) while-Schleifen

Schleifen, die mit `while` gebildet werden, werden so oft durchlaufen, bis die Bedingung im Bedingungsteil der Anweisung nicht mehr erfüllt ist. Sollte diese Bedingung bereits falsch sein, wenn das Programm auf die `while`-Schleife trifft, so wird der Schleifenkörper nicht durchlaufen.

```
char c=0;                        // 'loop initialization'
while( c != 'y' && c != 'n' )  // first time c==0, so we enter the loop
  std::cin >> c;               // can also be a block in { }
```

Das obige Beispiel erfordert die Initialisierung der Zeichenvariablen mit einem Wert, so daß die Schleife auf jeden Fall durchlaufen wird. In solchen Fällen ist häufig die Verwendung einer nicht abweisenden Schleife mit `do ... while(...)` intuitiver:

```
char c;
do {
  std::cin >> c;
} while( c != 'y' && c != 'n' );
```

2.4.4 for-Schleifen

Wie sich oben bereits andeutete, enthalten Schleifen neben der Abbruchbedingung häufig noch einen Schleifeninitialisierungsteil und möglicherweise Anweisungen, die den Wahrheitsgehalt der Schleifenbedingung beeinflussen und jedesmal während eines Durchlaufes ausgeführt werden.

Bei der Arbeit mit Feldern benötigt man häufig Schleifen, die die Feldelemente einzeln ansprechen. Solch eine Schleife, programmiert mit `while`, könnte etwa so aussehen:

```
const int Size = 20;
// ..
int a[Size];
// ..
int i=0;                // loop initialization
while ( i < Size ) {    // loop condition, rejecting if false
  a[i] = i;
  i++;                  // loop control, performed after each pass
}
```

Zur Abkürzung läßt sich in diesem Falle `for(;;)` verwenden. Die folgende Konstruktion mit `for` ist der obigen (bis auf den Sichtbarkeitsbereich der Variablen `i`) vollständig äquivalent, aber übersichtlicher und kürzer.

```
int a[20];
// ..
for( int i=0; i < 20; i++ )
  a[i] = i;            // work on elements 0 .. 19
```

Diese Formulierung besitzt zwei Vorteile. Erstens ist die Schleifenkontrolle in den drei durch Semikolon getrennten Teilen der `for`-Anweisung gut sichtbar zusammengefaßt und nicht über den Schleifenkörper verstreut. Zweitens ist der Verwendbarkeitsbereich der Variablen `i` auf die Schleifenkontrollstruktur und den Schleifenkörper beschränkt.

2.4.5 break und continue

Aus einer einzigen nicht verschachtelten Ebene einer mittels `for` oder `while` gebildeten Schleife kann man durch die Benutzung von `break` herausspringen. Die Schleifenbearbeitung wird abgebrochen und das Programm direkt nach der Schleife fortgesetzt, ohne noch weitere Kontrollanweisungen oder Schleifentests zu beachten. Das `break` kann also z.B. dazu dienen, Schleifen frühzeitig abzubrechen oder "Endlosschleifen" zu verlassen.

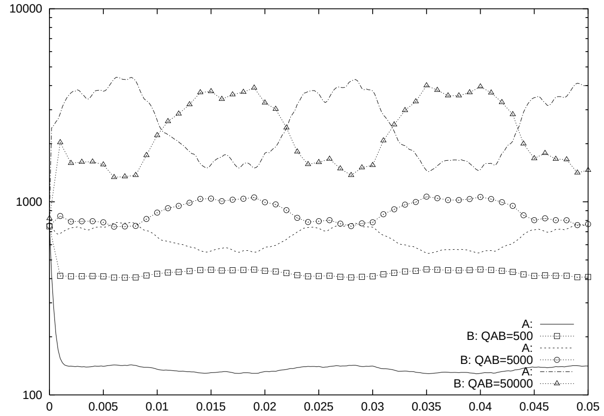


Abbildung 12.3: Lösung einer zeitabhängigen Diffusionsgleichung für zwei Spezies mit Erzeugungsrate Q_{AB} und Reaktionswahrscheinlichkeit $K_{AB} = 10^4$.

Das hier beschriebene Lösungsverfahren ist allerdings sehr einfach und ungenau. Bei komplizierteren Fragestellungen ist man auf genauere Verfahren angewiesen, die komplizierter zu implementieren sind. Trotzdem kann man mit diesem einfachen Verfahren bereits interesante physikalsiche Probleme bearbeiten. Bei einem vereinfachten Katalysator kann man annehmen, daß zwei Spezies A und B (z.B. Sauerstoff und Kohlenmonoxid) auf der Oberfläche diffundieren. Bei Kontakt reagieren A und B zu AB Molekülen (Kohlendioxid) und verschwinden vom System. Man löst dann die Diffusionsgleichung für beide Spezies und führt außerdem noch Zufuhr- und Reaktionsterme ein.

$$\frac{\partial \phi_A}{\partial t} = \left[\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right] \phi_A + Q_A(t) - K_{AB}\phi_A\phi_B \ , \qquad (12.16)$$

$$\frac{\partial \phi_B}{\partial t} = \left[\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right] \phi_A + Q_B(t) - K_{AB}\phi_A\phi_B \ , \qquad (12.17)$$

Die Felder ϕ_A und ϕ_B stehen dabei für die Dichte der einzelnen Spezies und der Quell-Term $Q(t)$ wird mit Zufallszahlen beschickt. Der Reaktionsterm hängt von der Dichte beider Spezies an einem Ort ab und hat die Form $K_{AB}\phi_A\phi_B$, wobei K_{AB} die Reaktionsrate ist. Beide Teilchenarten haben die gleichen Eigenschaften und existieren Anfangs zufällig verteilt auf dem Gitter. Die Lösung des resultierenden Gleichungssystems ist in Abb. 12.3 für verschiedene Erzeugungsraten Q_{AB} dargestellt. Man stellt Oszillationen des Systems fest, deren Periode mit zunehmender Zufuhrrate stark abnimmt. Obwohl die Zufuhr im Mittel für beide Spezies gleich ist kann eine Spezies über die andere siegen. Wenn z.B. von Spezies A viele Exemplare auf dem Gitter sind während nur wenige von Spezies B existieren, so werden all frisch zugeführten Teilchen von Spezies B sofort mit A reagieren und wieder verschwinden. Deshalb kann bei schwacher Zufuhr die Übermacht einer Spezies lange vorhalten. Diese Fragestellung ist Gegenstand aktueller Forschung

```
  return 0;
}
```

In Abb. 12.2 ist die zeitabhängige Lösung für vier Zeitpunkte dargestellt. Diese Problem eignet sich gut zum Test des Programms, da die Summe aller ϕ_{ij} erhalten bleiben sollte. Wählt man den Zeitschritt zu groß (oder hat man einen Fehler einprogrammiert) so wird die Summe in der Regel nicht erhalten bleiben. In Abb. 12.2 stellt man fest, daß die Lösung (die bei $t = 0$ nur in der Mitte den Wert $\phi_{20\,20} = 2000$ hatte) mit der Zeit immer weiter auseinanderläuft und gleichzeitig flacher wird.

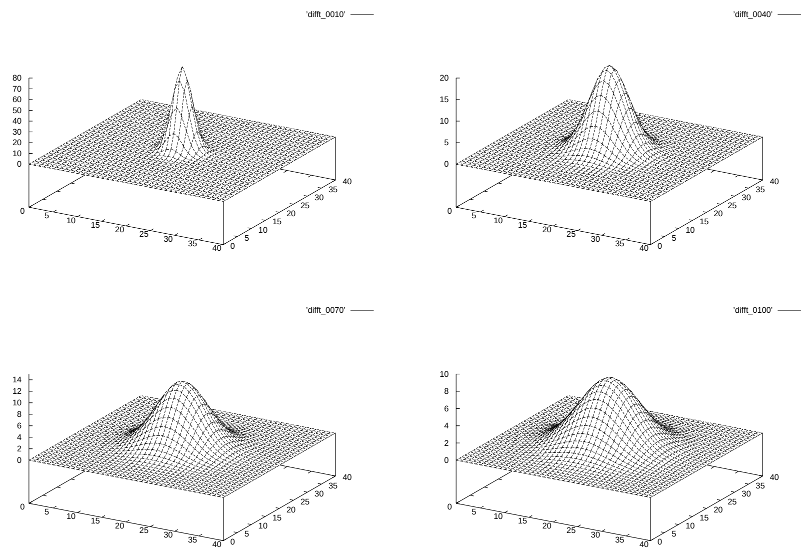


Abbildung 12.2: Zeitabhängige Lösung der Diffusionsgleichung nach 10, 40, 70 und 100 Schritten. Man beachte die verschiedenen vertikalen Achsenskalierungen.

Verwendet man die gleichen Quellen und Senken sowie die gleichen Randbedingungen $\phi_{\mathrm{Rand}} = 0.1$ wie im vorigen Abschnitt, erhält man nach ca. 10000 Zeitschritten genau dieselbe stationäre Lösung wie durch die Gauss-Seidel Iteration. Zusätzlich hat man nun allerdings auch die zeitliche Annäherung an das Gleichgewicht erhalten. Weiterhin sei bemerkt, daß diese Lösung auch durch direkte Monte-Carlo Simulation erhalten werden kann. ϕ ist dann die Wahrscheinlichkeit zur Zeit t ein Teilchen am Ort (x, y) zu finden.


```
  // copy input to output
  while( true ) {
    char c;
    cin >> c;
    if ( cin.fail() ) break;   // end of input or something wrong
    cout << c;
  }
```

Die `continue` Anweisung bewirkt, daß die Bearbeitung des Schleifenkörpers an dieser Stelle abgebrochen und dann je nach Art der Schleife bei der Kontrollanweisung (`for(;;)`), bzw. dem Schleifentest (`(do{}) while()`) wieder aufgenommen wird.

```
  #include <cmath>               // for sqrt() function

  double a[20];
  // ...
  double sum_sqrt=0.;
  for( int i=0; i < 20; i++ ){
    if ( a[i] < 0. ) continue;   // work on next element
    sum_sqrt += sqrt( a[i] );
  }
```

Es sei nochmals betont, daß sich `continue` und `break` nur auf die innerste Schleifenebene beziehen und nicht dazu benutzt werden können, tiefer verschachtelte Schleifen zu verlassen.

2.4.6 goto-Anweisung, Labels

Zu dem Zweck, tiefer verschachtelte Schleifen zu verlassen, kann die `goto` Anweisung benutzt werden.

```
void f(){
  int a[20][30];

  for( int i=0; i<20; i++ )
    for( int j=0; j <30; j++ )
       if ( a[i][j] == 42. )  // found the answer
         goto end_of_loop;    // multi level break
end_of_loop:                  // label, we jump here from inside the loop
```

Der Label (Name) `end_of_loop` kennzeichnet das Sprungziel des `goto`. Der Name des Sprungzieles kann innerhalb des Funktionskörpers frei gewählt werden und wird durch einen Doppelpunkt abgeschlossen. Das Problematische an `goto`-Anweisungen ist, daß man einem Label nicht ansehen kann, *von wo aus* sie mit `goto` angesprungen werden. Außerdem erzwingt häufig die Verwendung eines `goto` die Verwendung mindestens eines weiteren. Im obigen Beispiel müßte vermutlich nach der Schleife eine Behandlung des "nicht-gefunden" Falles anschließen und man müßte dann das Ansprungziel des `goto` mit einem weiteren `goto` umgehen. Besser setzt man oben eine Variable `found` auf einen bestimmten Wert und benutzt diesen dann in weiteren Verzweigungsanweisungen.

Mit `goto` dürfen lokale Blöcke/Klammerebenen verlassen, aber nicht betreten werden. Man darf beispielsweise nicht vom `if`-Teil einer Entschiedung in den `else`-Teil springen, oder von außen in einen Block/Schleifenkörper hinein. Ebenso darf der Kontext einer Funktion nicht verlassen werden.

Programmierweisheit besagt: `goto`'s können immer vermieden werden (im Prinzip durch Ersetzen des `goto` durch die Anweisungen nach dem Sprungziel), sind allerdings in seltenen Einzelfällen bei der Strukturierung hilfreich.

2.4.7 switch-Anweisung

Ein gezähmtes `goto` ist die switch-Anweisung, die je nach Wert einer ganzzahligen Variablen verschiedene Sprungziele einer direkt anschließend mit `case`-Anweisungen gebildeten Liste auswählt und die Verarbeitung dann an dieser Stelle fortsetzt. Dieses Konstrukt ist häufig anzutreffen, wenn eine Variable dazu benutzt wird, eine Auswahl aus einer vorgegebenen Anzahl möglicher Alternativen festzulegen.

Im Prinzip können `switch`-Anweisungen immer durch `if...else if`-Konstrukte ersetzt werden, sind aber häufig übersichtlicher.

```
char c;
cin >> c;
switch( c ) {                 // begin case block
case 'a': cout << "hello world!";
          break;
case 'b': cout << "HELLO ";   // fallthrough intended, comment missing break
case 'c': out << "WORLD!";
          break;
default:  cout << "unknown input";
          break;              // not necessary, but good style
}                             // end case block
```



```
{
  int ISTEP=100;
  double h=1.0/NL;
  double dt=0.0001;
  char cfile[20];

  for(int i=0; i<=NL; i++ ){
    for(int j=0; j<=NL; j++ ){
      world[i][j]=0.0;                // initial values
      source[i][j]=0.0;               // source field
    }
  }
  world[NL/2][NL/2]=NL*NL;

  for( int it=1; it<=ISTEP; it++ ){   // iteration loop
     double wsum=0;
     for(int i=1; i<NL; i++ ){
        for(int j=1; j<NL; j++ ){     // Time step ...
           mirror[i][j]=world[i][j] + dt/(h*h)*
                        (-4*world[i][j]+world[i-1][j]+world[i+1][j]+
                                        world[i][j-1]+world[i][j+1])
                        + dt*source[i][j];
           wsum+=mirror[i][j];
        }
     }
     for(int i=1; i<NL; i++ ){
        for(int j=1; j<NL; j++ ){     // Exchange old and new image
           world[i][j]=mirror[i][j];
        }
     }

     cout << it << ' ' << wsum << ' ';

     sprintf( cfile, "difft_%4.4d", it );
     cout << cfile << '\n';
     std::ofstream outfile1( cfile );
     for(int i=0; i<=NL; i++ ){
       for(int j=0; j<=NL; j++ ){
       outfile1 << world [i][j] << '\n';
       }
       outfile1 << '\n';
     }
     outfile1.close();
  }
```

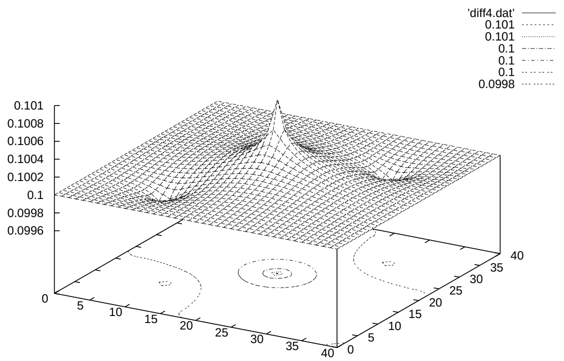


Abbildung 12.1: Lösung einer Diffusionsgleichung mit dem Gauss-Seidel Verfahren. Die Randbedinungen sind im Text angegeben.

die man in der einfachsten Form durch

$$\frac{\phi_{ij}(t+\Delta t)-\phi_{ij}(t)}{\Delta t} = \frac{1}{h^2}(\partial_i^2+\partial_j^2)\phi_{ij}(t)+S_{ij} \tag{12.14}$$

diskretisiert. Aufgelöst nach den Werten von ϕ zur neuen Zeit $t+\Delta t$ hat man

$$\phi_{ij}(t+\Delta t) = \phi_{ij}(t)+\frac{\Delta t}{h^2}(-4\phi_{ij}(t)+\phi_{i-1\,j}(t)+\phi_{i+1\,j}(t)+\phi_{i\,j-1}(t)+\phi_{i\,j+1}(t))+\Delta t S_{ij}\ . \tag{12.15}$$

Ein kleines Programm, mit dem man die Lösung der zweidimensionalen Diffusionsgleichung in Abwesehnheit von Senken ($S=0$) erhält ist hier abgedruckt. Das Programm wurde aus dem Beispiel des letzten Abschnitts entwickelt und enthält nun zwei Felder, da die Berechnung des neuen Feldes diesmal das alte Feld nicht überschreiben darf.

```
#include<iostream>
#include<fstream>
using std::cout;
using std::cin;

static const int NL=40;
static double world[NL+1][NL+1];     // define the lattice
static double mirror[NL+1][NL+1];    // define the lattice image
static double source[NL+1][NL+1];    // define the source

int main()
```


Das obige Programmsegment gibt in Kleinbuchstaben `"hello world!"` aus, wenn der Buchstabe `'a'` übergeben wurde, im Falle von `c` das Wort `WORLD!` und im Falle von `b` die Zeichenfolge `HELLO WORLD!`. Dieses merkwürdig erscheinende Verhalten entsteht dadurch, daß nach Ansprung des passenden `case`-Labels alle folgenden Anweisungen bis zu einem eventuellen `break` durchlaufen werden ("fallthrough"). Da dies häufig nicht das beabsichtigte Verhalten ist, lohnt sich an einer solchen Stelle anstelle des `break` immer ein Kommentar. Das `default`-Label ist optional und wird angesprungen, falls sonst kein passendes in der `case`-Liste gefunden wird.

2.5 Funktionen und Operatoren

Längere Programme enthalten in der Regel Programmteile, die sich mehr oder weniger häufig wiederholen. Diese Teile will man gern an anderer Stelle ausführlich definieren, und im Programm nur durch einen "Platzhalter", also einen *Funktion*saufruf ersetzen. Ein Beispiel ist das Einholen einer Bestätigung durch den Benutzer, ob eine bestimmte numerische Aufgabe das Löschen oder das Umsortieren von Daten ausgeführt werden soll. Weiterhin möchte man häufig aus Gründen der Übersicht bestimmte Programmteile gruppieren und wenn möglich, die dazugehörigen Anweisungen nur noch durch einen Namen kennzeichnen, der ihre "Funktion" charakterisiert.

Das zu diesem Zweck existierende Sprachmittel sind die Funktionen, die in anderen Sprachen auch Prozeduren oder Unterprogramme genannt werden. Eine Funktion ist gekennzeichnet durch Parameter, die eine Art "Eingabe" in den Funktionsblock darstellen und solche, die eine "Ausgabe" darstellen. Im einfachsten Falle erfolgt die Übergabe der Eingabeparameter als Argument an die Funktion und die "Ausgabe" durch einen entsprechenden Rückgabewert.

Außer in wenigen Ausnahmefällen erzwingt C++ die Spezifikation von Argument- und Rückgabetypen von Funktionen. Hier *definiert* `power` eine Funktion, die aus einer Eingabe eines doppeltgenauen Wertes `base` und einer ganzen Zahl `exponent` einen neuen "`double`"-Wert berechnet und zurückgibt. Der Name deutet an, welcher Algorithmus sich dahinter versteckt.

```
// raise 'base' to the power 'exponent', with exponent being integer,
// base being double; accept also negative exponents
double power(double base, int exponent){
   double result  = 1.;    // will multiply this by base 'exponent' times
   bool   neg_exp = false;
   if (exponent < 0 ) {    // for negative exponents we use a single
                           // division at the end of the function
      neg_exp = true;
      exponent = -exponent;
```

```
   }
   for (int i=0; i < exponent; i++ )
      result *= base;      // successive multiplication
   if ( neg_exp )          // have to take the inverse
      result = 1./result;
   return result;          // return the result to the calling program
}
```

Die Namen der Argumente sind wie Variablennamen frei wählbar; unter diesem Namen können sie dann im Funktionskörper angesprochen werden. Der Anweisungsblock des Funktionskörpers wird wie bei Schleifen nicht durch Semikolon abgeschlossen.

Das Schlüsselwort `return` bricht das Unterprogramm an der spezifizierten Stelle ab, legt den Wert des dahinter angegebenen Ausdrucks auf dem Stack ab und kehrt ins Hauptprogramm zurück. Im obigen Fall besteht dieser Ausdruck nur aus der Variable `result`. Runde Klammern werden an dieser Stelle nicht benötigt. Ein Unterprogramm kann mehrere `return` Anweisungen enthalten. Dies kann ein nützliches Mittel sein, um z.B. bei Angabe unpassender Argumenttypen oder bei Problemen mit der Berechnung die Bearbeitung frühzeitig abzubrechen.

Eine Funktion muß definiert oder deklariert sein, bevor sie im Programm verwendet werden kann. Eine Deklaration ist im wesentlichen die erste Zeile der Funktionsdefinition, ohne ihren Körper. Sie teilt dem Compiler den Namen der Funktion und die Typen der Argumente und des Rückgabewertes mit.

```
  double power(double base, int exponent);
```

Anstelle des Anweisungsblocks tritt ein einzelnes Semikolon. Während die Angabe des Argumenttyps zwingend ist, ist die eines Namens für die Argumentvariable dem Benutzer freigestellt. Es ist jedoch guter Stil, hier Namen anzugeben, um die Bedeutung der Argumente klarzustellen.

Deklarationen können im Prinzip beliebig häufig im Programm wiederholt werden, sie sind sozusagen ein Versprechen an den Compiler, daß diese Funktion später oder in einem anderen Programmteil definiert wird. Funktionen sind immer globale Objekte innerhalb eines Namesraumes. Man kann Funktionen nicht wie Variablen innerhalb lokaler Bezugsrahmen definieren, z.B. innerhalb eines geschweiften Klammer-Paares (oder eines begin/end-Blockes wie in PASCAL).

Ein Programm, das von der Tastatur eine Zahl liest, dann `power` benutzt und schließlich das Ergebnis ausgibt, könnte so aussehen:


```
  source[NL/2][NL/2]=2.0;              // source in the center
  source[NL/4][NL/4]=-1.0;             // source in the center
  source[3*NL/4][3*NL/4]=-1.0;         // source in the center

  for( int it=1; it<=ISTEP; it++ ){    // iteration loop
     double wsum=0;
     for(int i=1; i<NL; i++ ){
        for(int j=1; j<NL; j++ ){      // Gauss-Seidel loop
           world[i][j]=(1.0-w)*  world[i][j] +
                         w/4*(  world[i+1][j] + world[i-1][j]
                              + world[i][j+1] + world[i][j-1]
                              + h*h*source[i][j] );
           wsum+=world[i][j];
        }
     }
     cout << it << ' ' << wsum << '\n';

     std::ofstream outfile1("diff4.dat");
     for(int i=0; i<=NL; i++ ){
       for(int j=0; j<=NL; j++ ){
       outfile1 << world [i][j] << '\n';
       }
       outfile1 << '\n';
     }
     outfile1.close();
  }
  return 0;
}
```

Das Ergebnis des Programms ist in Abb. 12.1 dargestellt. Als Randbedinung wurde $\phi = 0.1$ am Rand gewählt und es wurden Quellen ($S(20,20) = 2$) bzw. Senken ($S(10,10) = S(30,30) = -1$) in dem ansonsten quellfreien Gitter der Größe $N = 40$ angebracht.

12.2 Parabolische partielle Differentialgleichungen

Erweitert man die Diffusionsgleichung aus dem letzten Kapitel um eine Zeitableitung, so erhält man eine parabolische DGL

$$\frac{\partial \phi}{\partial t} = \left[\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right] \phi + S(x,y) \; , \tag{12.13}$$

Bei Verwendung der Trapezregel erhält man so die Gleichungen 12.4 wenn man fordert

$$\frac{\partial E}{\partial \phi_{ij}} = 0 \tag{12.10}$$

Aufgelöst nach den ϕ_{ij} ergibt sich

$$\phi_{ij} = \frac{1}{4}\left[\phi_{i+1\,j} + \phi_{i-1\,j} + \phi_{i\,j+1} + \phi_{i\,j-1} + h^2 S_{ij}\right] \ , \tag{12.11}$$

was noch keine Lösung von ϕ ergibt, aber stets zu einer Verbesserung führt wenn man sie auf alle ϕ_{ij} nacheinander (z.B. von links nach rechts und von oben nach unten) anwendet. Man beachte, daß bei der Berechnung des nächsten Werts der vorherige überschrieben wird; das Feld muß also nicht zwischengespeichert werden. Die verallgemeinerte Form

$$\phi_{ij} = (1-w)\phi_{ij} + w\frac{1}{4}\left[\phi_{i+1\,j} + \phi_{i-1\,j} + \phi_{i\,j+1} + \phi_{i\,j-1} + h^2 S_{ij}\right] \ , \tag{12.12}$$

stellt die sog. Gauss-Seidel Iteration mit dem Gewicht w dar. Benutzt man Glg. 12.9 um die Änderung der Energie bei einem Schritt zu berechnen (und beachtet dabei, daß sich nur ϕ_{ij} ändert) erhält man einen Wert der immer positiv ist falls $0 < \omega < 2$. Man nähert sich also mit jedem Schritt dem Minimum der Funktion E. Welche Werte von w optimal sind wird in der Übung getestet.

Eine Realisierung der Gauss-Seidel Iteration findet sich in nachfolgendem Programm.

```
#include<iostream>
#include<fstream>
using std::cout;
using std::cin;

static const int NL=40;
static double world[NL+1][NL+1];    // define the lattice
static double source[NL+1][NL+1];   // define the source

int main()
{
  int ISTEP=200;
  double w=1.5;
  double h=1.0/NL;

  for(int i=0; i<=NL; i++ ){
    for(int j=0; j<=NL; j++ ){
      world[i][j]=0.1;                // initial values
      source[i][j]=0;                 // source field
    }
  }
```



```
#include <iostream>
using std::cout;
using std::cin;

double power(double base, int exponent);  // declaration of 'power'

int main(){                               // definition of main
  double in;
  cin >> in;                              // read value from keyboard

  cout << "-3rd to 3rd power of " << in << ": ";
  for( int i= -3; i <= 3; i++ ){
    cout << power(in, i) <<  " ";         // call power and use return value
  }
  cout <<  "\n";

  return 0;                               // return to operating system
}

double power(double base, int exponent){  // definition of power
  double result = 1.;
  // ... program text as above

  return result;
}
```

Die Definition einer Funktion ist die Festlegung des Programmtextes, der bei Aufruf der Funktion ausgeführt werden soll. Geht die Definition einer Funktion ihrer Benutzung voraus, so kann sie eine Deklaration ersetzen. Die Spezifikation des Programmsegmentes für die Funktion erfolgt innerhalb eines geschweiften Klammerpaares, das *nicht* durch ein Semikolon abgeschlossen wird.

Die Übergabe von Argumenten an eine Funktion erfolgt "by value," d.h., daß in C vor dem Aufruf der Funktion Kopien der Argumente gemacht werden, die üblicherweise auf dem Stack abgelegt werden. Dann erfolgt der Aufruf der Funktion, die dann nur auf den Kopien dieser Variablen auf dem Stack operiert. Es können so niemals die Werte der Variablen im aufrufenden Programm verändert werden. Möchte man dies erzielen, z.B. in Funktionen, die Werte aus einer Datei oder Benutzereingaben einlesen, so benutzt man Zeiger- oder Referenztypen als Argumente, die wir später in Abschnitt 2.6 besprechen werden.

2.5.1 Funktionen ohne Argumente oder Rückgabewert, void

Das Schlüsselwort `void` wird wie ein Typ benutzt, ist aber kein wirklicher Typ. Er dient dazu, Funktionen zu kennzeichnen, die keinen Rückgabewert aufweisen. Dies ist machmal sinnvoll bei Funktionen, deren Wirkung darin besteht nur etwas auf dem Bildschirm auszugeben oder in einer bestimmten Weise ein Argument zu manipulieren.

```
void error(string message){
  cout << "error occured: " << message << "\n";
}
```

Auch Funktionen ohne Argumente können sinnvoll sein. In C++ wird eine Funktion ohne Argumente durch eine leere Argumentliste angedeutet. Ein typisches Beispiel ist ein Pseudozufallszahlengenerator, der bei jedem Aufruf eine andere Zahl zurückliefert. Die C-Bibiothek stellt einen (schlechten) Generator zur Verfügung, der so deklariert ist (in `stdlib.h`):

```
  void srand(unsigned int seed);
  int  rand(void);
```

Dieser wird so benutzt, daß man zunächst eine Sequenz von Zufallszahlen auswählt, durch Aufruf der Funktion `srand(.)` mit einer positiven ganzen Zahl. Danach erhält man bei jedem Aufruf von `rand()` eine neue, ganze Zahl zurück.

2.5.2 Bibliotheken

Wie wir oben gesehen haben, reichen Deklarationen dem Compiler als Information aus, um ein Programm zu übersetzen, und er erzeugt an der entsprechenden Stelle im Programm einen Funktionsaufruf. Allerdings muß dann natürlich später die Information hinzugefügt werden, welche Anweisungen denn wirklich mit diesem Aufruf ausgeführt werden sollen. Mit anderen Worten, wir müssen die übersetzten Funktionsdefinitionen "anbinden" (Computerdeutsch: linken). Dies geschieht durch Angabe des Namens der Programmbibliothek, oder der Objektdatei (`.o`), in der diese enthalten sind.

Ein klassisches Beispiel liefern die Funktionen der Mathematikbibliothek, hier `sqrt()`. Diese sind deklariert in `cmath` und ihre übersetzten Definitionen befinden sich in `libm.a`. Ein Programm, das eine solche Funktion benutzt, sieht vielleicht so aus:

```
// myprog.cc:
```


12.1.2 Randbedingungen

In der Regel werden die Randbedingungen nicht genau auf den Gitterpunkten bekannt sein, d.h. die glatten Ränder eines realen Systems werden durch das Diskretisierungsgitter nur ungenau angenähert. Dieses Problem kann durch eine Gitterverfeinerung in der Nähe der Ränder oder durch Ableitungsdiskretisierung höherer Ordnung umgangen werden [13.1]. Der Einfachheit halber benutzen wir einfach ein quadratisches System, dessen Ränder genau mit den Randpunkten unseres Gitters übereinstimmen. Alle Einträge ϕ_{i0}, ϕ_{iN}, ϕ_{0j} und ϕ_{Nj} sind also vorgegeben. Für einen Punkt, z.B. $(N-1, j)$ mit $1 < j < N-1$, in der Nähe des rechten Rands ergibt sich aus Gleichung 12.4:

$$-\left[-4\phi_{N-1\,j} + \phi_{N-2\,j} + \phi_{N-1\,j+1} + \phi_{N-1\,j-1}\right] = h^2 S_{N-1\,j} + \phi_{N\,j} \ , \tag{12.5}$$

mit allen bekannten Größen auf der rechten Seite. Sind die Randpunkte festgelegt, ergibt sich somit ein lineares Gleichungssystem der Dimension $(N-1)^2$ in den Unbekannten ϕ

$$\mathbf{T} \cdot \mathbf{x} = \mathbf{s} \ , \tag{12.6}$$

welches allerdings für zwei- oder dreidimensionale Systeme sehr schnell unhandlich groß werden kann.

12.1.3 Iterative Lösungsverfahren

Um ein iteratives Lösungsverfahren zu begründen führen wir die Größe

$$E = \int_0^1 dx \int_0^1 dy \ \left[\frac{1}{2}(\nabla\phi)^2 - S\phi\right] \tag{12.7}$$

ein, die in der Elektrostatik den Sinn der Gesamtenergie des Systems hat ($\nabla\phi$ ist der Feldgradient und S ist die Ladungsdichte). Im Falle der Diffusionsgleichung gibt es keine gleichermassen angenehme Analogie, man kann E nur als eine nützliche Größe ansehen. Die Variation von E durch eine kleine Änderung von ϕ,

$$\delta E = \int_0^1 dx \int_0^1 dy \ \left[\nabla\phi \cdot \nabla\delta\phi - S\delta\phi\right] \tag{12.8}$$

kann partiell integriert werden. Da das dabei entstehende Linienintegral am Rand des Volumens verschwindet (solange die Randbedingung erfüllt ist) bleibt im Integral ein Term übrig, der genau Gleichung 12.1 entspricht. Die Forderung, daß E unverändert bleibt ist damit äquivalent zu der Aussage daß ϕ eine Lösung von Gleichung 12.2 ist. Mit anderen Worten ist E minimal wenn man die "richtige Lösung" ϕ einsetzt. Gleichung 12.7 läßt sich einfach diskretisieren, indem man für den Gradienten $\nabla\phi$ die Trapezregel (oder eine bessere Näherung) einsetzt

$$E = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\left[(\phi_{ij} - \phi_{i-1\,j})^2 + (\phi_{ij} - \phi_{i\,j-1})^2\right] - h^2 \sum_{i=1}^{N-1}\sum_{j=1}^{N-1} S_{ij}\phi_{ij} \tag{12.9}$$

eine Eigenwert und Eigenmodenanalyse, da man oft weiß daß sich die Lösungen aus Schwingungen zusammensetzen.

12.1 Elliptische partielle Differentialgleichungen

Wegen der etwas einfacheren Form elliptischer Differentialgleichugen (DGL) und da man elliptische Differentialgleichungen erhält wenn man z.B. die Zeitableitung zu Null setzt, also einen zeitlich unveränderlichen Zustand fordert, werden hier zuerst elliptische DGL behandelt.

Insbesondere sind wir an der Lösung der Gleichung

$$-\left[\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right]\phi = S(x,y) \tag{12.1}$$

interessiert, wozu allerdings noch die Spezifikation von Randbedingungen nötig ist. In einem Gleichgewichts-Wärmeleitungsproblem (Diffusion) entspricht das Feld ϕ der Temperatur und S der lokalen Wärmeerzeugungs- bzw. -verlustrate. Die Randbedingung soll hier in *Dirichlet-Form* angegeben werden, d.h. ϕ wird am Rand eines geschlossenen Gebietes fixiert (und evtl. zusätzlich an geschlossenen Kurven innerhalb des Gebiets). Eine andere mögliche Randbedingung hätte die *Neumann-Form*, d.h. die Ableitung von ϕ wird am Rand vorgegeben.

12.1.1 Diskretisierung

Im folgenden soll das quadratische Volumen der Länge $L = N$ mit einem kartesischen Quadratgitter von $i, j = 0, ..., N$ diskretisiert werden. Mit anderen Worten wird das System in $N \times N$ gleichgroße Quadrate der Seitenlänge h unterteilt, bzw. man hat $N+1 \times N+1$ Gitterpunkte. Nun definieren wir $\phi_{ij} = \phi(x_i, y_j)$ und $S_{ij} = S(x_i, y_j)$, mit $x_i = ih$ und $y_i = jh$, und verwenden die Näherung für die zweite Ableitung aus Kapitel 6.1.2, woraus sich ergibt:

$$-\left[\frac{\phi_{i+1\,j}-2\phi_{ij}+\phi_{i-1\,j}}{h^2}+\frac{\phi_{i\,j+1}-2\phi_{ij}+\phi_{i\,j-1}}{h^2}\right] = S_{ij}\ . \tag{12.2}$$

Verwendet man die Abkürzung

$$\partial_i^2\phi_{ij} = \phi_{i+1\,j} - 2\phi_{ij} + \phi_{i-1\,j} \tag{12.3}$$

und die analoge Abkürzung für $\partial_j^2\phi_{ij}$, so ergibt sich aus Gleichung 12.2 die kürzere Form:

$$-\left[\partial_i^2\phi_{ij} + \partial_i^2\phi_{ij}\right] = h^2 S_{ij}\ . \tag{12.4}$$

Diese Diskretisierung ist relativ einfach und intuitiv, aber manchmal leider zu ungenau. Erfordert das Problem höhere Genauigkeit oder mehr Flexibilität bei der Wahl des Gitters, so kann man andere Diskretisierungen mit einem Variationsverfahren ableiten [13.1].


```
#include <cmath>          // declaration of (C) math functions
#include <iostream>       // declaration of (C++) I/O functions and objects
using std::cin;
using std::cout;

int main(){               // program to compute the sqrt of a number
                          //  entered from the terminal
  double  d;
  cin >> d;
  if ( d < 0. )
    cout << "cannot compute the square root of negative numbers\n";
  else
                          // compute sqrt(d) and use value
    cout << "sqrt(" << d << ") = " << sqrt(d) << "\n";
}
```

Eine UNIX-Kommandozeile, um dieses Programm zu übersetzen und mit den Funktionen der Mathematikbibliothek zu linken, lautet:

```
g++ myprog.cc -o myprog -lm
```

Hier steht `-l` für "library" und `m` für die Datei `libm.a`, die die mathematischen Funktionen enthält; man findet die hinter `-l` anzugebenden Zeichen durch Weglassen von `lib` und der Endung `.a`

2.5.3 Operatoren

Operatoren liefern einen wesentlichen Anteil zur Lesbarkeit eines Programmes. Während wir ohne weiteres einen Ausdruck der Form `3 * z + 5` lesen können, bereitet die Schreibweise `plus(mult(3,z),5)` wesentlich größere Schwierigkeiten. Wir unterscheiden binäre Operatoren mit zwei Argumenten, wie die Multiplikations und Additionsoperatoren, und unäre mit einem, wie den Adressoperator `&` oder das unäre `-`. Seltener sind ternäre Operatoren (siehe Abschnitt 2.4.2).

Arithmetische Operatoren

Die arithmetischen Operatoren sind `+, -, *, /` und `%`. Der Operator `%` (modulo) bildet den Rest der ganzzahligen Division des linken mit dem rechten Operanden (beide ganzzahlig). Wenn beide positiv sind, ist das Ergebnis positiv und echt kleiner als der rechte

Operand. Für einen oder zwei negative Operanden ist das Vorzeichen des Ergebnisses implementierungsabhängig. Die Kombination mit einem Gleichheitszeichen (Zuweisungsoperator) erspart die Wiederholung des linken Operanden:

```
int i=3, j=7;      // initialize
i    = 3 + j % i;  // i = 3 + (j mod i),   i is set to 4
i   += 3 * 4 + j;  // i = i + (3 * 4 + j), i is set to 23
i   /= 5;          // integer division, i is now 4
i   /= 5;          // integer division, i is now 0 !
```

Der Typ eines Operatorausdrucks hängt von den Typen der Argumente ab. Wenn beide Operanden vom Typ `T` sind, so ist auch das Resultat wieder vom Typ `T`. Bei verschiedenen Typen entscheidet der "bessere" Typ (sagen wir `U`), und es wird der `T` Operand erst in `U` gewandelt, bevor die Operation durchgeführt wird. Das Ergebnis des Ausdrucks ist dann vom Typ `U`. C/C++ führt folgende "type promotions" durch:

```
bool -> char -> short -> int -> long
             -> float -> double -> long double
signed -> unsigned  (!)
```

Insbesondere die Verwandlung von vorzeichenbehafteten Größen nach rein positiven führt häufig zu unliebsamen Überraschungen. Der Ausdruck `3U - 5` wird in `3U - 5U` gewandelt und ergibt ein implementierungsabhängiges Resultat, denn das Ergebnis der Subtraktion kann nicht in einer `unsigned int` Variablen gespeichert werden. Diese Regeln der automatischen Typ-Umwandlung gelten sinngemäß auch für die weiteren Operatoren.

Die unären Operatoren `++` und `--` gibt es in postfix und prefix-Form, und sie bewirken, daß der Operand um Eins inkrementiert, bzw. dekrementiert wird. In der postfix-Form ist der Wert der Operation der des Operanden *vor* der In/Dekrementierung, in der prefix-Form der Wert nach der Operation.

```
int i=5, j;
j = i++;        // j is 5 now (i before increment, i is incremented to 6
double a[20];
i=j;                             // i=j=5
std::cout << a[++j] << a[j++];   // prints a[6] twice,    j is now 7
std::cout << a[i++] << a[++i];   // prints a[5] and a[7], i is now 7

// j++ = i;     // ERROR: cannot assign to a temporary
// i = (j++)++; // ERROR: cannot apply ++ to a temporary
```

Kapitel 12

Partielle Differentialgleichungen

Fast jedes physikalische Problem, bei dem Systemgrößen von Ort und Zeit abhängen, läßt sich in Form einer (oder eines Systems von) partiellen Differentialgleichung(en) darstellen. Beispiele dafür sind die schon bekannte Diffusionsgleichung, die elektromagnetischen Wellengleichungen, hydrodynamische Gleichungen, und die Schrödinger Gleichung aus der Quantenmechanik. Bis auf die einfachsten Spezialfälle sind solche Problemstellungnen nicht analytisch zu lösen, man ist also (wieder) auf numerische Hilfsmittel angewiesen. Man diskretisiert die partiellen Differentialgleichungen im Ort und in der Zeit und erhält ein – unter Umständen riesiges – System von Differenzengleichgungen, wie in diesem Kapitel an einigen Beispielen gezeigt wird.

Eine Vielzahl der physikalisch relevanten partiellen Differentialgleichungen enhthalt nur Ableitungen bis zur zweiten Ordnung, und kann deshalb in folgendes grobe Typenschema klassifiziert werden.

- *Parabolische Differentialgleichungen* enthalten eine erste Ableitung einer Variablen und zweite Ableitungen der Anderen. Beispiele sind die Diffusionsgleichung und die zeitabhängige Schrödingergleichgung, die beide eine erste Zeit- und zweite Ortsableitung enthalten.

- *Elliptische Differentialgleichungen* enthalten zweite Ableitungen aller Variablen, wobei diese dasselbe Vorzeichen haben wenn man sie auf eine Seite bringt. Beispiele hierfür sind die Poisson Gleichung für ein elektrostatisches Potential und die zeitunabhängige Schrödingergleichung, beide mit zwei oder mehr Ortsvariablen. Beachtenswert ist, daß der stationäre Zustand einer parabolischen Differentialgleichung eine elliptische Differentialgleichung darstellt.

- *Hyperbolische Differentialgleichungen* enthalten zweite Ableitungen aller Variablen mit verschiedenen Vorzeichen. Ein Beispiel ist die Wellengleichung einer gespannten Saite. Die Lösungsverfahren hyperbolischer Differentialgleichungen beinhalten oft

```
T :=  linalg[matrix]( 3, 3, [ 1, -1, 0,  -1, 2, -1,  0, -1, 1 ] );
det( T );
eigenvects( T );
```

ergeben sich die Eigenwerte $\lambda_3 = 3$, $\lambda_2 = 1$ und $\lambda_1 = 0$ mit den zugehörigen Eigenvektoren $\mathbf{x}_3 = (1, -2, 1)$, $\mathbf{x}_2 = (-1, 0, 1)$ und $\mathbf{x}_1 = (1, 1, 1)$. Die Lösung $\mathbf{x}_3$ entspricht also der Schwingung der mittleren Masse gegen die äußeren, die sich gleichphasig bewegen, bei Lösung $\mathbf{x}_2$ ist die mittlere Masse in Ruhe und die äußeren schwingen gegenphasig. Der Eigenwert $\lambda_1 = 0$ entspricht einer Translationsbewegung des ganzen Systems.

Vergleichs- und logische Operatoren

Für Operationen zwischen zwei integralen Datentypen stehen bitweise, logische und Schiebe-Operatoren zur Verfügung, für die C++ zur Unterstützung beschränkter Zeichensätze auch Schlüsselwörter reserviert. Auch diese Operatoren (siehe Tafel 2.1) lassen sich mit dem Zuweisungsoperator `=` verbinden.

`|`	`bitor`	bitweises oder
`&`	`bitand`	bitweises und
`^`	`xor`	bitweises ausschließendes oder
`<<`		Linksschieben, n-fach
`>>`		Rechtsschieben, n-fach
`~`	`compl`	Komplement, bitweises not, unär
`|=`	`or_eq`	bitweises oder, Zuweisung nach links
`&=`	`and_eq`	bitweises und, Zuweisung
`^=`	`xor_eq`	bitweises ausschließendes oder, Zuweisung
`<<=`		Linksschieben, n-fach mit Zuweisung
`>>=`		Rechtsschieben, n-fach mit Zuweisung

Tabelle 2.1: Bitmanipulationsoperatoren

Die logischen Operatoren sind `&&` (oder auch `and`) und `||` (`or`) und die Negation `!`. Die Operatoren `&&` und `||` haben die besondere Eigenschaft, daß die Auswertung eines Ausdruckes abgebrochen wird, sobald dessen Wahrheitswert feststeht (Sequencing). Diese Eigenschaft teilen sie mit dem Komma-Operator `,` — letzterer wird manchmal benutzt, um komplexe `for`-Schleifen zu konstruieren. Sein Wert ist der Wert des rechtsstehenden Ausdruckes.

```
double    a[20];
unsigned ind[5];

// safe, even if some ind[i] >= 20, since the last expression will not
//  be evaluated in that case
for (int i=0; i < 5 && ind[i] < 20 && a[ind[i]] >= 0 )
   sqrt(a[ind[5]]);

// sequence operator used to combine two expressions
int i, j;
for ( i=0, j=2; i < 18; i++, j++ )
   a[i] = a[j];

 // bizarre but legal use of ,
 i = 5*i, 3;    // i is set to 3, 5*i is computed, but discarded
```

Die (arithmetischen) Vergleichsoperatoren sind `==`, `!=`, `<`, `<=`, `>` und `>=`. Der Wert eines logischen Ausdruck und das Resultat einer Vegleichsoperation in C++ sind vom Typ `bool`, d.h. entweder `true` oder `false`. Dieser Typ ist frei in `int` konvertierbar und liefert dann eine 1 (true) oder 0 (false).

Weitere Operatoren

Der einzige ternäre Operator ist `?:`, der zur Selektion von Subausdrücken (gleichen Typs) dient. Falls der Ausdruck links von `?` wahr ist, wird der erste, sonst der zweite Ausdruck verwendet (siehe Abschnitt 2.4.2).

```
  int i;
  std::cout << (i > 0) ? i : 0;      // never prints a negative number
```

Der Operator `()` bewirkt Funktionsaufruf, `[]` dient als Indizierungsoperator für Zeiger und Felder, ein Punkt `'.'` zur Auswahl eines Elementes aus einer Struktur und `->` dem gleichen Zweck, wenn ein Pointer auf eine Struktur gegeben ist.

```
struct Person {
  string name;
  int age;
};

void f(){
  Person    p;
  Person *p_p = &p;
  p.age    = 25;
  p_p->age = 25;  // equivalent
}
```

Die unären Operatoren `*` und `&` dienen zum Finden des Wertes einer Zeigervariablen, bzw. zum Finden der Adresse einer (beliebigen) Variablen (siehe Kapitel 2.6).

Operatorausdrücke werden nach den in C üblichen Regeln für Rang und ggf. bei Mehrfachvorkommen mit der in Tafel 2.2 zusammengefaßten Assoziativität behandelt:

2.5.4 inline Funktionen

Folgende Operationen finden bei einem Funktionsaufruf und beim Rückkehren aus einer Funktion statt:

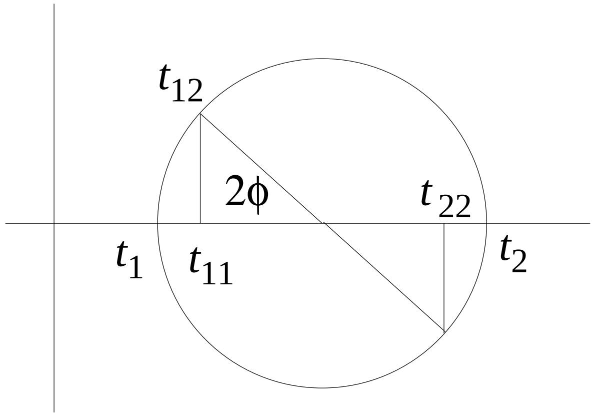


Abbildung 11.1: Mohr'scher Kreis, wobei man nach Antrag der Diagonalelemente t_{11} und t_{22}, sowie des Nicht-Diagonalelements t_{12} des Tensors, direkt vom Kreis die Eigenwerte t_1 und t_2 sowie die Orientierung ϕ ablesen kann.

11.2.3 Die lineare Kette

Eine schöne Aufgabe in der Physik ist die lineare Kette von Massen m, die jeweils mit Federn der Stärke k verbunden sind. Die Bewegungsgleichungen sind

$$\begin{aligned} m\ddot{u}_1 &= -ku_1 + ku_2 & (11.17)\\ m\ddot{u}_2 &= ku_1 - 2\,ku_2 + ku_3 & (11.18)\\ m\ddot{u}_3 &= \phantom{-ku_1 + {}} ku_2 - ku_3 & (11.19)\end{aligned}$$

oder in Matrixschreibweise

$$m\ddot{\mathbf{u}} = \mathbf{K}\mathbf{u} \quad \text{mit } \mathbf{K} = k\begin{pmatrix} -1 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -1 \end{pmatrix} . \tag{11.20}$$

Setzt man den Ansatz $\mathbf{u} = \sum_j \alpha_j \mathbf{x}_j \exp(i\omega_j t)$ ein ergibt sich

$$-m\sum_j \alpha_j \mathbf{x}_j \omega_j^2 \exp(i\omega_j t) = \sum_j \alpha_j \mathbf{K}\mathbf{x}_j \exp(i\omega_j t) \ . \tag{11.21}$$

Jeder Term in der Summe wird (da die $\exp(i\omega_j t)$ linar unabhängig voneinander sind) der Eigenwertgleichung

$$\mathbf{K}\mathbf{x}_j = -m\omega_j^2 \mathbf{x}_j = \lambda_j \mathbf{x}_j \tag{11.22}$$

gehorchen, man muß also wieder nur die Eigenwerte der Matrix finden, um die Lösungen zu kennen. Von den zugehörigen Eigenvektoren kann man dann direkt die Eigenschwingungen ablesen. Mit MAPLE

Aus dem Tensor $\mathbf{T}$ ergibt sich also der Tensor im Eigenraum durch geeignete Drehnung des Koordinatensystems $\mathbf{T}_E = \mathbf{R}^T(\pi/4, \pi/4)\mathbf{T}\mathbf{R}(\pi/4, \pi/4)$, wobei der Superskript T das Transponieren, d.h. den Vorzeichentausch aller Nicht-Diagonalelemente, anzeigt.

Die MAPLE Befehle

```
restart: with(linalg):
matrixRz:=linalg[matrix](3,3,[cos(phi),sin(phi),0,-sin(phi),cos(phi),0,0,0,1]);
matrixRx:=linalg[matrix](3,3,[1,0,0,0,cos(psi),sin(psi),0,-sin(psi),cos(psi)]);
matrixR2:=multiply( matrixRx, matrixRz );

matrixM := linalg[matrix] ( 3, 1, [ 0, 1, 0 ] );
matrixM1 := multiply( matrixR2, matrixM );
evalf( subs( phi=Pi/4, psi=Pi/4, multiply( matrixR2, matrixM ) ) );

T := linalg[matrix] ( 3, 3, [1, 0, 0, 0, 0, 0, 0, 0, 1 ] );
TT:=subs( phi=Pi/4, psi=Pi/4, multiply(matrixR2, T, transpose(matrixR2)));
evalf(");
TTT:=subs( phi=Pi/4, psi=Pi/4, multiply(transpose(matrixR2), TT, matrixR2));
evalf(");
eigenvals( T ); eigenvals( TT ); eigenvals( TTT );
```

erlauben ein einfaches Überprüfen dieser Aussagen und natürlich das Probieren anderer Trägheitstensoren.

11.2.2 Der Spannungstensor

In der Kontinuumsmechanik ist der Spannunstensor von großer Bedeutung. Um ein Gefühl für dieses Objekt zu bekommen betrachten wir der zweidimensionalen Tensor

$$\mathbf{T} = \begin{pmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{pmatrix} , \tag{11.16}$$

wobei wir auch vom symmetrischen Fall $t_{12} = t_{21}$ ausgehen wollen (statisches Gleichgewicht). Für den zweidimensionalen Fall, der einem dreidimensionalen System mit Rotationssymmetrie in einer beliebigen Richtung entspricht, gibt es die besonders anschauliche grafische Darstellung auf dem Mohr'schen Kreis. Die Richtigkeit läßt sich leicht durch elementare Mathematik überprüfen.

<table>
<tr><th>Operator</th><th>Auswertung von</th></tr>
<tr><td><code>::</code></td><td>links nach rechts</td></tr>
<tr><td><code>() [] -> . X++ X-- typeid XXX_cast<TYPE>()</code></td><td>links nach rechts</td></tr>
<tr><td><code>! ~ ++X --X + - * & (TYPE) sizeof new delete</code></td><td>rechts nach links</td></tr>
<tr><td><code>.* ->*</code></td><td>links nach rechts</td></tr>
<tr><td><code>* / %</code></td><td>links nach rechts</td></tr>
<tr><td><code>+ -</code></td><td>links nach rechts</td></tr>
<tr><td><code>>> <<</code></td><td>links nach rechts</td></tr>
<tr><td><code>< <= > >=</code></td><td>links nach rechts</td></tr>
<tr><td><code>== !=</code></td><td>links nach rechts</td></tr>
<tr><td><code>&</code></td><td>links nach rechts</td></tr>
<tr><td><code>^</code></td><td>links nach rechts</td></tr>
<tr><td><code>|</code></td><td>links nach rechts</td></tr>
<tr><td><code>&&</code></td><td>links nach rechts</td></tr>
<tr><td><code>||</code></td><td>links nach rechts</td></tr>
<tr><td><code>||</code></td><td>links nach rechts</td></tr>
<tr><td><code>= *= /= %= += -= <<= >>= &= |= ^=</code></td><td>rechts nach links</td></tr>
<tr><td><code>?:</code></td><td>rechts nach links</td></tr>
<tr><td><code>throw</code></td><td>–</td></tr>
<tr><td><code>,</code></td><td>links nach rechts</td></tr>
</table>

Tabelle 2.2: Bindungsstärke (Priorität) und Assoziativität von Operatoren

1. Sichern des lokalen Programmkontextes (z.B. Variablen, die in CPU-Registern gehalten wurden, Rücksprungadresse) auf den Stack.
2. Kopieren der Funktionsargumente (Pointer oder Klassen) aus dem lokalen Kontext auf den Stack, um Überschreiben aus der Funktion heraus zu verhindern.
3. Anspringen der Einsprungadresse der Funktion.
4. Ausführen der Anweisungen in der Funktion. Berechnung des Rückgabewertes.
5. Ablegen/Kopieren des Rückgabewertes auf den Stack.
6. Rücksprung in das aufrufende Programm, Wiederherstellen des lokalen Kontextes

Die Schritte 1, 5 und 6 stellen dabei Aufwand dar, der vermieden werden kann, indem man den Funktionsaufruf umgeht. Für kurze Funktionen, die in einem Programm sehr häufig aufgerufen werden, können diese Schritte einen signifikanten Anteil der Gesamtausführungszeit der Funktion darstellen. Diesen Zusatzaufwand umgeht man durch sogenanntes *Inlining* der Funktion; gewissermassen wird dabei der Funktionstext an der entsprechenden Stelle im ablaufenden Programm eingefügt. In C hat man an dieser Stelle häufig Makros verwendet, die aber den Nachteil unintuitiver Nebeneffekte haben.

```
#define SQR(a) ((a)*(a))
int   b=2,c;
c   = SQR(b++);  // c = ((a++) * (a++));
                 // c is now 2*3 = 6, b is 4 after 2(!) increments
```

Der Schritt 2 bleibt in der Regel nötig, weil natürlich auch weiterhin keine Variablen unkontrolliert verändert werden dürfen.

Wenn man in C++ im Header-file (siehe Abschnitt 2.10) eine Funktion *definiert* und mit dem Schlüsselwort `inline` kennzeichnet, so versucht der Compiler, den Funktionsaufruf durch Inlining wegzuoptimieren.

```
inline int sqr(int a){ return a*a; }
int  b=2, c;
c   = sqr(b++);  // b++ is evaluated once, hence c is now 4, b is 3
```

Da der Nachteil des Inlining vergrößerte Programme sind, sollte man mit diesem Mittel sparsam umgehen. Lange Funktionen gehören in Implementierungsdateien und nicht in die header-Dateien. Klassen-Elementfunktionen, die im Klassenkörper definiert werden, behandelt der Compiler automatisch als inline. Zusammenfassung:
(i) inlining ist gut für kurze Funktionen, die effizient gemacht werden müssen;
(ii) inline Funktionen müssen im header definiert werden;
(iii) Elementfunktionen, die im header definiert sind, sind per default inline.

2.5.5 Defaultargumente

Häufig stellt sich die Frage, wie man eine Funktion "erweitern" kann, ohne die Funktionalität eines bestehenden Programms, das diese Funktion verwendet, zu verändern, oder wie man eine Funktion, die zur vollständigen Charakterisierung mehrere Argumente benötigt, einfach bedienbar gestalten kann. Diese Aufgaben können mit Defaultargumenten gelöst werden. Beispielsweise könnte eine Funktion zum Löschen eines Feldes so aussehen:

```
void clear(int size, int *array){
  for (int i=0; i < size; i++ )
     array[i] = 0;
}
```

Diese läß sich später leicht zu einer Funktion erweitern, die das Feld mit einem beliebigen Wert füllt:

11.2.1 Der Trägheitstensor

Ein Beispiel für einen Tensor ist der Trägheitstensor eines starren Körpers im Raum:

$$\mathbf{T} = \sum_\alpha m_\alpha[(\mathbf{r}_\alpha \cdot \mathbf{r}_\alpha)\mathbf{I} - \mathbf{r}_\alpha \otimes \mathbf{r}_\alpha] \ , \tag{11.11}$$

hier alternativ in Indexschreibweise angegeben

$$t_{ij} = \sum_\alpha m_\alpha[r_k^2\delta_{ij} - r_i r_j] \ , \tag{11.12}$$

wobei die Summe über alle Punktmassen mit Masse m_α geht und bei gleichen Indizes eine Summation über alle Dimensionen des Raums impliziert wird. Man findet für die Ansammlung von Punktmassen $\mathbf{r}_1 = (1/\sqrt{2}, 1/2, -1/2)$ und $\mathbf{r}_2 = (-1/\sqrt{2}, -1/2, 1/2)$ mit Massen $m_1 = m_2 = 1$ z.B. folgenden Trägheitstensor

$$\mathbf{T} = \begin{pmatrix} 1/2 & -1/2^{3/2} & 1/2^{3/2} \\ -1/2^{3/2} & 3/4 & 1/4 \\ 1/2^{3/2} & 1/4 & 3/4 \end{pmatrix} \ . \tag{11.13}$$

Betrachtet man sich $\mathbf{T}$ in Glg. 11.13 genauer so stellt man verschiedene (unangenehme) Eigenschaften fest. Der Tensor hat einen Eigenwert Null und zwei (entartete) Eigenwerte Eins. Damit weiß man bereits, daß das zugehörige Objekt ein länglicher (eindimensionaler) Körper sein muß – ein Stab oder die Anordnung von Punktmassen auf einer Linie. Der Eigenwert Null entspricht einer Drehung um die Längsachse, die beiden gleichen Eigenwerte entsprechen Drehungen um die beiden Achsen senkrecht dazu.

Durch Berechnung des zu Null gehörenden Eigenvektors erhält man die Orientierung der beiden Massen im Raum, also deren Ortsvektoren. In unserem Beispiel wurde der Tensor $\mathbf{T}$ in Glg. 11.13 durch zweimalige Drehung des Koordinatensystems mittels der beiden Drehmatritzen

$$\mathbf{R}_x(\psi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\psi) & \sin(\psi) \\ 0 & -\sin(\psi) & \cos(\psi) \end{pmatrix} \quad \mathbf{R}_z(\phi) = \begin{pmatrix} \cos(\phi) & \sin(\phi) & 0 \\ -\sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \ . \tag{11.14}$$

aus dem Trägheitstensor der Punktmassen an den Orten $\mathbf{r}_1 = (0, 1, 0)$ und $\mathbf{r}_2 = (0, -1, 0)$ gewonnen. Die kombinierte Drehmatrix erhält man durch Matrixmultiplikation der beiden einfachen Drehungen um die x-Achse und um die z-Achse: $\mathbf{R}(\psi, \phi) = \mathbf{R}_x(\psi)\mathbf{R}_z(\phi)$. Angewendet auf einen Vektor entspricht die Operation der Drehung um ϕ um z und um ψ um x, womit z.B.

$$\begin{pmatrix} 1/\sqrt{2} \\ 1/2 \\ -1/2 \end{pmatrix} = \mathbf{R}(\pi/4, \pi/4) \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \ . \tag{11.15}$$

```
yk:=y;
for i from 1 to 20 do
yk:=multiply( T, y ): y:=yk:
od:
print(yk);
eigenvals(T); eigenvects(T);
```

multipliziert man den Tensor

$$\mathbf{T} = \begin{pmatrix} 2 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 2 \end{pmatrix} , \tag{11.7}$$

20 mal an den zufällig gewählten Vektor

$$\mathbf{y} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \tag{11.8}$$

und erhält so den Vektor

$$\mathbf{y}_3 = \begin{pmatrix} 6973568801 \\ 2 \\ 6973568801 \end{pmatrix} , \tag{11.9}$$

der durch Anschauen als der Basisvektor

$$\mathbf{x}_3 = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \tag{11.10}$$

identifiziert werden kann.

Die Konvergenz kann allerdings sehr langsam sein, wenn das Verhältnis der beiden größten Eigenwerte nahe bei Eins liegt. Hat man den größten Eigenwert gefunden, so entfernt man die entsprechende Komponente im Anfangsvektor $\mathbf{y}$, indem man seine Projektion auf $\mathbf{x}_3$ von $\mathbf{y}$ abzieht. Damit hat man die in Richtung von $\mathbf{x}_3$ zeigenden Komponenten von $\mathbf{y}$ entfernt und wird beim nächsten Durchlauf nur den nächstkleineren Eigenwert mit zugehörigem Eigenvektor finden.

11.2 Beispiele

In der klassischen Mechanik, der Kontinuumsmechanik, der Elektrodynamik und auch in der Quantenphysik tauchen sog. Tensoren auf, deren Eigenschaften es genauer zu verstehen gilt.


```
void clear(int size, int *array, int value=0){
  for (int i=0; i < size; i++ )
     array[i] = value;
}
```

Ein Defaultargument muß einen konstanten Wert besitzen, der zur Compilezeit feststeht und eindeutig ist. Das bedeutet, daß dieser in jeder Übersetzungseinheit nur einmal spezifiziert werden darf. Wenn sichergestellt ist (z.B. mit include guards im header-file), daß der Compiler nur eine Deklaration der Funktion sieht, dann ist der beste Platz, um den Defaultwert festzulegen, in einer Funktionsdeklaration in einem header-file. Ansonsten sollte dieser Wert bei der Definition der Funktion festgelegt werden.

2.5.6 Überladen von Funktionen

Im Gegensatz zu C und einigen anderen Sprachen sind in C++ die Argumenttypen zusätzlich zum Funktionsnamen relevant bei der Auswahl der aufzurufenden Funktion. Es ist möglich, eine `print`-Funktion für `int` zu haben, eine für `double` und eine weitere für `struct Date` {...}. Die Mehrfachverwendung des Funktionsnamens erlaubt, diesen von Typinformation freizuhalten und sich auf die Funktion zu beschränken. Als C-Gegenbeispiel können hier `sprintf` und `fprintf` dienen, in denen `f` und `s` den Typ des ersten Argumentes angeben. Diese Häßlichkeit (die darüberhinaus noch zu Fehlern führen kann, wenn beispielsweise der `FILE *` einmal durch `char *` ersetzt wird) wird in C++ vermieden; der Compiler wählt automatisch die richtige Funktion aus.

Man kann sich vorstellen, daß der Compiler dafür sorgt, daß gewissermaßen programmintern komplexere Namen für Funktionen verwendet werden, die neben dem Funktionsnamem auch noch die Argumente enthalten.

```
void swap(int &a, int &b){   // internal name e.g. swap_int_int
  int tmp = a;
  a = b; b = tmp;
}

void swap(double &a, double &b){     // swap_double_double
  double tmp = a;
  a = b; b = tmp;
}

void f(){
  double a, b;
  int    i, j;
```

```
    // ...
    swap(a, b);  // double version
    swap(i, j);  // int    version
  }
```

Um zu verhindern, daß diese internen Namen auch für Funktionen erzeugt werden, die aus C-Bibliotheken geladen werden, müssen diese für den Compiler sichtbar als `extern "C"` deklariert werden.

```
  extern "C" double sqrt(double);
```

Dieser Mechanismus erlaubt dem Compiler sowohl das Erzeugen der richtigen Namen als auch die Berücksichtigung anderer Eigenarten der Sprache, deren Prozeduren der Linker an das Programm binden soll. Bei einem Funktionsaufruf kann z.B. die Reihenfolge, in der Parameter auf dem Stack abgelegt werden eine andere als die in C++ sein.

Durch Überladen läßt sich ein ähnlicher Effekt erzielen wie mit Defaultargumenten, nämlich, daß nachträglich weitere Versionen der "gleichen" Funktion mit erweiterten Argumentlisten bereitgestellt werden können. Nachteil des Überladens zu diesem Zweck ist, daß dabei der Programmcode dupliziert wird, was zu größeren Programmen führt und größere Disziplin bei der Unterhaltung des verdoppelten Codes erfordert (Fehler (bugs) müssen in beiden Versionen entfernt werden).

Falls `@` für einen eingebauten Operator steht, so erlaubt C++ auch, diesen als `operator@()` anzusprechen. Der Ausdruck `3 * z + 5` könnte also auch, weniger intuitiv, geschrieben werden als `operator+(operator*(3,z),5)`.

In dieser Syntax lassen sich Operatoren genau wie Funktionen überladen. Dies kann dazu dienen, eine natürliche Syntax für benutzerdefinerte Typen wie z.B. Matrizen oder komplexe Zahlen einzuführen.

2.6 Zeiger, Zeiger-Feld-Dualität, und Referenzen

2.6.1 Zeiger

Vom Standpunkt des Computers betrachtet ist jede erreichbare Speicherzelle im (virtuellen) Speicher durch eine eindeutige Zahl gekennzeichnet (Teile des virtuellen Adressbereiches können auf Harddisk gehalten sein, andere können im RAM liegen, wieder andere, gegenwärtig unbenutzte, müssen vom Betriebssystem gar nicht repräsentiert sein). Auf regulären PC's liegen diese Adressen im Wertebereich normaler `int` Variablen, auf anderen

letzten Kapitel vorgestellte Verfahren läßt sich ebenfalls zur Matrixinversion verwenden, wenn man beachtet, daß die rechte Seite nun die Einheitsmatrix ist wobei der unbekannte Vektor $\mathbf{x}$ zur gesuchten Matrix $\mathbf{T}^{-1}$ wird. Die inverse der oben als Beispiel benutzten Matrix ist

$$\mathbf{T} = \begin{pmatrix} -1 & 0 & 1 \\ 1 & 1 & -1 \\ 1 & 0 & -1/2 \end{pmatrix} , \tag{11.5}$$

wie man wieder durch Berechnung, bzw. mit MAPLE findet:

```
I1 := linalg[matrix] ( 3, 3, [ 1, 0, 0,  0, 1, 0,  0, 0, 1 ] );
T_1 := eval( linsolve( T, I1 ) );
test := multiply( T_1, T );
```

Kennt man die inverse einer Matrix, so hat man nebenbei auch das zugehörige Gleichungssystem gelöst, da $\mathbf{x} = \mathbf{T}^{-1} \cdot \mathbf{T} \cdot \mathbf{x} = \mathbf{T}^{-1} \cdot \mathbf{a}$.

11.1.3 Berechnung von Eigenwerten

Der Eigenwert einer Matrix ist der Wert λ, der die Gleichungen $\mathbf{T} - \lambda\mathbf{I} = 0$ löst. Da das Gleichungssystem homogen ist, weiss man, daß es nur dann Lösungen gibt, wenn die Determinante $|\mathbf{T} - \lambda\mathbf{I}| = 0$ ist. Im Fall einer dreidimensionalen Matrix ergibt sich also ein Polynom dritten Grades, dessen Nullstellen die Eigenwerte λ_i $(i = 1, 2, 3)$ der Matrix sind. In zwei und drei Dimensionen ist das Problem also leicht zu lösen. In höheren Dimensionen muß man allerdings oft auf numerische Methoden zurückgreifen.

Ein relativ einfaches Näherungsverfahren besteht darin, die D dimensionale Matrix $\mathbf{T}$ wiederholt (k mal) an einen zufällig gewählten Vektor $\mathbf{y}$ zu multiplizieren:

$$\mathbf{y}_k := \mathbf{T}^k \cdot \mathbf{y} = \sum_{i=1}^{D} \beta_i \lambda_i^k \mathbf{x}_i , \tag{11.6}$$

wobei die $\mathbf{x}_i$ die Eigenvektoren der Eigenwerte λ_i sind, d.h. $\mathbf{T} \cdot \mathbf{x}_i = \lambda_i \mathbf{x}_i$. Man kann den Vektor $\mathbf{y} = \sum_{i=1}^{D} \beta_i \mathbf{x}_i$ in der Basis der Eigenvektoren des Tensors $\mathbf{T}$ darstellen und hat so die (unbekannten) Koeffizienten β_i. Nach vielen Matrixmultiplikationen wird $\mathbf{y}_k$ gegen den Eigenvektor $\mathbf{y}_D$ des größten Eigenwerts λ_D konvergieren, und das Verhältnis $|\mathbf{y}_k|/|\mathbf{y}_{k-1}|$ wird sich λ_D nähern. Mit dem MAPLE Arbeitsblatt:

```
restart: with(linalg):
T:=linalg[matrix]( 3,3,[2,0,1,0,1,0,1,0,2]);
y:=linalg[matrix]( 3,1,[1,2,3]);
```

die Matrix:

$$\mathbf{Q} = \left(\begin{array}{ccc|c} t_{11} & t_{12} & t_{13} & b_1 \\ 0 & q_{22} & q_{23} & b_2 \\ 0 & 0 & q_{33} & b_3 \end{array}\right) \quad , \tag{11.2}$$

wobei die Elemente q_{ij} aus den t_{ij} durch die oben beschriebenen Operationen hervorgegangen sind, z.B. $q_{22} = t_{22} - t_{12}\, m_{21}$. Man beachte, daß alle Zeilenoperationen auch auf den Vektor **a** angewendet wurden, woraus sich das Diagonal-Gleichungssystem $\mathbf{Q} \cdot \mathbf{x} = \mathbf{b}$ ergibt. Um das Gleichungssystem zu lösen muß man nur noch von unten beginnend die Lösungen für alle x_i berechnen, d.h. $x_3 = b_3/q_{33}$, etc.

Als numerisch problematisch erweist sich dieses Verfahren, wenn der Nenner in m_{ij} klein oder Null wird, in welchem Fall man evtl. das Gleichungssystem umstellen muß. Bei großen Matrizen besteht das Hauptproblem in der Rechenzeit, da dieses Verfahren proportional zu N^3 ist, wobei N die Dimension der Matrix ist.

Die Lösung für das System

$$\mathbf{T} = \left(\begin{array}{ccc|c} 1 & 0 & 2 & a_1 \\ 1 & 1 & 0 & a_2 \\ 2 & 0 & 2 & a_3 \end{array}\right) \tag{11.3}$$

ist

$$\mathbf{x} = \begin{pmatrix} a_3 - a_1 \\ a_1 + a_2 - a_3 \\ a_1 - a_3/2 \end{pmatrix} \quad , \tag{11.4}$$

wie man (mühsam) per Hand nachrechnen kann, oder viel einfacher durch die MAPLE Kommandos

```
restart: with(linalg):
T := linalg[matrix] ( 3, 3, [ 1, 0, 2,  1, 1, 0,  2, 0, 2 ] );
a := linalg[matrix] ( 3, 1, [ a1, a2, a3 ] );
x := eval( linsolve( T, a ) );
test := multiply( T, x );
```

überprüfen kann.

11.1.2 Matrixinversion

Kennt man die Matrix $\mathbf{T}^{-1}$, für die gilt $\mathbf{T} \cdot \mathbf{T}^{-1} = \mathbf{I}$, wobei $\mathbf{I}$ die Einheitsmatrix mit Diagonalelementen gleich Eins ist, so hat man die sog. inverse Matrix gefunden. Das im

Architekturen können sie darüber hinaus gehen. C/C++ bietet Möglichkeiten diese “Namen” oder **Adressen** der Speicherstellen für Programmierzwecke nutzbar zu machen. An Übersichtlichkeit gewinnt das Programm dadurch, daß man die Adressen selbst benennen darf. Das Programmsegment

```
int    i=5;
double a=12.34;
char   *c1="hello";   // declaration of an array of char
int    j=0;
```

erzeugt beispielsweise die in Abb. 2.1 abgebildeten Speichereinträge (linke Spalte). Nur die Speichereinträge benötigen wirklich Platz im Computer die Daten enthalten. Die Adressen bzw. Namen belegen nicht mehr Platz wenn man im Programm anstelle von `i` den weniger kryptischen Namen `counter` verwenden würde.

Speichereintrag	Adresse (computerdef.)	Variablenname (benutzerdef.)
5	0xfd10	**i**
12.34	0xfd18	**a**
0xff10	0xfd20	**&c1**
0	0xfd28	**j**
	...	
h	0xff10	**c1[0]**
e	0xff11	**c1[1]**
l	0xff12	**c1[2]**
l	0xff13	**c1[3]**
o	0xff14	**c1[4]**
\0	0xff15	**c1[5]**
	...	

Abbildung 2.1: Speicherbelegung (schematisch) mit realem Speichereintrag (Links), einer willkürlich gewählten, Computer-internen Adresse (Mitte) und dem benutzerdefinierten Variablennamen (Rechts).

Im allgemeinen besteht leider keine Garantie, daß hintereinander definierte Variablen auch direkt nacheinander im Speicher liegen (Ausnahme: Felder). Die Variable `c1` vom Typ `char` enthält einen Verweis auf einen ganz anderen Speicherbereich, in dem die gespeicherten Zeichen abgelegt sind. Die Definition von `*c1` reserviert im Speicher einen Bereich,

der genügt um das Wort `"hello"` abzulegen. Auf die einzelnen Elemente `'h'`, `'e'`, `'l'`, `'l'` und `'o'` kann man allerdings auch mit der Formulierung `c[0]`, `c[1]`, `c[2]`, `c[3]`, und `c[4]` direkt zugreifen. Man beachte dabei daß Numerierungen in C++ immer mit `0` beginnen.

Wenn z.B. nötig wird Daten in eine geordnete Liste schnell einzufügen oder zu löschen, so ist diese Operation nicht durch die Anordnung der Zeichen in einem derartigen linearen Feld zu erreichen. Ein *lineares Feld* ist dadurch ausgezeichnet, daß alle Daten in aufeinanderfolgenden Speicherbereichen liegen (die genaue Lokalisierung wird vom Computer übernommen). Sowohl das Einfügen als auch das Entfernen einer Zahl aus diesem Feld erfordert das Umkopieren aller "oberhalb" des modifizierten Elementes liegenden Daten, im Mittel also etwa halbsoviele Operationen wie das Feld Elemente aufweist. Dieser Aufwand ist nur bei kleinen Feldern akzeptabel. Eine Teillösung dieses Problems liegt in einer *verketteten Liste*, in der jedes Element noch die Information erhält "wo" das jeweils nächste im Speicher liegt (d.h. welches der Name der Speicherzelle ist, die den Wert des Elementes speichert). Die tatsächliche Position des Elementes spielt keine Rolle, so daß immer nur die Speicherzellennamen manipuliert werden müssen. Um zum Beispiel ein Element x zu entfernen, muß nur der Namenseintrag des Vorgängers von x in der Liste so geändert werden, daß er nicht mehr auf x sondern auf dessen Nachfolger zeigt. Kopieren anderer Daten ist nicht mehr nötig, allerdings benötigt eine verkettete Liste evtl. mehr Speicher als eine lineare Liste, da für eine Zahl sowohl ihr Wert als auch eine Adresse abgelegt werden muß.

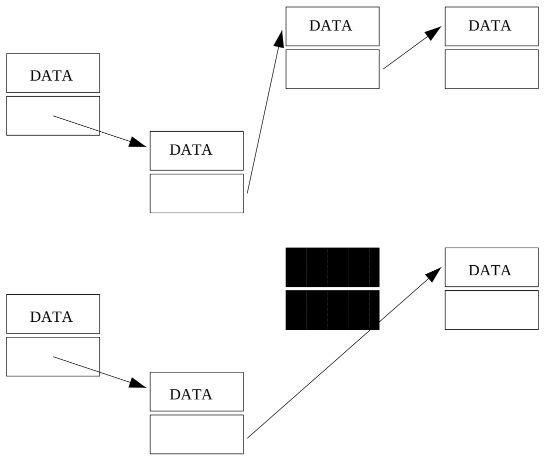


Abbildung 2.2: Benutzung von Zeigern am Beispiel einer verketteten Liste. Oben sieht man die Liste vor Entfernen eines Elementes. Unten ist es nicht in der Liste vorhanden.

Kapitel 11

Lineare Algebra

In diesem Kapitel werden Operationen der linearen Algebra diskutiert, wie sie immer wieder in alltäglichen Problemen der Physik auftreten. Als Beispiele werden zuerst Matrizen (Tensoren) und deren Eigenschaften genauer diskutiert. Danach steht das Lösen linearer Gleichungssysteme, die Invertierung von Matrizen und das Finden von Eigenwerten auf dem Programm. Da viele Probleme in drei Dimensionen, oder bei Rotationssymmetrie sogar in zwei Dimensionen, formuliert sind beschränken wir uns im Folgenden auf dreidimensionale Beispiele.

11.1 Elementare Verfahren

11.1.1 Lineare Gleichungssysteme

Gegeben sei der allgemeine Tensor:

$$\mathbf{T} = \begin{pmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix} . \qquad (11.1)$$

Gesucht ist die Lösung des Gleichungssystems $\mathbf{T} \cdot \mathbf{x} = \mathbf{a}$, wobei $\mathbf{a}$ ein bekannter Vektor und $\mathbf{x}$ der gesuchte Vektor ist. Da man ganze Zeilen mit einer Konstanten multiplizieren, und dann von einer anderen Zeile subtrahieren kann ohne das Ergebniss zu verändern, wird diese Operation im folgenden solange ausgeführt, bis eine Dreiecksmatrix verbleibt, die direkt zur Lösung des Gleichungssystems führt. Um die Elemente t_{i1} $(i > 1)$ zu entfernen subtrahiert man die mit $m_{i1} = t_{i1}/t_{11}$ multiplizierte Zeile 1 von Zeile i. Danach verfährt man in analoger Weise mit den Elementen t_{ij} $(i > j > 1)$ und erhält schließlich

Um den Umgang mit solchen Speicherzellennamen unabhängig vom jeweiligen Rechner zu gestalten, hält C/C++ den Zeiger/Pointer-Typ bereit. Ein Zeiger kann auf einen beliebigen Datentyp verweisen. Dieser Typ kann auch wieder ein Zeiger sein. Zur Deklaration benutzt man einen `*`. Um die Addresse einer Variablen zu erhalten, verwendet man den `&` Operator. Die Deklarationen im Programmsegment

```
// Declaration part
  int     i = 5;        // initialisation
  int  *p_i = &i;       // p_i is a pointer to an int,
                        //   here the variable i
  int **pp_i = (&p_i);  // pointer to pointer to int

// How to use * and & legally in the program
    *p_i = 1;            // set i to 1, p_i remains unchanged
   **pp_i = 2;           // set i to 2, p_i remains unchanged
   int j = 7;            // declare another integer j
   *pp_i = &j;           // change p_i to point on j, *p_i != i
     p_i = &i;           // let p_i point to i again
```

führen zu der in Abb. 2.3 abgebildeten Speicherbelegung.

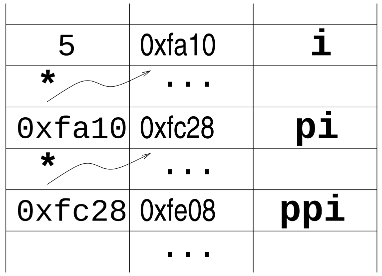


Abbildung 2.3: Speicherbelegung: Bedeutung der Spalten wie in Abb. 2.1.

Um den Wert einer Speicherstelle, auf die ein Zeiger verweist, zu ändern oder in einer Zuweisung oder Rechnung zu benutzen, benutzen wir ebenfalls den `*` Operator.

```
  *p_i = 27;              // i is 27 now
  *p_i = 2 * (*p_i) + 5;  // equivalent to:  i = 2 * i + 5;
```

Konstante Pointer werden durch das Setzen von `const` rechts vom Pointersymbol `*` deklariert. Sie müssen bereits bei der Deklaration initialisert werden. Jede weitere Zuweisung auf den Pointer wird vom Compiler als Fehler erkannt.

```
const int *cp_i = &i;                // ok, i cannot be changed
                                     //     through use of cp_i
                                     //     cp_i can be changed
const char newline = '\n';
const char * const p_nl = &newline;  // const pointer to const char
                                     // note: p_nl = '0'; -> error!
```

Der * Operator hat damit zwei Bedeutungen. Steht er in einer Variablendeklaration, d.h. bei einem Typ, so kennzeichnet er einen Pointer. Vor einer Variablen (vom Typ Pointer) bewirkt er den Zugriff auf den Inhalt der Speicherstelle auf die der Pointer zeigt.

Analog hat der `&` Operator zwei Bedeutungen. Bei einer Variablendeklaration, d.h bei einem Typ kennzeichnet er eine Referenz. Vor einer Variablen ist es der sog. `adress of` Operator, d.h. er liefert die Adresse der Variablen.

Mit Zeigern kann Arithmetik getrieben werden; im obigen Beipiel läßt z.B. `p_i = p_i + 5` den Zeiger um 5 Speicherstellen für `int` weiterwandern. In dieser Operation benötigt der Compiler die Information, wie groß der Datentyp ist, auf den verwiesen wird, um richtig inkrementieren zu können. Ebenfalls kann die Differenz von Pointern gebildet werden, wobei das Ergebnis ein `int`eger ist. Ein definiertes Ergebnis hat diese Operation jedoch nur, wenn die beiden pointer in ein und dasselbe Datenfeld verweisen, s.u.. Wenn der Wert einer Speicherstelle benötigt wird, die man durch Fortschalten des Pointers um `n` Speicherstellen erhält, so gibt es dafür eine kurze Notation mittels `[]`:

```
int p[30];          // declare a field of int
*(p_i + 25)  = 15;  // the value of the 25th entry (past the one
                    // pointed to by p_i) is set to 15
p_i[25]      = 15;  // equivalent ...
```

Die Sprache garantiert, daß 0 niemals als Adresse eines wirklichen Datenelementes vorkommt. Dieser Wert läßt sich also verwenden, um irgendwie geartete Fehler anzuzeigen, die bei einer Operation mit Pointern aufgetreten sind.

Zeiger auf void

Der Typ Zeiger auf void ist eine generische Zeigervariable. Es ist garantiert, das jeder Zeiger auf einen beliebigen Datentyp in einen Zeiger auf `void` umgewandelt werden kann, ohne daß Informationsverlust auftritt. In C wurden daher `void *` häufig dazu eingesetzt, Information an Routinen zu übergeben, die mit beliebigen Pointern arbeiten konnten (z.B. Sortierroutinen wie `quicksort`, Rückgabewert der Speicheralloziierungsroutine `malloc`). In C++ sollten in solchen Fällen `template`-Funktionen zum Einsatz kommen, die die Typsicherheit des Programmes erhalten.

organisierten Abbildung von externen Funktionen (Greifen, Hand, Fuß, Schmecken, ...) auf die Großhirnrinde kommt. Hier ist [1] eine gut lesbare Referenz zur Vertiefung. Hopfield-Netze lassen die Komplexität des Themas “assoziative Speicher” nur erahnen, und natürlich gibt es von allen Modellen viele Spielarten, vielleicht sollten hier asymmetrische Hopfield-Netze ($w_{ij} \neq w_{ji}$) und Netze für die zeitliche Mustererkennung Erwähnung finden. Die folgenden 2 Werke sind für einen Einstieg gut geeignet.

[1] Ritter, Martinetz, Schulten, *Neuronale Netze*, Addison-Wesley, Bonn, 1991.

[2] Christopher M. Bishop, *Neural Networks for Pattern Recognition,* Clarendon Press, Oxford, 1994.

wenden wir jetzt so häufig an, bis keine Änderung der Spin-Muster mehr eintritt, das System also in einen *Fixpunkt* der Dynamik gelaufen ist.

Es sollte klar sein, daß es sich dabei wieder um ein Gradientenabstiegsverfahren handelt und wir uns damit in ein Energieminimum begeben. Wir könnten diese Vorschrift auch als ein Hill Climbing mit deterministischer Zugauswahl, Nullpunkts-Monte-Carlo mit parallelem Spin-update, etc. bezeichnen.

Wie lernt nun so ein Netz? Bisher konnten wir mit überwachtem Lernen arbeiten, d.h. wir kannten die Ausgabe, die das Netz zu bestimmten Eingaben produzieren mußte und konnten die Gewichte entsprechend der Fehler des Netzes mit der Zeit modifizieren. Im Gegensatz dazu präpariert man im Hopfield-Modell die Gewichte von Anfang an so, daß das Netz durch seine Dynamik in diese gelernten Muster hineinläuft. Dazu setzt man

$$w_{ij} = \frac{1}{N}\sum_\nu x_i^{(\nu)} x_j^{(\nu)}, \tag{10.13}$$

wobei N die total Anzahl der Neuronen/Spins im System ist. Diese Kopplungen sind wie oben gefordert symmetrisch.

Weiterhin stellt jedes einzelnes Muster ein Energieminimum des Modelles dar, denn jedes gelernte Muster ist tatsächlich (fast) ein Fixpunkt,

$$\begin{aligned}\sum_j w_{ij} x_j^\mu &= \frac{1}{N}\sum_j\sum_\nu x_i^{(\nu)} x_j^{(\nu)} x_j^\mu \\ &= \frac{1}{N}\sum_j \left(x_i^{(\mu)} (x_j^{(\mu)})^2 + \sum_{\nu\neq\mu} x_i^{(\nu)} x_j^{(\nu)} x_j^{(\mu)} \right) \\ &= x^{(\mu)} + \frac{1}{N}\sum_{j,\nu\neq\mu} x_i^{(\nu)} x_j^{(\nu)} x_j^{(\mu)}. \end{aligned} \tag{10.14}$$

Die letzte Zeile zeigt, daß für große N und wenige gelernte Muster jedes dieser Muster ein Fixpunkt ist. Die verbleibende Summe läuft über $N(p-1)$ Summanden mit zufälligen[1] Werten ± 1. Diese Summe hat den Mittelwert 0, aber die Varianz $\sqrt{N(p-1)}$, so daß die als Rauschamplitude des zweiten Termes etwa $\sqrt{p-1)/N}$ beträgt. Das Modell verliert seine Klassifizierungskraft, wenn diese Rauschamplitude in die Nähe von 1 kommt, also die Anzahl der Muster einen nennenswerten Anteil der Anzahl der Neuronen ist. Genauere Untersuchungen zeigen, daß der kritische Übergang bei $p \approx 0.146N$ stattfindet.

10.6 Weiterführende Literatur

Leider konnten wir das Thema neuronale Netze hier nicht erschöpfend behandeln. Nicht angesprochen wurden die Kohonen-Netze, die uns verstehen helfen, wie es zu einer selbst-

[1] Hier setzen wir voraus, daß die Muster selbst unkorreliert sind

2.6.2 Zeiger-Feld-Dualität

Zwischen Zeigern und Feldern besteht in C/C++ ein enger Zusammenhang (Zeiger-Feld-Dualität), der sich bereits oben in der Verwendung von `[]` zur "indirekten" Dereferenzierung von Pointern andeutete. Ein Feld `a[..]` wird intern durch den Compiler sofort in einen Pointer auf dessen erstes Element verwandelt. Der Feldzugriff wird dann vollständig nach den Regeln der Pointerarithmetik abgewickelt. Dies ist auch der tiefere Grund dafür, daß C/C++-Felder immer bei 0 beginnen (man spart dabei Rechenoperationen bei der Lokalisierung von Adressen).

```
int a[20];
int *p = a;          // ok, a is name of the field and can
                     //     be used as address to the field
int *p = &(a[0]);    // ... equivalent

p[5] = 4;            // access to a through pointer arithmetic,
                     //     a[5] is set to the value of 4
a[5] = 4;            // ... equivalent

int b[20][30];       // b is now an array of 20 arrays of
                     //     30 elements of type int
                     //     int *p = b;  error! wrong type, since
                     //     b is of type pointer to 30 int here
int *r = b[0];       // ok, r points to the first int
                     //     in the first (of 20) array(s) of 30 ints
int (*q)[30] = b;    // ok, q points to the first set of 30 int's,
                     //     q and b can now be used synonymous
q[12][15]   = 26;    // ok, set element (13,16) to the value 26
(*(q+12))[15] = 26;  // ... equivalent
*(*(q+12)+15) = 26;  // ... equivalent
b[12][15]   = 26;    // ... equivalent
```

2.6.3 Feldvariablenübergabe an Funktionen, dynamische Speicherverwaltung

Um ein Feld an eine Funktion (vgl. Abschnitt 2.5) zu übergeben, muß man einen entsprechenden Zeiger übergeben. In einer Dimension ist das einfach ein Zeiger auf das erste Datenelement, in zwei Dimensionen ein Zeiger auf ein Feld der passenden (festen) Anzahl von Elementen. Nur die erste Dimension in der Deklaration kann dabei entfallen, alle anderen müssen zur Compilezeit feststehen, damit der Compiler den für die Zeigerarithmetik nötigen Code erzeugen kann. Dies macht C-Felder sehr unpraktisch im Gebrauch, wenn variable Dimensionen erforderlich sind.

```
void f( int b[][30] )
void g( int (*q)[30] )

int  b[20][30];
int  c[10][30];
int  d[15][31];

f(b); g(b); f(c); g(c); // ok
f(d); g(d);             // errors, since second dimension not ==30
```

Wir wollen hier kurz auf die dynamische Speicherverwaltung vorgreifen und eine weitere Technik zur dynamischen Erzeugung mehrdimensionaler Felder schildern. Der `operator new` *type*[N] gibt einen Zeiger auf das erste Element eines linearen Speicherbereiches von `N` Variablen des Typs *type* zurück. Um ein zweidimensionales Feld von, sagen wir, Dimension N×M zu erzeugen, kann man zunächst einen Speicherbereich von N×M Elementen anfordern. Dann kann man ein Hilfsfeld von Pointern einrichten, die auf die Positionen der Variablen `a[i][0]` verweisen, `i` $= 0..N-1$. Ein weiterer Zeiger auf die erste Position dieses Hilfsfeldes kann syntaktisch wie ein zweidimensionales Feld behandelt werden. Dieser kann dann auch unproblematisch an Unterprogramme weitergereicht werden und erlaubt damit die Übergabe dynamisch angelegter Felder.

```
void f(int **a);         // function expecting 2D array
                         // ...
int  nx, ny;
std::cin >> nx >> ny;    // read dimensions
int **array;             // pointer to first element of
                         //   the 'secondary' array
array = new int *[nx];   // create memory for nx pointers to int
array[0] = new int[nx*ny];
                         // nx*ny elements in the entire array,
                         // array[0] is a pointer to the first
                         // ``row'' of ny elements and equals
                         // the pointer to the beginning of the
                         // entire array
for(int i=1; i < ny; i++ )
  array[i] = array[0] + i * ny;
                         // points to i'th segment of ny integers

if ( nx > 23 && ny > 10 )
  array[23][10] = 17;    // ok if 23 < nx and 10 < ny

f( array );              // OK, array is int **, but subscripting
                         // behaves as if it were a 2D field
```


Man beachte die Ähnlichkeit der Regel (10.10) mit der Perzeptron-Delta-Regel, wenn man die Ableitung der Aktivitätsfunktion für einen Moment in ϵ absorbiert. Wir können daher auch jene Regel als Gradientenabstieg interpretieren.

Diese Lernregel ist zwar sehr allgemein, hat aber den gleichen Schönheitsfehler wie das Hill Climbing, daß wir im letzten Kapitel über Optimierungstechniken besprochen haben. Der Erfolg hängt davon ab, bei welchen Werten der Gewichte wir starten und ob wir ggfs. in einem lokalen Minimum steckenbleiben. Im Prinzip können wir hier das Wissen über Simulated Annealing anwenden, das wir uns dort erworben haben und den Lernalgorithmus entsprechend verbessern.

10.5 Rückgekoppelte Netze

Eine besonders wichtige Rolle für die Mustererkennung spielen Netzwerke, deren Ausgabemuster auf den Eingang zurückgekoppelt werden. Aufgabe des Netzes ist es, für ein verrauschtes Eingabemuster das ursprünglich gelernte unverrauschte Pendant zu ermitteln.

Ein geeignetes Netz wurde 1987 von John Hopfield vorgeschlagen. Es ähnelt in seiner Funktionsweise sehr den im vorigen Kapitel behandelten Spingläsern und es macht aus der Not, daß diese viele Zustände aufweisen, die energetisch niedrig liegen, aber geometrisch sehr unterschiedlich sind, eine Tugend. Wir wollen daher jetzt die Zustände $s_i = y_i = x_i = \pm 1$ zulassen, jeder Spin kann so mit einem Neuron identifiziert werden und ist sogleich Ein- als auch Ausgabewert. Die Gewichte w_{ij} zwischen den Neuronen/Spins entsprechen den (negativen, s.u.) Ising-Kopplungskonstanten J_{ij}. Wir fordern daher, daß die Kopplungen symmetrisch sind $w_{ij} = w_{ji}$, um eine Energiefunktion aufstellen zu können. In dieser sollen Selbstwechselwirkungen (d.h. ggf. von 0 verschiedene w_{ii}) wie im physikalischen Falle nicht gezählt werden,

$$E = -\sum_{i \neq j} w_{ij} s_i s_j. \qquad (10.12)$$

Allerdings sind die w_{ij} langreichweitig, d.h. daß jeder Spin in der Regel mit jedem anderen verbunden ist.

Die Aktivierungsfunktion soll die Vorzeichenfunktion $\text{sign}(x)$ sein, die positiven Argumenten oder Null die +1 und den negativen Argumenten die -1 zuordnet.

Wir definieren nun eine Dynamik des Netzes. An jedem Neuron i ermitteln wir die synaptische Aktivität/das magnetische Feld als Summe $\sum_j w_i j s_j$. Ist dieser Wert negativ, so soll der Spin i den Wert $\text{sign}(\sum_j w_{ij} s_j) = -1$, anderenfalls den Wert +1 annehmen, sich also in Feldrichtung stellen. Denken wir uns die Gewichte als die Kopplungskonstanten $w_{ij} = J_{ij}$, dann entspricht das einer Energieabnahme des Systems. Diese Vorschrift

Ein Maß für den dabei auftretenden Fehler ist die Summe der Fehlerquadrate

$$E = \frac{1}{2} \sum_{\nu}^{p} \sum_{i} \left(y_i^{(\nu)} - y_i(\mathbf{x}^{(\nu)}) \right)^2 . \tag{10.5}$$

Dieser Ausdruck, der von den Gewichten der Zwischen- und Ausgabeschicht abhängt, soll jetzt durch ein Gradientenabstiegsverfahren (Hill Climbing) minimiert werden. Dazu ändert man alle Gewichte w_{jk} und w_{ij} wiederholt um

$$\Delta w_{ab} = -\epsilon \frac{\partial E}{\partial w_{ab}} \tag{10.6}$$

ab. Für hinreichend kleine ϵ bewegt man sich dadurch entlang der steilsten Abstiegsrichtung von E. Die Änderung von E bei Änderung aller Gewichte gemäß dieser Regel ist tatsächlich negativ,

$$\Delta E = -\epsilon \sum_{ab} \left(\frac{\partial E}{\partial w_{ab}} \right)^2 \leq 0. \tag{10.7}$$

Die in (10.6) benötigten Ableitungen erhält man durch die Anwendung der Kettenregel, zunächst für die Verbindungen von der Zwischen- zur Ausgabeschicht

$$\frac{\partial E}{\partial w_{ij}} = -\sum_{\nu} (y_i^{(\nu)} - y_i(\mathbf{x}^{(\nu)}) f'(\sum_{j}^{\prime} w_{ij'} n_{j'}) n_j \tag{10.8}$$

und in ähnlicher Weise für diejenigen von der Eingabe- zur Zwischenschicht,

$$\begin{aligned} \frac{\partial E}{\partial w_{jk}} &= -\sum_{\nu} \sum_{i} (y_i^{(\nu)} - y_i(\mathbf{x}^{(\nu)}) f'(\sum_{j'} w_{ij'} n_{j'}) w_{ij} \frac{\partial n_j}{\partial w_{jk}} \\ &= -\sum_{\nu} \sum_{i} (y_i^{(\nu)} - y_i(\mathbf{x}^{(\nu)}) f'(\sum_{j'} w_{ij'} n_{j'}) w_{ij} \\ &\quad \times f'(\sum_{k'} w_{jk'} x_{k'}) x_k. \end{aligned} \tag{10.9}$$

Diese Ausdrücke lassen sich vereinfachen, wenn man für jeden Schritt nur nur ein einziges Muster und nicht die Summe in (10.8,10.9) verwendet. Das ist erlaubt, wenn wir sicherstellen, daß alle Muster gleich häufig präsentiert werden und wenn ϵ nicht zu groß ist. Verwenden wir außerdem noch für die Ableitungen der Fermi-Funktion $f()$ die Beziehung $f'() = f()(1 - f())$, und bezeichnen die Soll-Ist-Differenz bei Präsentation von $\mathbf{x}^{(\nu)}$ als $e^(\nu)$, dann finden wir

$$\Delta w_{ij} = \epsilon e_i^{(\nu)} n_j y_i (1 - y_i), \tag{10.10}$$

$$\Delta w_{jk} = \epsilon \left(\sum_i e_i^{(\nu)} \right) x_k y_i (1 - y_i) w_{ij} n_j (1 - n_j). \tag{10.11}$$

Diese Technik läßt sich für Felder mit beliebig vielen Indizes anwenden und führt dann zu entsprechend hohem Grad der verwendeten Zeiger.

2.6.4 Referenzen

Eine Referenzvariable etabliert einen zweiten Namen für eine bereits bestehende Variable gleichen Typs. Daher ist klar, daß eine Referenzvariable bei der Deklaration initialisiert werden muß. Danach ändern Zuweisungen nur den den Inhalt der referenzierten Variable, aber nicht die Variable selbst. Viele Compiler implementieren den zugrundeliegenden Mechanismus mit Hilfe von Pointern und wir wollen zwei Programmsegmente, die die gleiche Aufgabe mit Pointern und mit Referenzen erledigen, gegenüberstellen. Es kann jedoch nicht genug betont werden, das Referenzen keine Pointer sind und die jeweilige Implementierung vollständig dem Compilerbauer überlassen ist.

```
int  i;                          |   int i;
int &i_2 = i;                    |   int * const i_2 = &i;
i_2  = 5;     // i is now 5      |   *i_2 = 5;
i = i_2 * i + 5; // i == 30      |   i = (*i_2) * i + 5;
std::cout << i_2;                |   std::cout << *i_2;
```

Man sieht, daß sich eine ähnliche Funktionalität auch mit konstanten Zeigern erreichen läßt, daß aber die Benutzung des "value of" Operators `*` entfällt. Referenzen sind daher einfacher zu verwenden als konstante Zeiger und daher auch weniger fehlerträchtig.

Es ist entscheidend, daß Referenztypen auch von Funktionen (s.d.) zurückgeliefert werden können und damit Funktionen sinnvoll auf der linken Seite von Zuweisungen stehen können.

```
int &access( int a[], int i ){ return a[i]; }

void f(){
   int a[20];
   access(a,10) = 5;  // ok, a[10] is 5
}
```

Der Sinn dieses Beipieles wird klarer, wenn man sich vorstellt, daß in `access()` Bereichsgrenzentests oder ähnliches stattfinden können.

Konstante Referenzen können als Funktionsargumente dienen, wenn die Funktion nur lesend auf die Variable zugreift. In C++ kann die Übergabe als `const &` häufig das

aufwendige Kopieren einer Klassenvariable verhindern und laufzeiteffektiveren Code erzeugen.

Die Übergabe als nichtkonstante Referenz wie die Übergabe von Information per Zeiger verhindert häufig Optimierungen (Stichwort: aliasing von Variablen) und sollte nur dort verwendet werden, wo sie wirklich nötig ist. Auch wird der Compiler nicht ohne Warnung erlauben, temporär erzeugte Variablen, wie die Werte von Ausdrücken oder Zwischenergebnisse einer Rechnung, als nichtkonstante Referenzen an Funktionen zu übergeben.

2.6.5 typedef

`typedef`'s erlauben, neue Namen für bestehende Typen einzuführen, die sich exakt wie der ursprüngliche Typ verhalten.

```
typedef   int    Int32;
typedef   struct { int lower, upper } TwoInt;
```

Sie können dazu dienen, die etwas kryptische Deklarationssyntax von C++ freundlicher zu gestalten. Das folgende zeigt die Definition eines Feldes von Zeigern auf Funktionen.

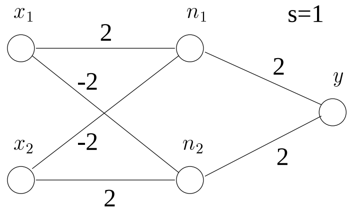
```
double  (*a[20])(double);     // declaration without typedef

typedef   double DblFctDbl(double);
typedef   *DblFctDbl  Ptr_DblFctDbl;
Ptr_DblFctDbl  a[20];         // array of 20 function pointers
```

2.7 Bezugsrahmen von Bezeichnern

`Scope` bedeuted Bezugsrahmen von Variablen, Funktionen usw.

Die Sprache C++ unterscheidet fünf verschiedene Bezugsrahmen
1. Datei
2. Lokal
3. Funktion
4. Klasse
5. Namensraum



x_1	x_2	n_1	n_2	y
0	0	0	0	0
0	1	0	1	1
1	0	1	0	1
1	1	0	0	0

Abbildung 10.5: Realisierung der XOR-Funktion mit Hilfe eines mehrschichtigen Netzes. Angedeutet sind die Gewichte, alle Schwellwerte betragen 1. Die Tabelle listet die Zustände der Zwischenschicht und den Ausgangszustand für die 4 möglichen Ausgangszustände auf. Die ersten zwei und die letzte Spalte geben die korrekten Werte der XOR-Funktion.

und wenden wieder die Delta-Regel an,

$$
\begin{aligned}
w_1'' &= 0.5 + 0.5 \cdot (0-1) \cdot 0 = 0.5 \\
w_2'' &= 1 + 0.5 \cdot (0-1) \cdot 1 = 0.5
\end{aligned}
$$

Mit diesen Gewichten haben wir bereits oben gesehen, daß sich die richtigen Ausgangssignale ergeben.

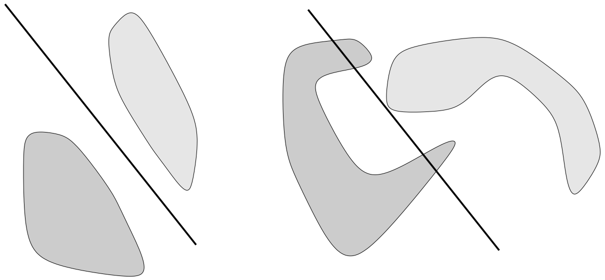
Minsky und Papert haben 1969 gezeigt, daß dieser Lernprozeß immer konvergiert, wenn die Eingangsvektoren linear separierbar sind und als $\epsilon = 1/||\mathbf{x}||$ die inverse Norm des präsentierten Eingangsvektors gewählt wird. Der Beweis findet sich in [1].

10.4 Mehrschichtige Netze

Leider ist bereits die logische XOR-Funktion (wahr genau dann, wenn beide Eingangszustände *verschieden* sind) nicht linear separierbar. In solchen Fällen können mehrschichtige Netzwerke weiterhelfen.

Auch für mehrschichtige Netze läßt sich eine Lernregel angeben. Das Analog zur Delta-Regel bei den Perzeptron-Netzen ist hier die sogenannte *Backpropagation*-Methode. Man nutzt hier als Aktivierungsfunktion $f()$ eine stetig differenzierbare Funktion wie die sigmoide Fermi-Funktion aus dem Abschnitt 10.2.

Wir ordnen der Zwischenschicht die Aktivitäten $n_j = f(\sum_k w_{jk} x_k)$ zu. Analog gelte als Aktivität der Ausgabeschicht $y_i = f(\sum_j w_{ij} n_j)$. Die Gewichte der beiden Schichten wollen wir nur durch die Wahl der Indizes unterscheiden. Wir präsentieren dem Sytem jetzt Eingabemuster $\mathbf{x}^{(\nu)}$ und erwarten die zugehörigen Antworten $\mathbf{y}^{(\nu)}$ für $\nu = 1, \dots, p$.



(a) (b)

Abbildung 10.4: Muster in Äquivalenzklassen, die (a) linear separierbar und (b) nicht linear speparierbar sind.

das Ist-Ausgangssignal fälschlich 0, so wird das Gewicht erhöht, sonst erniedrigt. Die obige Regel läßt sich auch vektoriell schreiben,

$$\Delta \mathbf{W} = \epsilon(\mathbf{y}^{soll} - \mathbf{y}^{ist})\mathbf{x}^T, \tag{10.4}$$

wobei hier die Gewichte die Elemente einer Matrix geworden sind, und auf der rechten Seite der Spaltenvektor mit den Differenzen von Soll- und Ist-Ausgang mit dem Zeilenvektor bestehend aus den Eingangssignalen des präsentierten Musters multipliziert wird. Man macht sich durch kurzes Nachdenken klar, daß dies tatsächlich nur eine andere Schreibweise für die in einem Matrixschema angeordneten Gleichungen 10.3 ist.

Den Ablauf dieses Lernprozesses kann man an folgendem einfachen Beispiel erkennen, in dem ein einzelnes Neuron die Gewichte zur Realisierung der AND-Funktion lernt. Wir verwenden einen Schwellwert $s = 0.75$ und beginnen mit den falschen Gewichten $w_i = 1$. Einsetzen der verschiedenen Kombinationsmöglichkeiten in die Delta-Regel ergibt mit $\epsilon = 0.5$):

$$\begin{array}{llll} \text{für } x_1 = 0,\ x_2 = 0 & \rightarrow & y = 0 \Rightarrow & \text{richtig} \\ \text{für } x_1 = 1,\ x_2 = 0 & \rightarrow & y = 1 \Rightarrow & \text{falsch!} \end{array}$$

Daher wenden wir die Delta-Regel an:

$$\begin{aligned} w_1' &= 1 + 0.5(0-1)\cdot 0 = 0.5 \\ w_2' &= 1 + 0.5(0-1)\cdot 0 = 1. \end{aligned}$$

Mit diesen neuen w_i präsentieren ein neues Eingangssignal

$$x_1 = 0,\ x_2 = 1 \rightarrow y = 1 \Rightarrow \text{falsch!}$$

Dateibezugsrahmen Ein Projekt besteht im Normalfall aus mehreren Dateien. Jedes Modul besteht aus `*.cc` und `*.h`. Bei ungewollten Namenskonflikten besteht die Möglichkeit die entsprechende Funktion `static` zu deklarieren. Ansonsten die Möglichkeit mit `extern` eine “gewollten” Namenskonflikt zu kennzeichnen

Lokaler Bezugsrahmen Hier gilt, daß lokale Namen globale verdecken. Will man explizit auf globale Namen zugreifen, so verwendet man den `scope operator ::`.

```
goto label;
{
  // hier eingefuehrte Variablen sind lokal
  // dieser Block hilft evtl. dem Compiler
}
label:
```

Eine Alternative zu Variablen mit Filebezugsrahmen sind lokale Variablen in Funktionen, die static deklariert sind. Sie haben einen lokalen Bezugsrahmen, aber eine Lebensdauer, die über den Funktionsaufruf hinausgeht.

Klassenbezugsrahmen Bezeichner im Klassenbezugsrahmen werden alternativ durch `.name`, `->name` oder für `static` Variablen `Klassenname::name` angesprochen. Static Variablen existieren unabhängig von der Instanzbildung nur einmal für die ganze Klasse, d.h. alle Objekte/Instanzen einer Klasse teilen sich eine Variable oder Funktion

Namensräume Namensräume sind dazu gedacht, größere Einheiten wie z.B. die gesamte Standardbibliothek zusammenzufassen. Durch Ihre Benutzung wird vermieden, daß es bei Benutzung von mehreren großen Programmpaketen zu Problemen wegen doppelt vergebener Namen kommt. Bei der Benutzung von Teilen dieser Pakete spezifiziert man den Namen mit vorangestelltem `namespace Name{...}` oder importiert diese Teile mit `using` Direktiven.

```
namespace Name{
  ....
}
```

Durch `using` Direktiven kann man einzelne Elemente oder auch einen ganzen Namensraum “importieren”, d.h. auf seine Elemente zugreifen ohne explizit anzugeben, in welchem Namensraum sie sich befinden. Allerdings kann man so nur einen Namensraum importieren.

2.8 Ein- und Ausgabe

Die Ein- und Ausgabe von C++ wurde gegenüber C vollständig überarbeitet. Dies war unter anderem deshalb notwendig, weil in C die Ein- und Ausgabefunktionen eine variable

Anzahl von Argumenten hatten und daher eine Überprüfung der Typen zur Compilezeit nicht stattfinden konnte.

Die beiden wichtigsten Objekte, mit denen man ständig zu tun hat, sind die Ein-/Ausgabe Kanal-Typen `istream` und `ostream`. Beide sind im header `iostream` deklariert. In den vorangegangenen Kapiteln sind die Standard Ein-/Ausgabe Kanäle '`cin`' und '`cout`' mit den Ein-/Ausgabe Operatoren '`<<`' oder '`>>`' bereits wiederholt benutzt worden. Im folgenden sollen die Anwendungs- und Erweiterungsmöglichkeiten genauer erklärt werden.

Wie funktioniert die Verkettung der Ein- oder Ausgabe?

```
#include <iostream>
//      Was der user eingibt:
     cout << "Gehalt: "   << Personal.mGehalt;

//      und wie es der Compiler interpretiert :
   ( cout << "Gehalt: " ) << Personal.mGehalt;
```

Zur Ausgabe des Strings `Gehalt:` wird der Operator `<<` vom `ostream cout` aufgerufen, dieser gibt wieder einen `ostream` zurück, mit dem wieder der Operator `<<` aufgerufen wird, diesmal mit `Personal.mGehalt` als zweitem Argument. Aus diesem Grund ist es unter Umständen auch erforderlich, die Argumente zu klammern. Im Zweifel ist es immer besser eine Klammer zuviel als keine Klammer zu setzen. Analog wird nach einer Eingabe wieder ein `cin` zurückgegeben, so daß man eine weitere Eingabe anfügen kann usw..

2.8.1 Elementfunktionen der iostreams

Anstatt die obige, bisher gebrauchte Formulierung der Ein- und Ausgabeoperationen zu verwenden, kann man für besondere Aufgaben auf Funktionen zurückgreifen, die in Tabelle 2.3 gezeigt sind.

```
   char c1='x';
   cout << c1 << '\n';    // Bisher verwendete Schreibweise
   cout.put( c1 );        // Kann ersetzt werden durch Funktionsaufrufe
   cout.put( '\n' );
```

Einige der Elementfunktionen erlauben Zugriffe, die in der bisherigen Formulierung schwieriger zu realisieren sind.

Diese N Eingangssignale können z.B. die Graustufen eines digitalisierten Bildes oder auch z.B. die Werte 0 oder 1 eines logischen Schaltkreises sein. Als Aufgabe stellen wir dem Perzeptron, für eine Menge von möglichen Eingangsmustern eine Zuordnung in Äquivalenzklassen vorzunehmen. Seien etwa die $x^{(\nu)}, \nu = 1 \dots p$ die möglichen Eingangsmuster (etwa verschiedene Fotografien von verschiedenen Studenten in einem mit N Personen besetzten Hörsaal, jedes gerastert mit einer Auflösung von L Pixel) und sei für jedes ν bekannt, in welche von N Äquivalenzklassen (repräsentiert durch die eindeutige Identität des dargestellten Studenten) das Muster fällt, dann soll bei Präsentation eines Musters $x^{(\nu)}$ aus einer bestimmten Äquivalenzklasse ein bestimmtes der N Neuronen feuern und damit die Äquivalenzklasse identifizieren.

Das Neuron r soll feuern, wenn

$$\sigma_r = \sum_i^L w_{ri} x_i^{(\nu)} > 0 \tag{10.2}$$

ist. Hier haben wir angenommen, daß ein möglicher Schwellwert s der Aktivierungsfunktion verschwindet. Diesen können wir ohne Probleme in das obige Modell einbauen, wenn wir jedem Muster ein weiteres Signal $x_0^{(\nu)} = 1$ zuordnen, das mit jedem der N Neuronen über das Gewicht $w_{r0} = -s$ verbunden ist.

Es stellen sich nun zwei grundsätzliche Fragen: (i) Kann ein so spezifiziertes System überhaupt die Klassifizierungsaufgabe lösen, und (ii) falls ja, wie kann man die Gewichte w_{ri} ermitteln?

Die Antwort auf die erste Frage ist "manchmal." Um das einzusehen, nehmen wir an, daß wir Gewichte w_{ri} auf irgeneine Weise ermittelt hätten. Dann wird durch die Beziehung $\sum_i^L w_{ri} x_i = 0$ eine Hyperebene in dem L-dimensionalen Raum gegeben, der die Mustervektoren beherbergt. Damit jetzt das Neuron r alle Muster der Äquivalenzklasse $A(r)$ korrekt klassifiziert, muß diese Hyperebene so liegen, daß ausschließlich Muster aus $A(r)$ "oberhalb" liegen. In Abb. 10.4(a) wird diese Situation in $L = 2$ Dimensionen für 2 Äquivalenzklassen gezeigt. Der Teil (b) der Abbildung zeigt ein Beispiel, in dem keine lineare Separierbarkeit erreicht werden kann.

Im Falle, daß eine Klassifikation möglich ist, kann man ein Perzeptron mit der folgenden Lernregel dazu bringen, die Gewichtsfunktionen w_i korrekt zu erlernen. Falls eine Klassifizierung korrekt war, bleiben die Gewichte unverändert, sollte sie inkorrekt gewesen sein, so ändert man die Gewichte gemäß der sogenannten *Delta-Regel* an jedem Neuron r, das eine falsche Antwort gegeben hat,

$$\Delta w_{ri} = \epsilon (y_r^{soll} - y_r^{ist}) x_i. \tag{10.3}$$

Die positive Zahl ϵ, die üblicherweise $\ll 1$ gewählt wird, damit bereits gelernte Gewichte nicht zu sehr verändert werden, heißt *Lernrate*. Mit dieser Regel werden die Gewichte erhöht/erniedrigt, die aktive Eingänge mit falsch berechneten Ausgängen verbinden. War

wie z.B.

$$f(x) = \frac{1}{1+\exp(-x)}. \tag{10.1}$$

Beide Aktivierungsfunktionen sind in Abb. 10.3 gegenübergestellt.

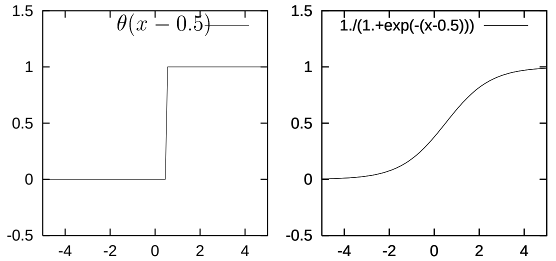


Abbildung 10.3: Mögliche Aktivierungsfunktionen sind die Heaviside-Funktion (links) und die Fermi-Funktion (rechts), hier dargestellt für Argumente mit einem positiven Schwellenwert von 1/2.

Mit Hilfe eines einzelnen Modellneurons lassen sich logische Grundoperationen wie die OR-Funktion oder die AND-Funktion realisieren. Ihre Wahrheitstabellen sind

OR	0	1
0	0	1
1	1	1

AND	0	1
0	0	0
1	0	1

Wir nehmen jetzt zwei Eingänge an und wählen als Aktivierungsfunktion die Sprungfunktion mit Schwellwert. Dann kann die OR-Funktion durch Wahl der Gewichtsfaktoren mit $w_{1,2} = 1$ und Schwellwert $s = 0.25$ dargestellt werden, für die AND-Funktion ist $w_{1,2} = 0.5$ und $s = 0.75$ eine geeignete Kombination.

Diese Neuronen, wie wir sie im folgenden weiterhin nennen werden, können auf verschiedene Weise miteinander verschaltet werden und ergeben dann Neuronale Netzwerke mit breit gestreuten, verschiedenen Eigenschaften und Anwendungen.

10.3 Das Perzeptron

Die einfachste Möglichkeit, ein solches Netz aufzubauen, ist das sogenannte *Perzeptron*, das eigentlich nur aus N unabhängigen Neuronen besteht, die jeweils die gleichen L Eingangssignale empfangen.

`ostream::put (char c);`	gibt einen character 'c' aus.
`char c='x';` `cout << c;`	Zum Vergleich ... schreibt den character 'c'
`istream::get (char c);`	liest einen character 'c' vom Eingabe-Kanal.
`char c;` `cin >> c;`	Zum Vergleich ... liest einen character 'c' ein, ignoriert aber SPACE, TAB, EOL.
`istream::get (char*pc, int anzahl, char terminator);`	liest maximal 'anzahl' character von `istream` oder endet, wenn der 'terminator'-character eingelesen wurde. Voreinstellungen: `anzahl=1`; `terminator='\n'`. Der 'terminator' wird im Kanal belassen.
`istream::getline (char*pc, int anzahl, char terminator);`	wie `istream::get`, aber der 'terminator'-character wird gelöscht.
`istream::putback (char c);`	legt den vorher mit `istream::get` aus dem Eingabe-Kanal gelesenen character `c` wieder zurück.
`istream::peek (char c);`	liest den character 'c' von `cin`, beläßt ihn aber darin.

Tabelle 2.3: Prototypen einiger Elementfunktionen der Ein-/Ausgabe Kanal-Typen `ostream` und `istream`. Die Kanal-Typen werden bei der Anwendung im Programm z.B. durch `cout` und `cin` ersetzt.

2.8.2 Formatierung

Oft ist es nötig das Format der Ausgabe zu ändern, um z.B. eine schönere, besser lesbare Tabelle zu erhalten, oder um die volle Genauigkeit einer Variablen auf dem Bildschirm auszugeben. Solche Operationen geschehen mit Hilfe der in Tabelle 2.4 zusammengefaßten Funktionen.

Dabei ist zu beachten, daß die Änderung, die durch solche Funktionen herbeigeführt wird, evtl. nur für die nächste Ausgabe gültig ist. Um diese Funktionen verwenden zu können, muß man zuerst `#include<iomanip>` aufrufen.

```
#include<iomanip>
   double x=1234.5;
   cout << x << '\n';   // Ergibt: 1234.5
   cout << setfill ('*') << setw (10) << setprecision (5);
   cout << x << '\n';   // Ergibt: ****1234.5
   cout << x << '\n';   // Ergibt wieder: 1234.5
```

Manipulator	Beschreibung
`int width=12;` `cout << setw(width);`	setzt die minimale Größe der direkt folgenden Ausgabe.
`int prec=5;` `cout << setprecision(prec);`	Gibt die Zahl der Stellen für Festkomma-, oder die Zahl der Stellen hinter dem Komma für Gleitkommadarstellung an; sonst die Maximalzahl der Stellen.
`char c='*';` `cout << setfill(c);`	Definiert ein Füllzeichen wie z.B. `*`.

Tabelle 2.4: Beispiele von Formatierungsfunktionen für die Ausgabe.

Weiterhin gibt es sog. Flags zum Schalten von Formatfunktionen. Dies sind Operationen, die keinen Parameter brauchen. Beispiele sind in Tabelle 2.5 gezeigt.

Flag	Gruppe	Funktion
left	adjustfield	links ausrichten
right	adjustfield	rechts ausrichten
internal	adjustfield	+/- links, Zahl rechts
dec	basefield	Dezimalzahlen
hex	basefield	Hexadezimalzahlen
oct	basefield	Oktalzahlen
showbase		zeigt Dezimalbasis von hex- und oct-Zahlen
showpos		'+' vor Zahl, wenn positiv
uppercase		E, X, A-F statt e, x, a-f
fixed	floatfield	Festkommazahl
scientific	floatfield	wissenschaftliche Darstellung

Tabelle 2.5: Schalter für formatierte Ausgabe.

Will man ein Flag setzen, d.h. einen Ausgabemodus aktivieren, so verwendet man alternativ:

```
cout.setf (ios:: 'flag', ios:: 'group')
cout << setiosflags (ios:: 'flag', ios:: 'group')
```

wobei die Möglichkeiten für `'flag'` und `'group'` Tabelle 2.5 zu entnehmen sind. Der zweite Übergabeparameter ist nicht unbedingt notwendig. Will man die Funktionalität rückgängig machen, bzw. die Voreinstellung wiederherstellen, so kann man dies (wieder alternativ) für jedes einzelne Flag, oder auch für eine ganze Gruppe tun.

Netz durch die Eigenschaften der synaptischen Kontakte.

Ein solcher Lernmechanismus, dessen experimentelle Details allerdings weiterhin umstritten sind, wurde bereits 1949 durch den Psychologen *Hebb* in seinem Werk *Organization of Behaviour* als Hypothese verwendet. Diese *Hebbsche Hypothese* besagt, daß sich das synaptische Gewicht proportional zur korrelierten Aktivität vor und hinter der Synapse ändert. Das bedeutet, daß Synpasen, die aktiv waren und damit ein Feuern des Neurons bewirkt haben, ihre synaptischen Gewichte verstärken/erhöhen.

10.2 Modellneuronen

Während für die detaillierte Untersuchung von Einzelneuronen ein genaues Modell der Reizleitung und des Zeitverhaltens der Neuronen nötig ist, reichen für den Aufbau von Modellnetzen und der Untersuchung ihrer prinzipiellen Eigenschaften vereinfachte Neuronenmodelle aus. Eine Möglichkeit ist in Abb. 10.2 gezeigt: Über die Eingänge $x_1, ..., x_L$

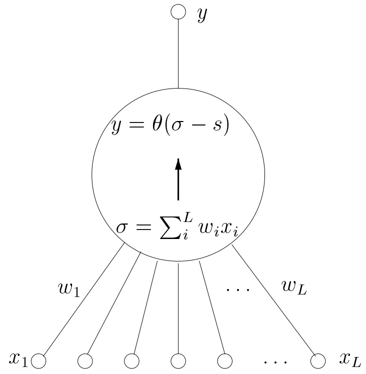


Abbildung 10.2: Ein Modellneuron mit Eingängen x_i, Gewichten w_i und einem Ausgang y. Das Ausgangssignal ergibt sich nach Gewichtung der Eingänge und Subtraktion eines Schwellwertes s durch Anwendung einer nichtlinearen Aktivierungsfunktion, hier der θ Funktion.

empfängt das Modellneuron Signale von außen. Diese werden an das Soma weitergeleitet und dabei mit einem Gewichtsfaktor $w_1, ..., w_L$ versehen. Im Soma wird nun die Summe über die Eingangssignale $\sigma = \sum_{i=1}^L w_i x_i$ gebildet. Das Ausgangssignal y der Zelle wird über eine nichtlineare Aktivierungsfunktion ermittelt, die z.B. die Form einer Sprungfunktion $y = \theta(\sigma - s)$ haben kann. Das resultierende Signal ist dann entweder "1" (Neuron feuert), wenn σ einen bestimmten Grenzwert s überschritten hat, oder "0" (Neuron feuert nicht), falls s nicht erreicht worden ist. Im Falle, daß kontinuierliche Ausgangssignale erwünscht sind, ersetzt man die Sprungfunktion durch eine stetig verlaufende Funktion,

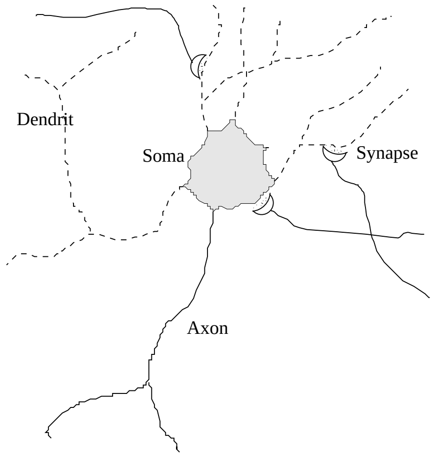


Abbildung 10.1: Struktur einer Pyramidenzelle. Dem Zellkörper (Soma) leiten Dendriten (gestrichelt) Reize zu, während Axonen (durchgezogen) den Erregungszustand vom Soma weg an andere Nerven weiterleiten. Der Kontakt zu anderen Nerven geschieht über Synapsen, in denen Erregungspotentiale die Auschüttung chemischer Botenstoffe bewirken. Diese lösen ihrerseits auf der dendritischen Seite elektrochemische Vorgänge aus, die das elektrochemische Potential lokal ändern. Diese lokale Störung wandert im Dendriten zum Soma, auf dem ebenfalls direkt Synapsen enden können. Die Nervenzelle feuert, wenn die "Summe" der einlaufenden Signale einen bestimmten Schwellwert überschreitet.

sich ebenfalls verzeigen kann und Verbindungen zu bis etwa 1000 anderen Nervenzellen herstellt.

Die Verbindung zwischen Axon einer Zelle und Dendrit oder Soma einer anderen Zelle wird durch Synapsen hergestellt. Je nach Typ der Synapse und Erregungszustand des Axons werden chemische Botenstoffe ausgeschüttet, die exzitatorische oder inhibitorische Wirkung auf die Zelle ausüben können. Diese äußert sich in einer lokalen Änderung der elektrochemischen Eigenschaften der Zellwand, die letztlich in einer lokalen Spannungsänderung resultiert, die im Dendriten propagieren kann. Bei mehreren aktiven Synapsen addiert (exhibitorische Synapse) oder subrahiert (inhibitorische Synapse) der Dendrit im wesentlichen die hereinkommenden Signale und leitet diese Information an das Soma weiter. Überschreitet dann das dortige Signal einen gewissen Schwellwert, so *feuert* das Neuron und gibt über das Axon ein Signal aus.

Die chemischen Eigenschaften der Synapsen können sich mit der Zeit abhängig von den Zuständen von Dendrit und Axon auf mehreren Zeitskalen ändern (*short term/long term potentiation*). Vereinfacht gesprochen, können Synapsen die einlaufenden Signale gewichtet an den Dendriten weitergeben. Dies ermöglicht die Speicherung von Informationen im


```
   cout.unsetf (ios:: 'flag')
   cout << resetiosflags (ios:: 'flag')

   cout.unsetf (ios:: 'group')
   cout << resetiosflags (ios:: 'group')
```

2.8.3 Dateien (Files)

Zum Lesen (von) und Schreiben (auf) Dateien gibt es die beiden Kanal-Typen `ifstream` und `ofstream`. Da diese beiden Kanal-Typen von den Standardstreams abgeleitet sind existieren für sie alle Elementfunktionen der Standardstreams.

```
# include <fstream>

   double  z=1.234;
   int     m=5;
   ofstream outs ( "filename", ios::out );
                        // open the file with name
                        // 'filename' for writing
                        // ios::out can be skipped
   outs << x << '\n';   // the use of the self-defined output-stream
   outs << j << '\n';   // 'outs' is analogous to the use of 'cout'
   outs.close();        // close the out-fstream 'outs'

   ifstream ins  ( "filename", ios::in );
                        // open the file with name
                        // 'filename' to read from
                        // ios::in can be skipped
   ins >> z;            // the use of the self-defined input-stream
   ins >> m;            // 'ins' is analogous to the use of 'cin'
   ins.close();         // close the in-fstream 'ins'
```

Spätestens bei der Bearbeitung von Dateien benötigt man Informationen darüber, ob man am Ende des Files (EOF) angelangt ist, oder ob das Öffnen erfolgreich war. Dafür gibt es die sogenannten Statusinformationen, die je nach Zustand der Datei `true` oder `false` zurückgeben:

```
   ins.good()   // true, if no error occured in stream 'ins'
   ins.eof()    // true, if the end of file/stream is reached
   ins.bad()    // true, if an hardware error occured
   ins.fail()   // true, if a logical error or a
```

```
            //         hardware error occured

ins.clear()  // set ins.good() to true
ins.clear( ios::failbit | ins.rdstate() )
             // set fail() manually to true
```

Wie liest man nun aus einer Datei bis zu deren Ende? Die folgenden Beispiele sollen die Schwierigkeiten aufzeigen, auf die man bei der Implementierung dieser einfachen Aufgabe stoßen kann.

```
int n;

while (!cin.eof()){
   cin >> n;
   // .... this reads too far
}

while((cin >> n).good){
   // ... this reads not far enough
}

while(cin >> n){
  // ... this is the way it works
}
```

2.9 Dynamische Speicherverwaltung

Zum Anfordern von Speicher dienen in C++ die Operatoren `new` und `new []` für einzelne Datenelemente oder Felder. Das Freigeben des Speichers erfolgt mit `delete` und `delete []`.

```
   int    *p_i  = new int;
   Person *p_p  = new Person("B.Stroustrup",45);
   int        n = 40;
   int    *pa_i = new int[n];
// ...
   delete   p_i;
   delete   p_p;
   delete[] pa_i;
```

Kapitel 10

Neuronale Netze

10.1 Einführung

Neuronale Netze sind der Versuch, ein System mit einer Funktionsweise aufzubauen, die in Teilaspekten der des menschlichen Gehirns ähnelt und auch für ähnliche Aufgaben geeignet ist. Ganz allgemein gesprochen sind sie Systeme aus miteinander verbundenen, zur Informationsverarbeitung fähigen Elementen, die Neuronen genannt werden. Diese Bezeichnung ist der Biologie entliehen, da im Gehirn die Neuronen (Nervenzellen) die informationsverarbeitenden Elemente sind. In einem künstlichen neuronalen Netz sind diese Neuronen als physikalische oder mathematische Modelle mit mehreren Ein- und Ausgängen realisiert, deren mathematisches Verhalten im Prinzip den biologischen Neuronen entspricht. Die wichtigsten Unterschiede zu den "normalen" Computern erkennt man, wenn man einen Vergleich zwischen den typischen Funktionsweisen von Computern und Gehirn zieht:

	Computer	Gehirn
Arbeitsprinzip	sequentiell	parallel
Berechnungen	digital/exakt	mit Fehlertoleranz/näherungsweise
Datenspeicherung	lokal	über das Netzwerk verteilt

Besonders interessant dabei ist die Möglichkeit, vom Prinzip her parallele Rechner mit hoher Rechenleistung bauen zu können.

Wir zeigen in Abb. 10.1 den Aufbau einer sogenannten Pyramidenzelle, deren Funktionsweise für Neuronen typisch ist, die aber nur einen von vielen Zelltypen im Gehirn darstellt. Die reizempfindlichen Bestandteile der Zelle sind der Zellkörper oder *Soma* und die *Dendriten*, lange, verzweigte Fortsätze des Zellkörpers. Diese stellen sozusagen die Eingangsleitungen des Neurons dar. Ein Ausgangssignal leitet die Zelle durch das *Axon* ab, welches

Die Anforderung geschieht durch Angabe des Typs oder des Klassennamens nach `new`. In diesem Fall wird ein Datenelement der Klasse oder des Typs im Speicher mit Hilfe des zugehörigen Defaultkonstruktors angelegt und der Operator gibt einen Zeiger auf dieses Element zurück. Um einen anderen als den Defaultkonstruktor zu verwenden, benutzt man nach dem `new` einfach den entsprechenden Konstruktoraufruf, wie oben für die Klasse Person gezeigt.

Für Felder wird `new[]` verwendet. Hier wird Speicherplatz für die Anzahl von Speicherelementen angefordert, die innerhalb der eckigen Klammern angegeben ist. Für jedes Element wird der Defaultkonstruktor aufgerufen. Der Operator `new[]` gibt einen Zeiger auf das erste Datenelement zurück. Zum Zugriff auf die Elemente jenseits des ersten verwendet man pointer-Arithmetik oder subscripting mit `[]`.

```
Person  *p = new Person[100];
p[0] = Person("Matthias",30);
p[1] = Person("Stefan",32);
```

Der Operator `new` wirft die exception `bad_alloc`, falls die Speicheranforderung fehlschlägt. Sollte diese unbehandelt bleiben, schreibt das Programm ein core file und bricht ab.

2.10 Organisation in Implementierungs- und Header-Dateien

Ein großes Programm in einer einzigen Datei zu verwalten ist schwierig! Dies führt zu langen Übersetzungszeiten, unübersichtlicher Anordnung der verschiedenen Teile und zu extremen Problemen, die Datei konsistent zu halten, wenn mehrere Personen an diesem Projekt arbeiten.

Man trennt Programme daher meist in mehrere Dateien auf, die getrennt voneinander compiliert werden. Übersetzbare (Implementierungs-) C++-Dateien erhalten den Suffix `.cc` (aber auch andere Endungen sind üblich). Dateien, die nur Deklarationen enthalten, heissen auch header-Dateien und dienen dazu, Funktionen und Daten aus einem Programmteil im anderen bekanntzumachen. Diese Dateien haben den Suffix `.h`. In der Regel (Ausnahme z.B. `main.cc`) gehört zu jeder Implementierungsdatei auch eine Header-Datei.

Header-Dateien sollten enthalten:

1. Sog. include-guards, die das mehrfache Einschließen von Header-Dateien verhindern (vom Sprachstandard verlangt)

```
#ifndef HEADER_H
#define HEADER_H
  // declarations
#endif
```

2. Deklarationen von Funktionen, die modulübergreifend verwendet werden sollen. Zu Deklaration von Funktionen, die nur jeweiligen Programmteil Verwendung finden sollen, können weitere Header-Dateien verwendet werden;
3. Definition von `inline` Funktionen
4. Deklaration von Klassen und deren Elementfunktionen
5. `extern` Deklarationen globaler Variablen, Deklaration statischer Klassenvariablen.
6. Deklaration und Definition von `template` - Funktionen und Klassen.

Implementierungsdateien sollten enthalten:

1. Implementierung von Elementfunktionen von Nicht-Template-Klassen,
2. Implementierung von regulären Funktionen.
3. Deklaration von file-scope, class-scope und globalen statischen Variablen.

2.11 Weiterführende Literatur

Die Vielzahl von unbekannten Begriffen in den beiden letzten Abschnitten weist darauf hin, das die Programmiersprache C++ noch viele Möglichkeiten für den interessierten Programmierer bereithält. Im begrenzten Rahmen dieser Einführung ist es leider nicht möglich auf höhere Sprachelemente wie Klassen, Templates, Überladung, Scopes etc. einzugehen. Wenigstens soll hier jedoch auf weiterführende Literatur hingewiesen werden.

1. Brian W. Kernighan and Dennis M. Ritchie, *The C-Programming Language*, 2nd edition, Prentice Hall, 1988.
2. B. Stroustrup, *The C++ Programming Language*, 3rd edition, Addison Wesley, 1997.
3. C. S. Horstmann, *Mastering C++*, John Wiley & Sons, New York, 1996.
4. S. B. Lippman, *C++ Einführung und Leitfaden*, 3. Auflage, Addison-Wesley, Bonn, 1998.

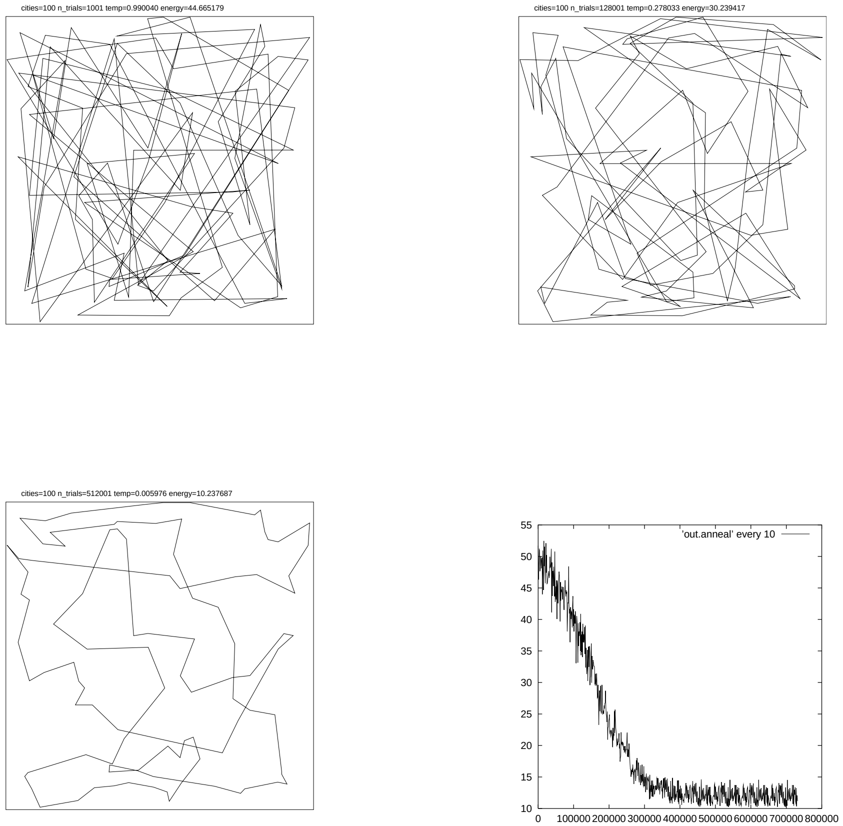


Abbildung 9.7: Entwicklung der Reiselänge bei Simulated Annealing. Ein Zeitschritt entspricht einem Monte-Carlo Schritt, ist also deutlich weniger aufwendig als ein Zeitschritt in der Vergleichssimulation für den genetischen Algorithmus. Dies erklärt die größeren Zahlen an der Zeitachse der Darstellung der Länge der Route gegen die Zeit (rechts unten).

Als Abkühlungsstrategien kann man z.B. die folgenden benutzen:

1. exponentielles Abkühlen $T' = (1-\epsilon)T$. Hier bietet sich das Abbrechen des Algorithmus nach Erreichen einer minimalen Temperatur an.
2. lineares Abkühlen $T' = (1-\epsilon n)T$, wobei n die Anzahl der bisherigen Durchläufe des Algorithmus ist. Beim linearen Abkühlen terminiert der Algorithmus nach $n = \frac{1}{\epsilon}$ Versuchen.

Zum Abschluß zeigen wir auch für diesen Fall die Zeitentwicklung der Weglänge des Handelsreisendenproblems mit den gleichen Städten wie im Abschnitt über genetische Programmierung. Auch hier sieht man noch, wie der Algorithmus in einem Nebenminimum gefangen bleibt und die Güte der Lösung etwa vergleichbar mit der des genetischen Algorithmus ist.

Kapitel 3

Lösung der Newtonschen Bewegungsgleichung

3.1 Der Harmonische Oszillator

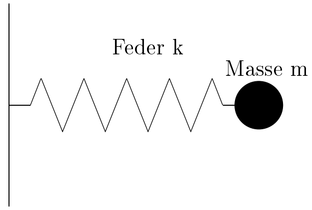


Abbildung 3.1: Schwingungsfeder

Wir gehen davon aus, daß die Masse punktförmig (die Federmasse ist vernachlässigt), die Gravitation gleich Null ist und die Bewegung reibungslos. Dann gilt das Newtonsche Bewegungsgesetz für den harmonischen Oszillator:

$$m\ddot{x} = -kx \tag{3.1}$$

Dabei ist k die Federkonstante. Es gelten die Anfangsbedingungen $x(0) = x_0$ und $\dot{x}(0) = v_0$. Die Lösung dieser Gleichung ist

$$x(t) = A\sin(\omega t + \delta) \qquad \text{mit} \quad \omega^2 = \frac{k}{m} \tag{3.2}$$

Die Anfangsbedingungen ergeben sich durch das Einsetzen von $t = 0$ zu $x_0 = A \sin\delta$ und $v_0 = A\omega\cos\delta$. Daraus werden dann die Konstanten A und δ berechnet:

$$\tan\delta = \frac{x_0\omega}{v_0} \quad \text{und} \quad A = \frac{x_0}{\sin\delta} \tag{3.3}$$

Um eine Funktion in einem Programm verwenden zu können, muß sie diskretisiert werden, d.h. eine kontinuierliche Kurve wird in Teilstücke zerlegt. Dafür benutzen wir ein Feld `x[i]`, wobei `i` nur ganzzahlige Werte annehmen kann. Unsere Zeit t berechnet sich dann durch $t =$`i`dt (dt ist das Zeitintervall zwischen zwei Stützstellen). Damit entspricht der Übergang $t \to t + dt$ dem Übergang `i` $\to$ `i+1`.

Das folgende Programm gibt die exakte Lösungsgleichung (2) wieder:

```
#include<iostream>
using std::cin;
using std::cout;
#include<fstream>
using std::ostream;
#include<cmath>

int main()
{
// define x(t)
   double x[1000], t;  // output-variables

// parameter definition
   double A, delta;     // A=amplitude, delta=phase-angle
   double mass, kfeder; // mass=mass, kfeder=spring-const.
   double t_max, dt;    // t_max=total time, dt=time-interval

// read parameters from std::cin
   cout << "Amplitude      ";  cin >> A;
   cout << "Phasenwinkel   ";  cin >> delta;
   cout << "Masse          ";  cin >> mass;
   cout << "Federkonstante ";  cin >> kfeder;
   cout << "Gesamtzeit     ";  cin >> t_max;
   cout << "Zeitintervall  ";  cin >> dt;
   double omega=sqrt(kfeder/mass);

// loop from t=0 to t=t_max
   t=0.0;
   for( int i=0; i<=t_max/dt; i++ ) {
        x[i]=A*sin(omega*t+delta);  // calculate the funktion
        t=t+dt;                     // increase time
```


Was passiert hier für absinkende Temperaturen $T \to 0$? Für die Besuchswahrscheinlichkeit eines beliebigen Zustandes s_i gilt

$$p(s_i) = \frac{e^{-\beta E(s_i)}}{\sum e^{-\beta E(s_i)}} \tag{9.7}$$

$$= \frac{e^{-\beta E(s_i)}}{e^{-\beta E(s_i)} + \sum_{i\neq j} e^{-\beta E(s_j)}} \tag{9.8}$$

$$= \frac{1}{1 + \sum_{i\neq j} e^{-\beta(E(s_j)-E(s_i))}}. \tag{9.9}$$

Falls also nur ein einziges globales Minimum s_0 existiert mit $E_0 < E_j \; \forall j \neq 0$, dann finden wir

$$\Rightarrow \lim_{\substack{\beta\to\infty \\ (T\to 0)}} p(s_0) = \frac{1}{1+0} = 1, \tag{9.10}$$

d.h., der Zustand s_0 wird mit Wahrscheinlichkeit 1 angenommen. Bei Entartung gilt in analoger Weise für die energetisch gleichliegenden Zustände s_i,

$$p(s_i) = \frac{1}{1 + \sum\limits_{i\neq j} 1} = \frac{1}{\text{Anzahl der Zustände}}, \tag{9.11}$$

also Gleichverteilung über die entarteten Grundzustände.

Wir sehen daher, daß wir mit einer geeigneten Abkühlstrategie auf $T = 0$, bzw. $\beta \to \infty$ die Minima finden können. Leider ist nicht nur der Grenzübergang $T \to 0$ zu machen, sondern in den obigen Gleichungen ist noch der Grenzübergang versteckt, der uns überhaupt die Definition der Wahrscheinlichkeit als die relative Häufigkeit des Besuchs von s_i erlaubt. Die obigen Argumentation bedingt also, daß dem System bei jeder Temperatur ausreichend Zeit zur Verfügung steht, verschiedene Zustände zu besuchen. Die Abkühlstrategie spielt also eine wesentliche Rolle, und sie bestimmt wie gut das System es vermeiden kann, in Nebenminima steckenzubleiben.

Wie sieht der zugehörige Algorithmus aus?

1. Wähle einen beliebigen Ausgangszustand: $s \leftarrow s_1$, und $T > 0$
2. Suche ein $s' \in G(s)$, so daß alle s_i prinzipiell nach endlicher Zugzahl zugänglich sind
3. Setze $p = e^{-\beta(E(s')-E(s))}$ Falls $p > 1 \quad (E(s') < E(s)) \quad s \leftarrow s'$
   Falls $p < 1$ wähle E(s') mit Wahrscheinlichkeit p.
4. Wähle eine neue (geringfügig kleinere) Temperatur $T' < T$ oder verbleibe für eine gewisse Zeit bei der alten.

Der Name *hill climbing* kommt daher, daß sich der Zustand bei diesem Verfahren immer zu niedrigeren Energien hin bewegt (also immer den "Hügel hinabsteigt"). Das Problem dieses Verfahrens sind allerdings die lokalen Minima, da es einmal in einem solchen gefangen, nicht mehr "herausklettern" kann. Somit ist also nicht gewährleistet, daß die bestmögliche Konstellation gefunden wird.

9.4.2 Stochastic Hill Climbing

Um auch globale Minima aufzufinden, müssen auch scheinbar schlechte Schritte gegen die Abstiegsrichtung mit einer bestimmten Wahrscheinlichkeit zugelassen werden. Dies geschieht im *Probabilistic Hill Climbing.* Dabei soll die Wahrscheinlichkeit p für einen solchen Schritt abnehmen, je größer die Verschlechterung ist. Ein Beispiel dafür ist:

$$p = \begin{cases} e^{-\beta(E(s')-E(s))} & \text{falls } E(s') > E(s), \\ 1 & \text{falls } E(s') < E(s). \end{cases} \tag{9.5}$$

Allerdings treten auch hier Probleme auf, denn

- ein einmal erreichtes Minimum wird auch wieder verlassen, und
- die Umgebungen $G(s)$ müssen hinreichend groß gewählt werden, um ein "Steckenbleiben" zu verhindern.

Beim *Simulated Annealing* verfolgt man daher die Strategie, langsam vom *Probabilistic Hill Climbing* zum normalen *Hill Climbing* überzugehen.

9.4.3 Metropolis Monte Carlo und Simulated Annealing

Es läßt sich zeigen (Metropolis, hier ohne Beweis), daß die oben für das *Stochastic Hill Climbing* eingeführten Zugregeln (9.5) im Grenzwert langer Simulationszeiten zu einer Besuchswahrscheinlichkeit der einzelnen Zustände s führen, die der Boltzmannstatistik in der Thermodynamik und Statistischen Physik entspricht. Der Beweis benötigt als Voraussetzung, daß unsere Manipulationsvorschrift $s \to s'$ es erlaubt, alle Systemzustände zu erreichen und daß der Zugvorschlag von s nach s' mit gleicher Wahrscheinlichkeit wie der Vorschlag s' nach s gemacht wird.

$$p(s_i) = \frac{1}{Z} e^{-\beta E(s_i)} \tag{9.6}$$

wobei $\beta = \frac{1}{kT}$ die inverse Temperatur und Z der Normierungsfaktor $Z = \sum e^{-\beta E(s_i)}$ ist, der die Summe der Besuchswahrscheinlichkeiten aller Zustände zu 1 ergibt.


```
   }

// output 2 columns to file harmon.dat
   ofstream outfile("harmon.dat");
   t=0.0;
   for( int i=0; i<=t_max/dt; i++ ) {
      outfile << t << " " << x[i] << "\n";
      t=t+dt;
   }
   outfile.close();                // close file
}
```

Die Funktion plottet man dann am besten mit dem Programm `gnuplot` und dem Befehl `plot 'harmon.dat'`. Auf dem Bildschirm sollte dann die in Abb. 3.2 dargestellte Zeichnung erscheinen.

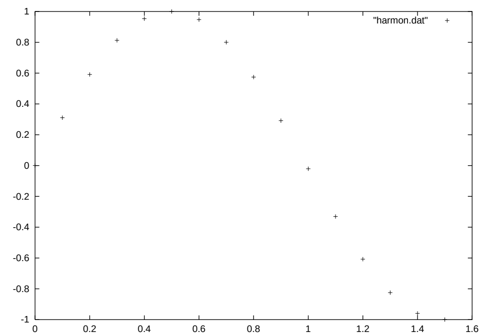


Abbildung 3.2: Ausdruck von gnuplot für $A = 1$, $\delta = 0$, $m = 1$, $k = 10$, $t_{\max} = 1.5$, und $\Delta t = 0.1$, direkt aus der analytischen Lösung berechnet.

3.1.1 Das Euler-Verfahren

Das Euler-Verfahren ist die einfachste Möglichkeit eine Differentialgleichung numerisch zu lösen. Zunächst muß man die Gleichung umformen. Das Newtonsche Bewegungsgesetz lautet $\ddot{x} = -\omega^2 x$. Mit der Geschwindigkeit $v = \dot{x}$ ensteht die Gleichung $\dot{v} = -\omega^2 x$. Die Ableitungen von x und v können für $\Delta t \to 0$ durch den Differenzenquotienten dargestellt

werden:

$$\dot{x} = \frac{x(t+\Delta t) - x(t)}{\Delta t} \quad \Rightarrow \quad x(t+\Delta t) = x(t) + v\Delta t \tag{3.4}$$

$$\dot{v} = \frac{v(t+\Delta t) - v(t)}{\Delta t} \quad \Rightarrow \quad v(t+\Delta t) = v(t) - \omega^2 x \Delta t \tag{3.5}$$

Mit Hilfe dieser Gleichungen kann nun eine numerische Approximation im Computer durchgeführt werden. Zuerst werden die Formeln wieder diskretisiert und anschließend wird der Einfachheit halber der Index `i` um Eins verringert. Dies entspricht der Berechnung der aktuellen Werte `i` aus den alten Werten `i-1`. Die Formeln haben einen Fehler in der Größenordnung $O(\Delta t^2)$. Dieser Fehler wirkt sich im Verlauf des Verfahrens durch ansteigende Energie aus. Die Gesamtenergie ist wegen Energieerhaltung aber konstant (Energie = kinetische Energie + Federenergie): $E = \frac{1}{2}mv^2 + \frac{1}{2}kx^2$. Zur Kontrolle der Energieerhaltung wird die Gesamtenergie E ebenfalls im Programm diskret berechnet.

```
// ...
   const int nx=1000;                              // fieldlength
   double x[nx], v[nx], e[nx], t;
// ...
// make sure that the number of intervals is smaller than nx
   if( t_max/dt >= nx) {
      cout << "ERROR: dt is too small, set to: " << t_max/nx << '\n';
      return -1;
   }
// setup initial conditions
   x[0]=A*sin(delta);
   v[0]=A*omega*cos(delta);
// loop from t=0 to t=t_max
   for( int i=1; i<=t_max/dt; i++ ) {
        v[i]=v[i-1]-omega*omega*x[i-1]*dt;         // next velocity
        x[i]=x[i-1]+v[i-1]*dt;                     // next position
        e[i]=0.5*(mass*v[i]*v[i]+kfeder*x[i]*x[i]);
   }
// output to file
   ofstream outfile("eulerh.dat");
   t=0.0;
   for( int i=0; i<=t_max/dt; i++ ) {
      outfile << t << " " << x[i] << ' ' << v[i] << ' ' << e[i] << "\n";
      t=t+dt;
   }
   outfile.close();
```


Wie bereits oben angedeutet wurde, werden bei sinkender Temperatur die statistischen Gewichte der in der Zustandssumme des Systems auftauchenden Grundzustände immer bedeutender. Das System geht bei Abkühlung auf $T = 0$ schließlich in einen der Grundzustände über. Anschaulich statten wir dazu ein “Teilchen,” das den Systemzustand (Zustandsvektor) repräsentieren soll, mit einer bestimmten kinetischen “Wärme”energie (charakterisiert durch die Temperatur T) aus. Das Teilchen/System ist in der Lage, je nach der Größe von T in Nachbarzustände zu wechseln, die energetisch nicht zu weit entfernt liegen. Nachbarzustände sind dabei solche, die wir durch geeignete Variationen aus dem Ausgangzustandsvektor erzeugen können.

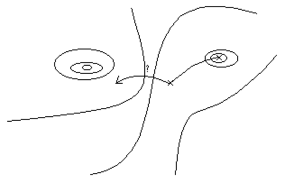


Abbildung 9.6: Energie*landschaft*: wie kann man das globale Minimum finden?

Durch Senken der Temperatur verliert das Teilchen Bewegungsenergie und gelangt tiefer und tiefer in die vorhandenen Potentialmulden. Es hat jedoch immer eine kleine Wahrscheinlichkeit, aus einer solchen Mulde wieder zu entweichen. So ist es möglich, die Minima der Energielandschaft abzusuchen und im tiefsten zu verbleiben. Wir wollen auch diesen Vorgang genauer verstehen und beginnen bei einer stark vereinfachten Vorversion des *Simulated Annealing.*

9.4.1 Hill Climbing

Diese Vorstufe ist das sogenannte *hill climbing.* Dazu

- nehme man einen Ausgangszustand s aus der Gesamtheit $Z(s)$ von Zustandsvektoren, etwa ein N-tupel des Handlungsreisendenproblems oder einen beliebigen Spinzustand des Isingmodelles.
- dann versuche man durch Anwendungen eines Satzes von Manipulationsvorschriften (etwa Vertauschen zweier Positionen bzw. Umklappen eines Spins) einen neuen Zustand $s' \in G(s)$ mit energetisch günstigerer Konfiguration zu erreichen. Dabei ist $G(s)$ die durch diese Manipulationsvorschrift definierte Nachbarschaft von s.
- Der Algorithmus endet, wenn keine günstigere Permutation gefunden werden kann (d.h. wenn $E(s') > E(s)$, $\forall s' \in G(s)$).

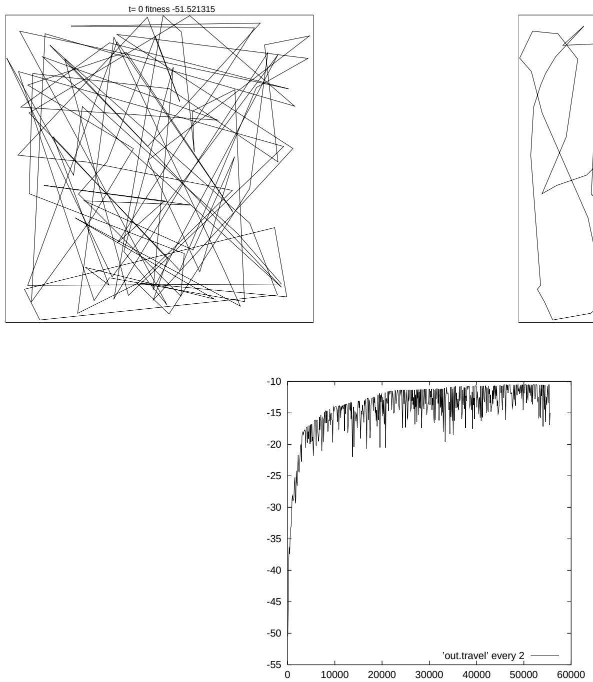


Abbildung 9.5: Entwicklung des “besten Individuums” einer Population von 50 und seiner Fitness als Funktion der Zeit. Ein Zeitschritt wurde mit der Sequenz `MCSMCSMCSSS` durchgeführt, wobei `M` für eine Mutation, `C` für eine Rekombination (crossover) und `S` für ein Selektionsereignis steht. Die Fitness wird maximiert, so daß sie zur negativen Länge der Rundreise gewählt wurde.

9.4 Simulated Annealing

Der englische Begriff *annealing* bedeutet aushärten, wie es etwa ein Schmied mit einem glühenden Stück Metall durch plötzliches Abkühlen am Ende eines Arbeitsganges durchführt.

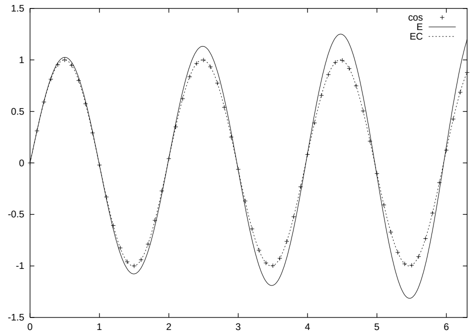


Abbildung 3.3: Ausdruck von gnuplot für $A = 1$, $\delta = 0$, $m = 1$, $k = 10$, $t_{\max} = 6.3$, und $\Delta t = 0.01$, berechnet mit dem Euler-Verfahren (E) im Vergleich mit der exakten Lösung aus Abb. 3.2, und dem Euler-Cromer Verfahren (EC).

3.1.2 Die Euler-Cromer-Methode

Eine besseres Ergebnis erhält man mit der Euler-Cromer-Methode. In der Zeile:

```
x[i]=x[i-1]+v[i-1]*dt;          // next position
```

ersetzt man den alten Wert `v[i-1]` durch den neuen Wert `v[i]`, der ja gerade eben berechnet wurde. Dies ist erlaubt, da durch die Diskretisierung zu jedem Wert ein Bereich von Δt gehört. In diesem Fall ist die numerische Lösung viel besser, wie man leicht sehen kann wenn man die analytische Lösung mit der numerischen vergleicht. Ausserdem ist beim Euler-Cromer-Verfahren die Energie (bis auf kleine Schwankungen) besser erhalten.

3.2 Das Pendel

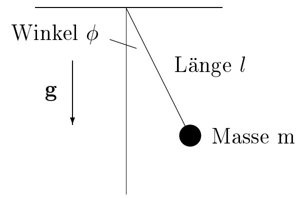


Abbildung 3.4: Das Pendel mit masseloser Aufhängung

Bei einem Pendel ist die (punktförmige) Masse m an einem starren, masselosen Stab der Länge l aufgehängt und schwingt reibungsfrei im Schwerefeld $\mathbf{g}$. $\varphi(t)$ ist der Auslenkwinkel des Pendels zur Zeit t aus der Ruhelage heraus. Um die Verlaufsfunktion aufzustellen benutzen wir den Energieerhaltungssatz: Die Gesamtenergie E, besthend aus kinetischer und potentieller Energie bleibt erhalten, d.h. $\dot{E} = 0$. Der maximale Auslenkwinkel ist dann $\varphi_{\max}$, mit der zugehörigen Winkelgeschwindigkeit $\omega(\varphi_{\max}) = \dot{\varphi}_{\max} = 0$. Die Gesamtenergie ist also

$$E = \frac{1}{2} m l^2 \dot{\varphi}^2 - mgl \cos\varphi = -mgl \cos\varphi_{\max} \ . \tag{3.6}$$

Die analytische Lösung erhält man durch Auflösen nach $\dot{\varphi}$

$$\dot{\varphi}^2 = \frac{2g}{l}(\cos\varphi - \cos\varphi_{\max}) \ , \tag{3.7}$$

Wurzel ziehen und Inversion

$$\frac{dt}{d\varphi} = \frac{1}{\sqrt{\frac{2g}{l}(\cos\varphi - \cos\varphi_{\max})}} \ , \tag{3.8}$$

Trennung der Variablen t und φ

$$dt = \frac{d\varphi}{\sqrt{\frac{2g}{l}(\cos\varphi - \cos\varphi_{\max})}} \ , \tag{3.9}$$

und anschließender Integration von der Zeit t_0 bis t und den entsprechenden Winkeln φ_0 bis $\varphi(t)$

$$t(\varphi) - t_0 = \sqrt{\frac{l}{2g}} \int_{\varphi_0}^{\varphi(t)} \frac{d\psi}{\sqrt{\cos\psi - \cos\varphi_{\max}}} \ . \tag{3.10}$$

Somit kann man dann die Schwingungsdauer T des Pendels bestimmen. Für eine Periode benötigt das Pendel viermal die Dauer der Bewegung von $\varphi_0 = 0$ nach $\varphi_{\max}$. Damit ergibt sich

$$T = 4\sqrt{\frac{l}{2g}} \int_0^{\varphi_{\max}} \frac{d\psi}{\sqrt{\cos\psi - \cos\varphi_{\max}}} \ . \tag{3.11}$$

Der Integrand divergiert, sobald $\psi \to \varphi_{\max}$. Das Integral ist ein sog. elliptisches Integral ohne geschlossene analytische Lösung. In einem der folgenden Kapitel wird die numerische Integration solcher Funktionen behandelt werden. Die Lösungen der elliptischen Integrale sind für die verschiedenen Gattungen tabelliert. Gleichung (3.11) läßt sich durch Variablensubstitution $\sin\psi^* = \frac{\sin(\psi/2)}{\sin(\varphi_{\max}/2)}$ auf ein elliptisches Integral erster Gattung in der Legendreschen Normalform zurückführen:

$$F(k, \varphi) = \int_0^{\varphi} \frac{d\psi}{\sqrt{1 - k^2 \sin^2\psi}} \ , \tag{3.12}$$

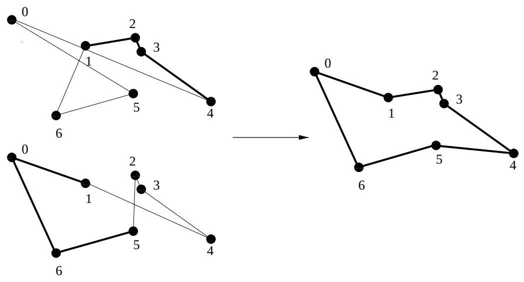


Abbildung 9.4: Durch Rekombination von (Teil-) Lösungsvorschlägen kann man zu günstigeren Lösungen gelangen. Diese Darstellung ist nur schematisch, die Ziffern beziehen sich nicht auf das Beispiel im Text

noch suboptimal ist, denn man erkennt durch näheres Betrachten leicht, daß sich der Lösungsvorschlag mehrfach um das Bildzentrum herumwindet. Die verwendete Implemetierung des genetischen Algorithmus ist nicht (oder nur nach sehr langer Zeit) in der Lage, aus diesem “Nebenminimum” zu entweichen. Aus der Entwicklung der Fitness-Funktion mit der Zeit sieht man, wie das “beste Individuum” immer wieder Mutationen oder crossover-Vorgängen zum Opfer fällt, der Vorrat an guten Genen in der Population jedoch schnell wieder das alte Niveau der Fitness Funktion erreicht.

Fortpflanzung). Dabei sollen ganze Gensequenzen aus einem Individuum entnommen und unter Beibehaltung möglichst vieler Information mit dem Genom des anderen Individuums verschmolzen werden. In unserem Falle sei etwa das Elternpaar

$$\begin{array}{cccccccccc} (& 1 & 5 & | & 4\ 3\ 2 & | & 6 & 7 &) \\ (& 3 & 4 & | & 7\ 6\ 5 & | & 1 & 2 &) \end{array}$$

gegeben, in dem wir bereits durch senkrechte Striche zufällig bestimmte Schnittstellen gekennzeichnet haben.

Wir möchten hieraus zwei Nachfahren mit ausgetauschten Genen erhalten, etwa

$$\begin{array}{lccccccccc} \text{1. Nachfahr:} & (& . & . & | & 7\ 6\ 5 & | & . & . &) \\ \text{2. Nachfahr:} & (& . & . & | & 4\ 3\ 2 & | & . & . &). \end{array}$$

Die ursprüngliche Sequenz wird am besten erhalten, wenn man die Leerstellen sukzessive mit ursprünglichem Genom auffüllt, wobei die bereits festliegende, übernommene Sequenz *nicht* erneut berücksichtigt wird (eingeklammerte Zahlen) und die anderen "Gene" der Reihe nach eingesetzt werden:

- 1. Nachfahr:

$$\begin{array}{lccccccccc} \text{1. Elternteil:} & (& 1 & (5) & | & 4\ 3\ 2 & | & (6) & (7) &) \\ \Rightarrow \text{Kind:} & (& 1 & 4 & | & 7\ 6\ 5 & | & 3 & 2 &) \end{array}$$

- 2. Nachfahr:

$$\begin{array}{lccccccccc} \text{2. Elternteil:} & (& (3) & (4) & | & 7\ 6\ 5 & | & 1 & (2) &) \\ \Rightarrow \text{Kind:} & (& 7 & 6 & | & 4\ 3\ 2 & | & 5 & 1 &) \end{array}$$

Die hinter diesem Verfahren steckende Idee ist, daß durch diese Rekombination günstige globale Konstellationen zwischen 2 Individuen ausgetauscht werden können und nicht nur lokale Änderungen wie bei Mutationen vorgenommen werden. Zusammenfassend läßt sich sagen, daß Evolutionsalgorithmen Beispiele sind für mehrskalige Verfahren, die Manipulationen auf lokalem Niveau wie die Mutation, die nur die Reihenfolge zweier anzufahrender Städte ändert, mit grobskaligen Manipulationen wie dem betrachteten crossover Mechanismus kombiniert. Eine solche Mehrskaligkeit oder auch ein hierarchischer Aufbau ist Kennzeichen vieler guter Algorithmen. Ein aktuelles Beispiel sind Mehrgitterverfahren zur Lösung großer linearer Gleichungssysteme oder auch der `quicksort` Sortieralgorithmus (vgl. Übungen in der ersten Kurshälfte).

Im Bild 9.3 zeigen wir noch als Beispiel einen Programmlauf eines genetischen Algorithmus für das Handelsreisendenproblem. Es fällt auf, daß auch der letzte Lösungsvorschlag

wobei $k = \sin(\varphi_{\max}/2)$ ist. Ein weiteres Beispiel ist die Normalform der 2.Gattung

$$E(k,\varphi) = \int_0^\varphi d\psi \sqrt{1 - k^2 \sin^2 \psi} \tag{3.13}$$

Für die numerische Lösung der Pendelbewegung werden wir die Energieerhaltungsgleichung (3.6) umformen, um wieder die Gleichung eines harmonischen Oszillators zu erhalten. Durch Ableitung von Glg. (3.6) ergibt sich

$$\dot{E} = 0 = ml^2 \dot{\varphi}\ddot{\varphi} + mgl\dot{\varphi}\sin\varphi \tag{3.14}$$

und nach Umformung erhält man die nichtlineare Gleichung:

$$\ddot{\varphi} = -\frac{g}{l}\sin\varphi \ . \tag{3.15}$$

In der Nährung für kleine $\varphi \ll 1$ ist $\sin\varphi \approx \varphi$ und man erhält die lineare Differentialgleichung

$$\ddot{\varphi} = -\frac{g}{l}\varphi \ , \tag{3.16}$$

die mathematisch identisch zu Gleichung (3.1) ist. Diese kann durch einfaches Umbenennen der Parameter wieder numerisch mit den Verfahren aus dem letzten Abscnitt gelöst werden. Die Anfangsbedingungen seien z.B. $\varphi(0) = \varphi_{\max}$ und $\dot{\varphi}(0) = 0$.

3.2.1 Die Verlet-Methode

Ein verbessertes Verfahren, zur Lösung von Gleichung (3.15) ist die Verlet-Methode. Sie ist allerdings nur gültig, wenn die Kräfte unabhängig von der Geschwindigkeit sind. Zur Herleitung schreibt man die Taylor-Entwicklung von $\varphi(t+\Delta t)$ und $\varphi(t-\Delta t)$ auf:

$$\varphi(t+\Delta t) = \varphi(t) + \Delta t\omega(t) + \frac{\Delta t^2}{2}a(t) + \cdots \tag{3.17}$$

$$\varphi(t-\Delta t) = \varphi(t) - \Delta t\omega(t) + \frac{\Delta t^2}{2}a(t) + \cdots \tag{3.18}$$

wobei $a(t) = \dot{\omega}(t) = -\frac{g}{l}\sin\varphi$ verwendet wurde. Addiert man die beiden Gleichungen (3.17) und (3.18) ergibt sich

$$\varphi(t+\Delta t) = 2\varphi(t) - \varphi(t-\Delta t) + \Delta t^2 a(t) + \cdots \tag{3.19}$$

Der Vorteil liegt darin, daß durch die Addition die ungeraden Potenzen von Δt wegfallen und damit der Fehler auf die Größenordnung $O(\Delta t^4)$ schrumpft, anstatt wie zuvor bei $O(\Delta t^2)$ zu liegen. Für die Formel benötigt man nun zwei Werte der $\varphi(t)$–Funktion, den Anfangswert $\varphi(0) = \varphi_0$ und z.B. den Wert $\varphi(-\Delta t)$, welcher sich wiederum durch den Differenzenquotienten $\dot{\varphi}(0) = \frac{\varphi_0 - \varphi(-\Delta t)}{\Delta t}$ näherungsweise errechnet zu:

$$\varphi(-\Delta t) = \varphi_0 - \Delta t\dot{\varphi}_0 \ . \tag{3.20}$$

Wir haben wieder das übliche Berechnungsprogramm für die numerische Lösung:

```
   const int nx=10001;   // fieldlength
   double p[nx], t;

// parameter definition
   double g, l, m;     // g=gravity, l=length of pendulum, m=mass
   double p0, dp0;     // p0=initial angle, dp0=initial angular velocity
   double t_max, dt;   // t_max=total time, dt=time step

// read parameters
   cout << "Gravitation    ";  cin >> g;
   cout << "Pendellaenge   ";  cin >> l;
   cout << "Masse          ";  cin >> m;
   cout << "Anf.Auslenkung ";  cin >> p0;
   cout << "Anf.Geschw.    ";  cin >> dp0;
   cout << "Gesamtzeit     ";  cin >> t_max;
   cout << "Zeitintervall  ";  cin >> dt;
   double c=-(g/l)*dt*dt;

// make sure that the number of intervals is smaller than nx
   if( t_max/dt >= nx) {
      cout << "ERROR: dt is too small, set to: " << t_max/nx << '\n';
      return -1;
   }

// setup initial conditions
   p[0]=p0-dt*dp0;
   p[1]=p0;
// loop from t=0 to t=t_max
   for( int i=2; i<t_max/dt; i++ ) {
        p[i]=2.*p[i-1]-p[i-2]+c*sin(p[i-1]);
   }

// output to file
   ofstream outfile("verletp.dat");
   t=0.0;
   for( int i=0; i<=t_max/dt; i++ ) {
      outfile << t << ' ' << p[i+1] << '\n';
      t=t+dt;
   }
   outfile.close();
```

Dabei wird der Anfangsindex auf `i=2` verschoben, um negative Indizes zu vermeiden. Mit den Parametern $g = 10$, $l = m = 1$, $\varphi_0 = 0.1$, $\dot{\varphi}_0 = 0$, $t_{\max} = 7.5$ und $\Delta t = 0.01$, erhält man bereits das Bild einer harmonischen Schwingung. Setzt man die Anfangsbedingung

Evolution hervorbringt, wird vielmehr durch die intrinsischen Regeln der Natur selbst bestimmt und nicht durch eine von außen festgelegte Gütefunktion.

Im Handelsreisendenproblem auf dem Computer wird entsprechend ein Genom als eine der möglichen Permutationen der Indizes $1...N$ der anzusteuernden Punkte angesehen (hier haben wir wieder den Ausgangsort 0 der Reise festgehalten). Die Genotypen haben dann folgende Form (hier ein Beispiel mit $N = 5$):

$$(1\ 2\ 5\ 3\ 4\ )$$

Die Fahrtroute (der Phänotyp), der diesem Genom entspricht, wäre die Rundreise

$$\mathbf{x}_0 \to \mathbf{x}_1 \to \mathbf{x}_2 \to \mathbf{x}_5 \to \mathbf{x}_3 \to \mathbf{x}_4 \to \mathbf{x}_0.$$

Die Fitness-Funktion eines Phänotyps ist die Länge der Fahrtstrecke,

$$\sum_{i=1}^{N-1} ||\mathbf{x}_{i+1} - \mathbf{x}_i||. \tag{9.4}$$

Mutationen bestehen in einer punktuellen, lokalisierten Änderung des Genoms, in unserem Falle etwa dem zufälligen Vertauschen zweier Positionen im N-Tupel. Dabei kann es allein schon durch durch diese Vertauschung zu einer Verbesserung des Lösungsvorschlages kommen (s. Abb. 9.3), aber die eigentliche Bedeutung der Mutation zeigt sich erst in

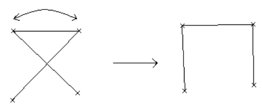


Abbildung 9.3: Durch Vertauschen (Mutation) erhält man eine bessere Lösung

Kombination mit einen Selektionsvorgang.

Eine Population bestehe etwa aus M Individuen. Selektion kann dann dadurch implementiert werden, daß zwei zufällige Genotypen ausgewählt werden, deren Fitness-Funktion berechnet wird, worauf dann (i) das schlechtere Genom verworfen, das bessere jedoch in der Population verdoppelt wird. Damit werden die schlechten Lösungsvorschläge aussortiert und die guten entsprechend vermehrt. So verbessert sich die Qualität der Lösungsvorschläge langsam.

Eine große Beschleunigung des Optimierungsprozesses kann man erreichen, wenn man das Genom zweier Individuen miteinander kombiniert (crossover, Rekombination, sexuelle

Die einfachste Technik zur Lösung des Problems ist natürlich das Durchprobieren aller möglichen N-tupel. Allerdings wächst die Anzahl dieser N-tupel wie $N!$ und führt daher schnell zu astronomisch hohen Rechenzeitbedürfnissen für eine solche Strategie. Daher benötigen wir einen intelligenteren Algorithmus. In Ermangelung einer allgemeinen Lösungsvorschrift, die in kürzerer Zeit zum Erfolg führt, wählen wir an dieser Stelle sogenannte *heuristische* Verfahren, die im wesentlichen auf ein *gezieltes Raten* der Lösung hinauslaufen. Um solche Algorithmen zu finden, orientiert man sich häufig an natürlichen Vorgängen. Man findet in der Literatur u.a.

1. genetische (Evolutions-)Algorithmen
2. das *simulated annealing* ("Simulierte Abkühlung")
3. Neuronale Netze (Mustererkennung im Hopfield-Modell als Bestimmung eines der Grundzustände eines Spinglases)

Diese Verfahren werden nun im folgenden behandelt.

9.3 Genetische Algorithmen

Hier ist die Grundidee, die Wirkungsweise der Evolution zu kopieren. Eine Gegenüberstellung von Biologie und informatischem Vorgehen kann in diesem Falle so aussehen:

Biologie	**Informatik**
Problem der "bestmöglichen" Lebensbewältigung	konkretes "genetisch codiertes" Problem
"Evolutionsziel"	problemspezifische, zu optimierende *Fitness*-Funktion
Individuum (Phänotyp)/Genotyp	Lösungsvorschlag/dessen abstrakte Formulierung
Population	Pool von Lösungsvorschlägen
Mutation, Rekombination,	problemspezifische zufällige Änderungen an Individuen, Kombination des Informationsgehaltes verschiedener Individuen
Selektion	Verwerfen "schlechter" Lösungsvorschläge

Man sollte diesen Vergleich natürlich nicht zu weit treiben, denn es gibt kein Ziel oder keinen Zweck der Evolution (Die Natur ist kausal, aber nicht zweckbestimmt). Was die

höher, z.B. $\varphi_0 = 3$, so ergibt sich ein deutlich anderes Schwingverhalten, das nicht mehr durch einen einfachen Cosinus genähert werden kann, wie in Abb. 3.5 zu sehen.

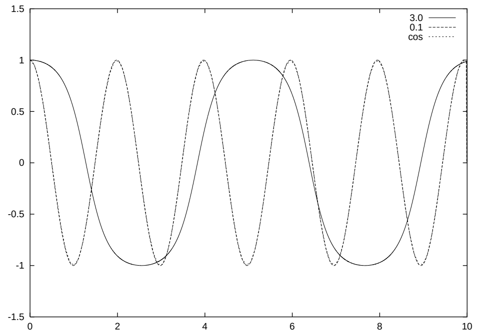


Abbildung 3.5: Ausdruck von $\varphi(t)/\varphi_0$ für $g = 10$, $l = 1$, $m = 1$, $\dot{\varphi}_0 = 0$, $t_{\max} = 10$ und $\Delta t = 0.01$, berechnet mit dem Verlet-Verfahren für $\varphi_0 = 0.1$ und $\varphi_0 = 3$, verglichen mit $\cos(\sqrt{10}t)$.

3.2.2 Das allgemeine Pendel

Zu dem speziellen Pendel aus dem vorherigen Kapitel fügen wir noch eine zur Geschwindigkeit proportionale Reibungskraft $-q\dot{\varphi}$ und eine treibende Kraft $F = F_0 \sin(\Omega t)$ hinzu, wobei die Frequenz Ω nicht gleich der Eigenfrequenz des Pendels sein darf ($\Omega \neq \omega_0 = \sqrt{\frac{g}{l}}$). Somit existieren die zusätzlichen Parameter q, F_0 und Ω. Das dazugehörige Bewegungsgesetz lautet

$$\ddot{\varphi} = -\frac{g}{l}\sin\varphi - q\dot{\varphi} + F_0 \sin\Omega t \tag{3.21}$$

Zur Lösung dieser Gleichung ist das Verlet-Verfahren nicht mehr geeignet, da eine Kraft von der Geschwindigkeit abhängt. Deswegen wird ein neues Verfahren zur numerischen Lösung von Gleichungen vorgestellt.

3.2.3 Das Runge-Kutta-Verfahren

Als Beispiel wird jetzt von einer Gleichung $\dot{\mathbf{x}} = f[\mathbf{x}, t]$ ausgegangen, wobei $\mathbf{x}$, einen Vektor, bestehend aus verschiedenen Größen, darstellen kann. Beim Euler-Verfahren wurde der

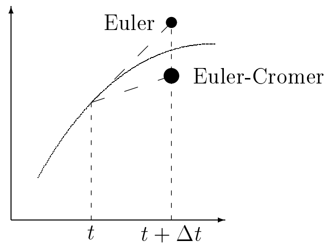


Abbildung 3.6: Fehler beim Euler- / Euler-Cromer-Verfahren

nächste Kurvenstützpunkt mit Hilfe der Steigung des alten Stützpunktes, beim Euler-Cromer-Verfahren mit der Steigung des neuen Stützpunktes berechnet. Beide Verfahren besitzen einen relativ großen Fehler (siehe Abb. 3.6).

Um die Genauigkeit zu verbessern wird beim Runge-Kutta-Verfahren die Steigung in der Mitte des Intervalls bestimmt. Der Funktionswert $\bar{x}$ in der Intervallsmitte wird durch eine Taylorentwicklung berechnet

$$\bar{x} = x\left(t + \frac{\Delta t}{2}\right) = x(t) + \frac{1}{2}\Delta t\ f[x(t), t]. \tag{3.22}$$

Dieser Zwischenwert wird dann in die allgemeine Formel eingesetzt.

$$x(t + \Delta t) = x(t) + \Delta t\ f\left[x\left(t + \frac{\Delta t}{2}\right), t + \frac{\Delta t}{2}\right] \tag{3.23}$$

Für die Anwendung auf das Pendel benötigen wir wieder die beiden Gleichungen

$$\dot{\omega} = f[\varphi, \omega, t] = -\frac{g}{l}\sin\varphi - q\omega + F_0 \sin\Omega t \tag{3.24}$$
$$\dot{\varphi} = \omega \tag{3.25}$$

Für das Runge-Kutta-Verfahren (R-K) werden dann die Zwischenwerte $\bar{\omega}$ und $\bar{\varphi}$ berechnet.

$$\bar{\omega} = \omega(t) + \frac{\Delta t}{2}\ f[\varphi(t), \omega(t), t] \tag{3.26}$$
$$\bar{\varphi} = \varphi(t) + \frac{\Delta t}{2}\ \omega(t) \tag{3.27}$$

Die Lösungen ergeben sich dann durch das R-K-Verfahren.

$$\varphi(t + \Delta t) = \varphi(t) + \Delta t\ \bar{\omega} \tag{3.28}$$
$$\omega(t + \Delta t) = \omega(t) + \Delta t\ f\left[\bar{\varphi}, \bar{\omega}, t + \frac{\Delta t}{2}\right] \tag{3.29}$$

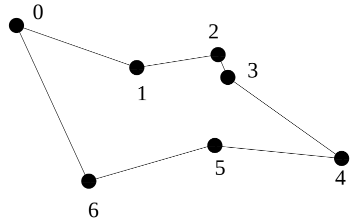


Abbildung 9.2: Handelsreisenden-Problem: Welches ist der kürzeste Weg einer Rundreise, die alle gezeigten Punkte genau einmal aufsucht?

Rundreise verschiedene verstreut liegende Städte besuchen muß und dabei den kürzesten Weg finden will. Die praktische Bedeutung dieser Fragestellung für logistische Probleme liegt auf der Hand. Etwas abstrakter formuliert suchen wir wie in den obigen Fällen wieder das Minimum einer reellwertigen Funktion, die hier allerdings nicht die Energie, sondern die Länge einer Rundreise ist. Wie im Beipiel mit den magnetischen Spins ist der Zustandsraum diskret: er enthält alle $N-1$-tupel mit Permutationen der Zahlen $(1, \ldots, N)$, hat also $(N-1)!$ Elemente. Hier zählen wir ab 1, weil wir o.B.d.A. den Anfangsort der Reise nach 0 verlegt haben.

Wir wollen die Eigenschaften zusammenfassen, die uns bei den beiden geschilderten Problemen begegnet sind und die wir bei der Ermittlung eines "energetisch günstigsten" oder anderweitig optimalen Zustandes berücksichtigen müssen:

- man arbeitet in einem extrem hochdimensionalen Suchraum. Wenn wir in den obigen Beispielen einfach alle Zustände absuchten, so müßten alle 2^N Möglichkeiten für Spinsysteme mit Spins von 2 Zuständen oder $(N-1)!$ im Handelsreisendenproblem berücksichtigt werden, einen Zustandsvektor mit N Elementen (den Positionen der Spins/Städte) zu füllen.

- die gesuchten Minima sind u.U. sehr zahlreich und liegen energetisch nahe beisammen. Selbst wenn ein eindeutiger energetisch günstigster Zustand existiert, so existieren möglicherweise viele Nebenminima fast gleicher Energie, die ggfs. wichtige physikalische Information tragen.

- die Energiefunktion ist nichtlinear. Eine quadratische Energiefunktion ist zwar im Prinzip für eine Optimierung geeignet (siehe auch das Beipiel der Bestimmung der Gewichte in mehrschichtigen Netzwerken im folgenden Kapitel) und führt in der Regel auf ein Problem aus der linearen Algebra, jedoch sind in unserem Falle zudem noch

- die Zustände diskret, so daß Methoden der Differentialrechung (Gradientenabstieg) nicht anwendbar sind.

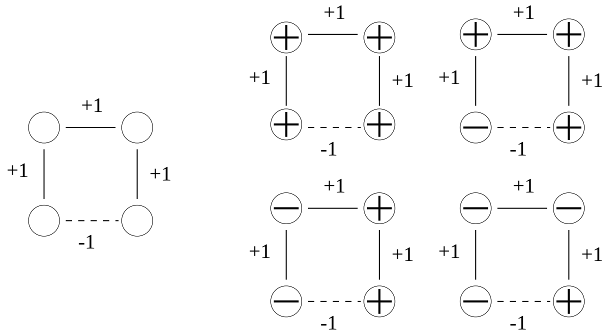


Abbildung 9.1: Spingläser: links ist eine mögliche Anordnung von Kopplungsstärken J_{ij} gezeigt, die nur nächste Nachbarn verbindet. Rechts sind die Hälfte aller möglichen Zustände geringster Energie gezeichnet. Die Konstruktion geschah dadurch (oben links), daß in der rechten unteren Ecke mit einem *up*-Spin begonnen wurde und dann zunächst entgegen dem Uhrzeigersinn soviele Bindungen wie möglich befriedigt, d.h. $J_{ij}s_is_j > 0$ gewählt wurde. Dies scheitert an der unteren Kopplung, die sich nicht mehr befriedigen läßt. Danach wurde einfach die Lage der nicht befriedigten Bindung variiert, was zu vier energetisch gleichwertigen Zuständen führt. Zu jeder gezeigten Konfiguration existiert auch noch diejenige mit invertierten Spins, die die gleiche Energie aufweist. In diesem Beispiel gibt es also bereits 8 Zustände gleicher (minimaler) Energie.

In der Hochtemperaturphase zeigen Spingläsern wie auch Ferromagnete keine spontane Magnetisierung, d.h. keine Ausrichtung der Spins überwiegend in eine Richtung. Sie weisen den üblichen Diamagnetismus auf, ggf. überlagert von paramagnetischen (das üßere Magnetfeld verstärkende) Effekten. In der Tieftemperaturphase zeigen Spingläser im Unterschied zum einfachen Isingmodel (und zu Ferromagneten) jedoch keine spontane Magnetisierung sondern “frieren” in einem der möglichen energetisch günstigen Zustände ein. Ein Übergang in einen anderen, energetisch ebenso günstigen Nebenzustand wird häufig durch Energiebarrieren kinetisch unterdrückt, so wie das auch in amorphen Phasen von Mischungen von SiO_2 mit ionischen Beimischungen auftritt, die gemeinhin als *Gläser* bezeichnet werden.

Wir haben uns hier den Ising-ähnlichen Modellen so ausführlich gewidmet, weil sie in der statischen Mechanik sehr wichtig sind und daher auch Gegenstand einer Übung sein werden.

9.2.2 Das Handelsreisendenproblem

Ein weiteres, vielleicht anschaulicheres Beispiel ist das sogenannte *traveling salesman* Problem (Handelsreisendenproblem). Die Aufgabe besteht darin, daß ein Händler auf einer

Im folgenden Programmsegment wird zuerst $\bar{\omega}$ berechnet und in einer Zwischenvariablen `obar` gespeichert. Damit wird der neue Winkel $\varphi(t + \Delta t)$ berechnet, $\bar{\varphi}$ wird in einer Zwischenvariablen `pquer` abgelegt und zur die Berechnung von ω benutzt. Die Zeit t wird zweimal um $\frac{\Delta t}{2}$ erhöht und somit bei jedem Durchlauf um Δt.

```
   const int nx=10001;  // fieldlength
   double p[nx], dp[nx], t;

// parameter definition
   double g, l;        // g=gravity, l=length of pendulum
   double q, om, am;   // q=friction, om=driving frequency, am=amplitude
   double p0, dp0;     // p0=initial angle, dp0=initial angular velocity
   double t_max, dt;   // t_max=total time, dt=time step
   double fp, obar, pbar;

// read parameters
   cout << "Gravitation    ";  cin >> g;
   cout << "Pendellaenge   ";  cin >> l;
   cout << "Reibung        ";  cin >> q;
   cout << "Anreg.Frequenz ";  cin >> om;
   cout << "Anreg.Amplitude";  cin >> am;
   cout << "Anf.Auslenkung ";  cin >> p0;
   cout << "Anf.Geschw.    ";  cin >> dp0;
   cout << "Gesamtzeit     ";  cin >> t_max;
   cout << "Zeitintervall  ";  cin >> dt;
   double c=g/l;
   double dt2=dt/2.;

// make sure that the number of intervals is smaller than nx
   if( t_max/dt >= nx) {
      cout << "ERROR: dt is too small, set to: " << t_max/nx << '\n';
      return -1;
   }

// setup initial conditions
   p[0]=p0;
   dp[0]=dp0;
// loop from t=0 to t=t_max
   t=0.0;
   for( int i=1; i<t_max/dt; i++ ) {
        fp=-c*sin(p[i-1])-q*dp[i-1]+am*sin(om*t);
        obar=dp[i-1]+dt2*fp;
        p[i]=p[i-1]+dt*obar;
```

```
      t=t+dt2;
      pbar=p[i-1]+dt2*dp[i-1];
      fp=-c*sin(pbar)-q*obar+am*sin(om*t);
      dp[i]=dp[i-1]+dt*fp;

      t=t+dt2;
   }

// output to file
   ofstream outfile("ruku.dat");
   t=0.0;
   for( int i=0; i<=t_max/dt; i++ ) {
      outfile << t << ' ' << p[i+1] << ' ' << dp[i+1] << '\n';
      t=t+dt;
   }
   outfile.close();
}
```

3.2.4 Chaos

Um die Ergebnisse des Progamms besser untersuchen zu können, legen wir folgende Werte fest: $g = l = 9.81$, $\Omega = 2/3$, $q = 0.5$. Die einzigen jetzt noch frei wählbaren Variablen sind: F_0 und φ_0. In Abbildung 3.7 sehen wir den zeitlichen Verlauf des Auslenkwinkels φ für verschiedene Zeitschritte Δt und verschiedene Amplituden F_0 zwischen 0 und 1.3.

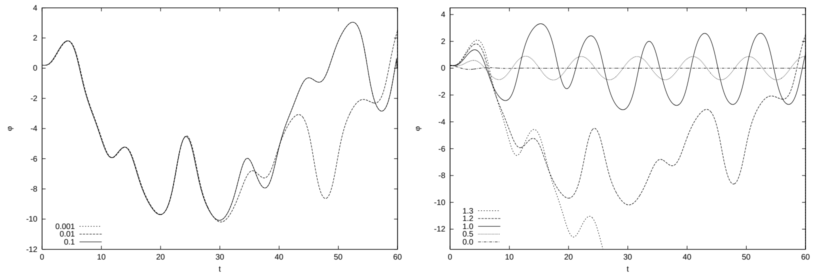


Abbildung 3.7: Links: Pendelausschlag φ als Funktion der Zeit t für verschiedene Δt. Rechts: φ als Funktion von t für verschiedene F_0.

Ist die Genauigkeit $\Delta t \leq 0.01$ so fallen die Kurven aufeinander, bei größerem Δt stellt man starke Abweichungen fest. Ohne externe Anregung $F_0 = 0$, kommt das Pendel aufgrund der Reibungskraft q schon nach kurzer Zeit zum Stillstand. Für einen kleinen Wert, zum Beispiel $F_0 = 0.5$, entsteht das gleichmäßige Schwingungsbild einer periodischen

“up” und “down”) bewirken. In magnetischen Systemen können beide Fälle auftreten, da die Spins häufig durch quantenmechanische Korrelationen (Pauli-Prinzip, Austauschwechselwirkung) gekoppelt sind.

Wir wollen wie oben angedeutet annehmen, daß sich die Energiefunktion $H\{s_i\}$ des Systemes schreiben läßt als die Summe aller Paar-Spin-Energien

$$H\{s_i\} = -\frac{1}{2}\sum_{i\neq j} J_{ij} s_i s_j, \quad s_i = \pm 1. \tag{9.1}$$

Je nach der Wahl der J_{ij} resultiert ein sehr unterschiedliches Systemverhalten.

In Isings ursprünglichem Modell gilt

$$J_{ij} = \begin{cases} 1 & \text{für direkte Nachbarn} \\ 0 & \text{sonst} \end{cases} \tag{9.2}$$

Der Grundzustand ist der Zustand minimaler Energie H. Dieser liefert bei $T \to 0$ in der *Zustandssumme* $Z = \sum_{\{s_i\}} \exp(-\beta H\{s_i\})$ den dominanten Beitrag. Hier haben wir mit β die inverse Temperatur $\beta = 1/kT$ bezeichnet und die Summation über die Menge $\{s_i\}$ aller möglichen 2^N Spinkonfigurationen asugeführt, wobei N die Zahl der Gitterplätze ist. Es gibt zwei solcher Grundzustände, in denen jeweils ausnahmslos alle Spins nach oben bzw. unten zeigen.

Nebenbemerkung: In der Folgezeit zeigten Experimente, daß die Eigenschaften seines Modelles in der Nähe des Para-/Ferromagnetüberganges in 2D geeigneter war, um Flüssigkeitsverhalten in der Nähe eines kritischen Punktes zu beschreiben als die Magnetisierungeigenschaften eines Magneten. Trotzdem ist dieses Modell eines der wichtigsten in der statistischen Mechanik, weil es sich aufgrund seiner Einfachheit häufig noch als Ausgangspunkt analytischer Betrachtungen eignet. Für Isings ursprüngliches Modell gelang es Lars Onsager, die Zustandssumme und damit alle thermodynamischen Eigenschaften in 2D analytisch zu bestimmen.

9.2.1 Spingläser

Sherrington und Kirkpatrick haben Spin*gläser* betrachtet, indem sie die Kopplung J_{ij} als eine Zufallszahl mit

$$J_{ij} \in [-1, 1] \quad \forall\, i, j \tag{9.3}$$

gewählt haben. Das besondere bei diesem Spinmodell ist eine mögliche Entartung des Grundzustandes. Das heißt, daß viele (bei unendlich großen Systemen auch unendlich viele) sehr unterschiedliche Zustände mit großen Abständen im Konfigurationsraum aber kleinen (bzw. gar keinen) Energieunterschieden auftreten. Ein sehr vereinfachtes Spinglas ist in der Abbildung 9.1 gezeigt,

- Suche nach dem quantenmechanischen Grundzustand (Wellenfunktion mit minimaler Energie) eines Systems: Variationsrechnung.
- Bestimmung der Dynamik eines Systems aus Extremalprinzipien:
  - Lagrange-Funktion(al) (Klassische Mechanik),
  - Fermatsches Prinzip (Strahlenoptik: Lichtlaufzeit ist extremal),
  - Thermodynamische Gleichgewichtszustände sind durch extremale freie Energien nach Gibbs oder Helmholtz gekennzeichnet, Nichtgleichgewichtsvorgänge zeichnen sich häufig durch maximal Entropieproduktion aus, usw.

Die aufgelisteten Probleme unterscheiden sich im wesentlichen durch die Anzahl der auftretenden Freiheitsgrade. Während das Konservendosenproblem und die Kurvenanpassung nur die Betrachtung einer kleinen Anzahl von Parametern erfordern, für die "fertige" Algorithmen existieren (siehe z.B. im bereits mehrfach erwähnten *Numerical Recipes*), so ist die Bestimmung eines quantenmechanischen Grundzustandes ein Problem mit vielen Parametern, denn die unbekannte Wellenfunktion nimmt ja prinzipiell unendlich viele Werte (einen an jedem Raumpunkt) an. Um die Probleme zu verdeutlichen, die bei derart hochdimensionalen Systemen eine Rolle spielen, wollen wir zunächst ein Standardmodell der statistischen Mechanik betrachten.

9.2 Das Ising-Modell und Verwandte

Der Versuch, das Phasenverhalten eines Systemes diskreter, klassischer, magnetischer "Spins" zu beschreiben, wurde 1924 durch Ernst Ising (geb. 10.05.1900) in seiner Doktorarbeit unternommen.

Stellen wir uns magnetische Domänen in einem dünnen Materialfilm vor und nehmen wir weiterhin an, daß die Orientierung der Domänen entweder aus der Ebene heraus oder in sie hinein gerichtet ist. Wir wollen uns diese Domänen etwa wie die zwei Spinorientierungen eines Spin-1/2 quantenmechanischen Teilchens vorstellen. Jeder "Spin" s_i soll durch einen Gitterplatz i in einem 2D Gitter und durch einen Wert von $+1$ oder -1 entsprechend "up" oder "down" repräsentiert sein.

Die Energie eines solchen Systems ist durch die Einstellung der Spins festgelegt. Im klassischen Fall (und näherungsweise meist auch im quantenmechanischen) wechselwirken Spins paarweise miteinander, zum Beispiel durch magnetische Dipol-Dipol-Wechselwirkungen, die das magnetische Moment eines Spins s_j an einem anderen Ort i als ein Feld einer Stärke $J_{ij}s_j$ zur Wirkung kommen läßt. Durch Ausrichten in Feldrichtung des Spins am Ort i kann die Energie des Systemes um den Wert $-J_{ij}s_is_j$ abnehmen. Positive Kopplungen J_ij bevorzugen energetisch gleichausgerichtete Spins (ferromagnetische Ordnung), negative können dagegen eine antiferromagnetische Ordnung (abwechselnd Spin

Bewegung. Die Frequenz entspricht jedoch nicht der Eigenfrequenz des Pendels, sondern entspricht ungefähr der Frequenz der antreibenden Kraft. Wird die Antreibkraft weiter erhöht, dann bewegt sich das Pendel nicht mehr gleichmäßig. Die Ausschläge lassen sich nicht mehr vorraussagen, das Verhalten des Pendels wird *chaotisch*. Es ist bemerkenswert, daß trotz des durch die Rechnung gegebenen Determinismus das Verhalten des Pendels chaotisch ist.

Als nächstes halten wir den Wert von F_0 fest und verändern den anfänglichen Auslenkwinkel φ_0. Im ersten Fall sei der Anfangsauslenkwinkel $\varphi_{01} = 0.2$, im zweiten Fall $\varphi_{02} = 0.201$. Der Unterschied beträgt also nur 0.5 Prozent. Die Differenz ("damage spreading" genannt) dieser beiden Auslenkwinkel $\Delta\varphi(t) = \varphi_2(t) - \varphi_1(t)$ wird nun im zeitlichen Verlauf untersucht. Die Grafiken der Abbildung 3.8 sind halblogarithmische Darstellungen, d.h. die x-Achse hat einen normalen, linearen Maßstab, während die y-Achse logarithmisch ist. Der Abstand zwischen den Auslenkungen wurde einmal mit der Euler-Cromer Methode (links) und dann mit dem Runge-Kutta Verfahren (rechts) berechnet. Die jeweils obere Kurve zeigt den chaotischen Fall: Der Fehler vergrößert sich, im Mittel auf einer steigenden Geraden. Eine Linie mit positiver Steigung bedeutet in der halblogarithmischen Darstellung, daß die Funktion exponentiell wächst, der Abstand läßt sich also durch $\Delta\varphi \approx e^{\lambda t}$, mit $\lambda > 0$ nähern. Im periodischen Fall (untere) verringert sich der Abstand in der Darstellung linear, also wieder exponentiell, aber diesmal mit negativer Steigung $\lambda < 0$. λ wird als der Lyapunovexponent bezeichnet. Bei $\lambda = 0$ ist der Übergang vom Periodischen ins Chaotische mit $\lambda > 0$.

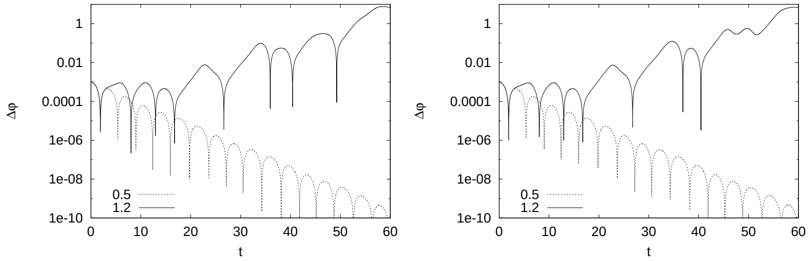


Abbildung 3.8: Damage Spreading: Der Abstand zwischen den Anfangsauslenkungen $\varphi_{01} = 0.2$ und $\varphi_{01} = 0.201$ wurde links mit Euler-Cromer, rechts mit Runge-Kutta berechnet.

3.2.5 Der Poincaréschnitt

Eine andere Darstellung der Pendelbewegung ist durch die Abbildung in einem sog. *Phasendiagramm* möglich. Ein Phasendiagramm ist ein Geschwindigkeit-Ort-Diagramm, in das die Winkelgeschwindigkeit ω über den Auslenkwinkel φ aufgetragen wird. Für den

harmonischen Oszillator entsteht dann als Bild eine Ellipse. Diese geschlossene Kurve zeigt die Energieerhaltung beim harmonischen Oszillator. Für das allgemeine Pendel entsteht im periodischen Fall nach der Einschwingzeit ebenfalls eine Ellipse (linke Grafik, Abbildung 3.9). Im chaotischen Fall weist das Phasendiagramm eine endlose Anzahl unterschiedlicher Linien auf, die irgendwann (fast) das ganze Diagramm überdeckt haben (rechte Grafik).

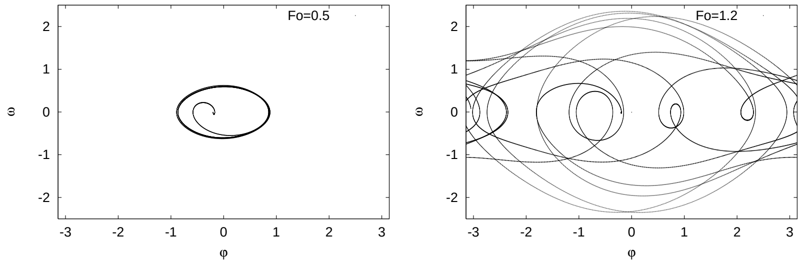


Abbildung 3.9: Phasendiagramme des allgemeinen Pendels, links für $F_0 = 0.5$, rechts für $F_0 = 1.2$.

Die Untersuchung des Bildes im chaotischen Fall ist sehr schwierig. Deswegen wird hier meist eine stroboskopische Analyse durchgeführt: Es werden nur Punkte in das Diagramm gezeichnet, die zu der gleichen Phase der angelegten periodischen Kraft vorkommen, beispielsweise für $\Omega t = 0,\ 2\pi,\ 4\pi$ etc.. Diese stroboskopischen "Schnitte" werden *Poincaréschnitte* genannt. Für den harmonischen Oszillator entsteht in dieser Darstellungsart im Diagramm nur ein einziger Punkt. Das entstehende Bild wird *Attraktor* genannt. Dieser erste Fall ist der punktförmige Attraktor. Ist die Bewegung chaotisch, dann entsteht eine Menge von Punkten. Dieses Bild heißt dann seltsamer oder fraktaler Attraktor. Für unser Pendel ist dies in der Abbildung 3.10 links zu sehen. Als drittes gibt es noch die geschlossenen Attraktoren: ein geschlossener Linienzug im Poincaréschnitt. Für ein Doppelpendel (ein Pendel, das an der Masse eines anderen Pendels aufgehängt ist) ensteht ein Diagramm, bei dem gleichzeitig zwei verschiedene Attraktoren, d.h. zwei verschiedene Fälle im gleichen Phasenraum, gezeigt werden können, ein geschlossener und ein fraktaler Attraktor.

3.2.6 Übergang periodisch – chaotisch

Betrachtet man das Pendel für noch größere Antreibkräfte, entsteht wieder periodisches Verhalten. Abbildung 3.11 zeigt das Pendelverhalten für die Antreibkräfte $F_0 = 1.35$, $F_0 = 1.44$ und $F_0 = 1.465$. Es fällt auf, daß die Perioden für die letzten beiden Werte doppelt, bzw. viermal so groß sind. Die Periode wird also jeweils verdoppelt. Dieses Verhalten wird Bifurkation genannt.

Kapitel 9

Optimierung

9.1 Motivation

Optimierungsprobleme treten nicht nur in der Physik an vielen Stellen auf. Die sogenannte lineare Optimierung (Suche der optimalen Lösung eines Problems, dessen Nebenbedingungen durch eine grosse Anzahl linearer Ungleichungen gegeben sind) hat lange in den Wirtschaftswissenschaften und deren Teilgebiet *Operations Research* eine herausragende Rolle gespielt. Man kann hier zeigen, daß es ausreicht, die Ecken des durch die Nebenbedingungen in einem hochdimensionalen Raum gebildeten erlaubten Gebietes (Simplex) systematisch nach immer besseren Lösungen abzusuchen.

Eine ähnlich grundlegende Rolle für nichtlineare Optimierungsprobleme spielte das *traveling salesman* Problem, bei dem die insgesamt kürzeste Verbindung zwischen fest vorgegebenen Orten gesucht wird. Dieses letzte Problem ist eine "schwierige" Optimierungsaufgabe in dem Sinne als daß kein Algortihmus polynomialer Ausführungszeit in der Anzahl der Städte bekannt ist, der mit Sicherheit die optimale Lösung liefert.

Wer kennt darüberhinaus nicht die einfacheren Optimierungsfragen vom Typus *Gegeben sei die Form einer Konservendose, wie läßt sich dann bei gegebenem Inhalt die Oberfäche minimieren?*, die sich durch eine Kurvendiskussion einer Funktion einer Variablen bestimmen lassen. Relativ wenige Variablen kennzeichen auch Ausgleichsprobleme der Form $\sum_i |y_i - f(x_i)|^2 \stackrel{!}{=} \min$, bei denen eine vorgegebene Datengesamtheit (x_i, y_i) möglichst genau durch einen durch Fitparameter vorgegebenen, ansonsten analytischen Funktionsverlauf $f(x)$ wiedergegeben werden soll.

Im folgenden geben wir zur weiteren Motivation eine Auflistung der Bereiche, in denen in der Physik weitere Optmierungsprobleme auftreten:

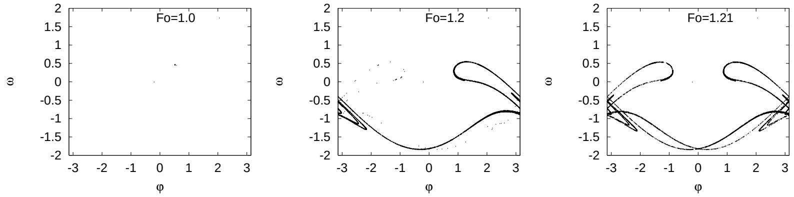


Abbildung 3.10: Poincaréschnitte für das allg. Pendel

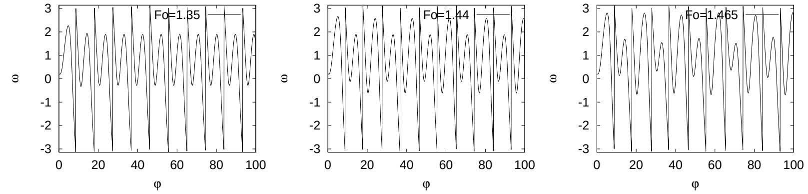


Abbildung 3.11: Unterschiedliche Perioden bei verschieden starken Antreibkräften

Das Bifurkationsdiagramm aus Abbildung 3.12 zeigt die Punkte des Poincaréschnittes in Abhängigkeit der Antreibfrequenz F_0. Die Bifurkationen sind in diesem Fall an den Stellen $F_0 \approx 1.43$ und $F_0 \approx 1.46$. Ab dem Wert $F_0 \approx 1.47$ beginnt das Chaos. Dieser Übergang zum Chaos wird auch Periodenverdopplungs– oder Kaskadenszenario genannt.

Die Längen der Perioden sind im Poincaréschnitt durch die Anzahl der Punkte gegeben. Eine zweifache Periode entspricht so z.B. einem Attraktor aus zwei Punkten. Sei jetzt f_n der Wert, an dem eine Bifurkation zu 2^n besteht, d.h. das Diagramm sich in 2^n Linien aufspaltet. Dann ist die Funktion

$$\delta_n = \frac{f_{n-1} - f_n}{f_n - f_{n+1}} \tag{3.30}$$

das Verhältnis der Abstände zwischen zwei aufeinanderfolgenden Bifurkationen. δ_n geht für große n gegen den Wert 4.669. Dieser Wert ist universell, d.h. für alle Zweier-Kaskadenszenarios derselbe. Man nennt ihn Feigenbaumkonstante.

3.3 Himmelsmechanik

Im Folgenden betrachten wir die Berechnung der Planetenbewegung, zunächst einmal der Erde, um die Sonne. Entscheidend hierbei ist die Gravitationskraft zwischen der Masse

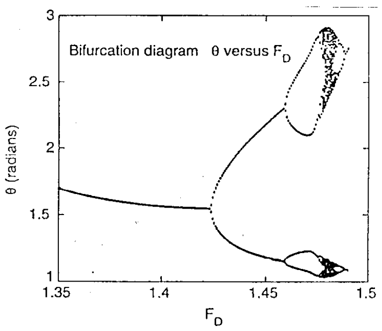


Abbildung 3.12: Bifurkationsdiagramm des Pendels

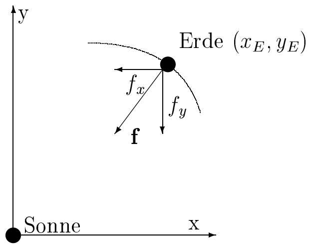


Abbildung 3.13: System Erde–Sonne

der Sonne (m_S) und der Masse der Erde (m_E). Die Formel für diese Kraft lautet :

$$\mathbf{f} = -\frac{G m_S m_E}{r^2} * \frac{\mathbf{r}}{r} \tag{3.31}$$

Der Faktor G ist die Gravitationskonstante und der Vektor $\mathbf{r} = (x, y)$ ist der Abstandsvektor zwischen den Massenschwerpunkten von Sonne und Erde. Zur Vereinfachung betrachten wir die Bewegung nur im Zweidimensionalen. Für die Berechnung benutzen wir die Formel in der Komponentenschreibweise für die x-Richtung f_x und in die y-Richtung f_y mit

$$f_x = -\frac{G m_S m_E x}{r^3}\ ,\ f_y = -\frac{G m_S m_E y}{r^3} \text{ und } r = \sqrt{x^2 + y^2} \tag{3.32}$$

Um geschicktere und angenehmere Zahlen zu bekommen, verwenden wir hier nicht das MKS-System (Meter–Kilo–Sekunde), sondern die Astronomische Einheit (1 AE= $1.5 * 10^{11}$ m≈ Abstand Erde – Sonne) für Längen und das Jahr (1 a) für die Zeit. Die Bahngeschwindigkeit bei der Kreisbewegung ist somit $v_0 = 2\pi * \frac{1\text{AE}}{1\text{a}}$. Da die Gravitationskraft

Z.B. ergibt für $N = 4$ die direkte zweimalige Anwendung von (8.60), mit der Abkürzung $W_N = \exp -i\frac{2\pi}{N}$

$$\begin{aligned} G_k = G_k^e + W_4^k G_k^o &= G_k^{ee} + W_2^k G_k^{eo} + W_4^k (G_k^{oe} + W_2^k G_k^{oo}) \\ &= f_0 + W_2^k f_2 + W_4^k (f_1 + W_2^k f_3). \end{aligned} \tag{8.63}$$

Hier kann man ablesen, daß die eo Kombinationen sich in die Indizes von f verwandeln lassen, indem man deren Reihenfolge umkehrt, jedes e durch eine 0 und jedes o durch eine 1 ersetzt und dann das Ergebnis als binäre Zahl liest. In der Regel werden FFT Algorithmen so aufgebaut, daß die f_i zunächst in eine Reihenfolge gebracht werden, die erlaubt, die $G_k^{..e}$ und $G_k^{..o}$ aus benachbarten Datenfeldern zu kombinieren (Bitreversal). Einzelheiten findet man in *Numerical Recipes* oder im Buch *Numerische Mathematik* von Schwarz. Hier sei nur kurz ein Beispiel einer FFT von 4 Werten angegeben mit den sich jeweils ergebenden Speicherinhalten.

$N = 1$	$N = 2, W_2 = e^{-i\pi} = -1$	$N = 3, W_3 = e^{-i\pi/2} = -i$
$G_0^{ee} = f_0$	$G_0^e = f_0 + f_2$	$G_0 = f_0 + f_2 + (f_1 + f_3)$
$G_0^{eo} = f_2$	$G_1^e = f_0 - f_2$	$G_1 = f_0 - f_2 - i(f_1 - f_3)$
$G_0^{oe} = f_1$	$G_0^o = f_1 + f_3$	$G_2 = f_0 + f_2 - (f_1 + f_3)$
$G_0^{oo} = f_3$	$G_1^o = f_1 - f_3$	$G_3 = f_0 - f_2 + i(f_1 - f_3)$

Tabelle 8.2: Beispiel einer FFT für 4 Werte. Durch geschicktes Sortieren der Daten vor Beginn der eigentlichen Rechnung kann man die Kombination immer längerer Transformationen stark vereinfachen. Außerdem ist es möglich, die Berechnung ohne zusätzliches Datenfeld von einer Stufe zur nächsten zu führen.

Wir wollen annehmen, daß die Anzahl N der zu transformierenden Punkte f_i eine Zweierpotenz ist. Dann gilt für alle $k = 0, \ldots N-1$

$$
\begin{aligned}
G_k &= \sum_{n=0}^{N-1} f_n) e^{-i\frac{2\pi}{N}nk} \\
&= \sum_{n \text{ gerade}}^{N-1} f_n e^{-i\frac{2\pi}{N}nk} + \sum_{n \text{ ungerade}}^{N-1} f_n e^{-i\frac{2\pi}{N}nk} \\
&= \sum_{n'=0}^{N/2-1} f_{2n'} e^{-i\frac{2\pi}{N/2}n'k} + \sum_{n'=0}^{N/2-1} f_{2n'+1} e^{-i\frac{2\pi}{N}(2n'+1)k} \\
&= \sum_{n'=0}^{N/2-1} f_{2n'} e^{-i\frac{2\pi}{N/2}n'k} + e^{-i\frac{2\pi k}{N}} \sum_{n'=0}^{N/2-1} f_{2n'+1} e^{-i\frac{2\pi}{N/2}n'k} \\
&= G_k^e + (e^{-i\frac{2\pi}{N}})^k G_k^o
\end{aligned}
\tag{8.60}
$$

Die G_k^e und G_k^o sind dabei die Transformationen der durch die geraden bzw. ungeraden Untermengen der f_i gebildeten Folgen. Zwar läuft k für die äußere Summe von 0 bis $N-1$, aber für die beiden kürzeren Transformationen entsprechen die $k > N/2 - 1$ den kleineren $k' = k - N/2$.

Falls $N = 2$ wären wir jetzt bereits fertig, denn die Transformation einer Folge mit nur einem Element ist dieses Element selbst. Aus (8.60) erhalten wir dann nach Setzen von $k = 0, 1$

$$G_0 = f_0 + f_1 \tag{8.61}$$
$$G_1 = f_0 + (e^{-i\frac{2\pi}{2}})^1 f_1 = f_0 - f_1 \tag{8.62}$$

Über den Rechenaufwand können wir jetzt folgendes sagen. Die Bestimmung der G_k aus den $G_k^{e,o}$ erfordert N Operationen (bestehend aus je einer Addition und einer Multiplikation). Die $G_k^{e,o}$ sind 2 Transformationen der Länge $N/2$ und erfordern zu ihrer Bestimmung daher insgesamt wiederum N Operationen auf den $G_k^{ee,eo,oe,oo}$. Insgesamt ist die Anzahl der nötigen rekursiven Anwendungen bis hinunter zur Einpunkttransformation durch den binären Logarithmus $\log_2 N$ von N gegeben. Damit benötigt eine FFT nur $N \log_2 N$ Operationen im Vergleich zu den N^2 Operationen der einfachen DFT. Diese enorme Zeitersparnis macht die Methode sehr wichtig bei praktischen Berechnungen in den unterschiedlichsten Gebieten wie z.B. in der Elektronik, Optik und in der Quantenmechanik.

In diesem Zusammenhang sollte auch bemerkt werden, daß die FFT *in place* durchgeführt werden kann, da auf jeder Ebene die Anzahl der Komponenten der verschiedenen benötigten Transformationen gerade wieder N beträgt.

Weitere Arbeit muß bei der FFT verrichtet werden, weil die Summanden der Teilreihen bei der sukzessiven Unterteilung in nicht trivialer Art und Weise kombiniert werden müssen.

betragsmäßig gleich der Zentrifugalkraft ist, haben wir

$$f = \frac{m_E v_0^2}{r} = \frac{G m_S m_E}{r^2} \Rightarrow G m_S = v_0^2 r = 4\pi^2 \frac{\mathrm{AE}^3}{\mathrm{a}^2} \tag{3.33}$$

Für die numerische Berechnung verwenden wir wieder die Newtonsche Bewegungsgleichung

$$\ddot{x} = \frac{f_x}{m_E} = -\frac{4\pi^2 x}{r^3} \text{ und } \ddot{y} = \frac{f_y}{m_E} = -\frac{4\pi^2 y}{r^3} \tag{3.34}$$

Wir benutzen in diesem Fall wieder die Verlet-Methode für unser Programm und lösen für die numerische Rechnung die beiden Gleichungen

$$x(t+\Delta t) = -x(t-\Delta t) + 2x(t) - 4\pi^2 x(t) r_3(t) \Delta t^2 \tag{3.35}$$
$$y(t+\Delta t) = -y(t-\Delta t) + 2y(t) - 4\pi^2 y(t) r_3(t) \Delta t^2 \tag{3.36}$$

Dabei ist die Abkürzung $r_3(t) = \left(\sqrt{x(t)^2 + y(t)^2}\right)^{-3}$ verwendet worden.

```
int main()
{
   double p0x, p1x, p2x, dp0x, dp1x, dp2x, t;
   double p0y, p1y, p2y, dp0y, dp1y, dp2y, r3;
   double alpha, t_max, dt;

// read parameters
   cout << "Anf.x  ";  cin >> p1x;
   cout << "Anf.y  ";  cin >> p1y;
   cout << "Anf.vx ";  cin >> dp1x;
   cout << "Anf.vy ";  cin >> dp1y;
   cout << "Alpha  ";  cin >> alpha;
   cout << "Gesamtzeit     ";  cin >> t_max;
   cout << "Zeitintervall  ";  cin >> dt;

   const double beta=(alpha)/2.;
   const double dt2=dt*dt;
   const double pi24=4.*M_PI*M_PI;

// setup initial conditions
   p0x=p1x-dt*dp1x;
   p0y=p1y-dt*dp1y;
   dp0x=dp1x;
   dp0y=dp1y;

// loop from t=0 to t=t_max
```

```
   t=0.0;
   ofstream outfile("orbit.dat");
   for( int i=0; i<t_max/dt; i++ ) {
        r3=1./pow(p1x*p1x+p1y*p1y,beta);
        p2x=2.*p1x-p0x-pi24*dt2*r3*p1x;  // Verlet algorithm
        p2y=2.*p1y-p0y-pi24*dt2*r3*p1y;
        outfile << t << ' ' << p1x << ' ' << p1y << '\n';
        t=t+dt;
        p0x=p1x;                         // change new->old
        p0y=p1y;
        p1x=p2x;
        p1y=p2y;
   }
   outfile.close();
}
```

Für ideale Anfangsbedingungen, das wäre $x_0 = 1$, $y_0 = 0$, $vx_0 = 0$ und $vy_0 = 2\pi$, erhält man für die Erdbewegung eine Kreisbahn, siehe Abbildung 3.14 (links). Verändert man die Anfangsgeschwindigkeit auf $vy_0 = 5$, so ergibt sich eine elliptische Bahn (rechts).

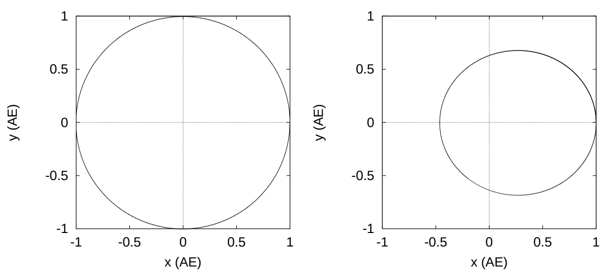


Abbildung 3.14: Bahnkurven der Erde um die Sonne für verschiedene Anfangsgeschwindigkeiten $vy_0 = 2\pi$ (links) und $vy_0 = 5$ (rechts).

Eine weitere Variationsmöglichkeit besteht in der Änderung des Gravitationsgesetzes. Anstatt die Gravitationskraft proportional zu $1/r^2$ abfallen zu lassen, kann man sie auch proportional zu $1/r^{\alpha-1}$ machen. Allgemein wird im obigen Programm `r=1/sqrt(pow(r, alpha))` verwendet, wobei `pow(r, alpha)` der Potenzierung r^α entspricht. Abbildung 3.15 zeigt vier verschiedene Beispiele mit Werten für β zwischen 3.00 und 2.01, wobei $\beta = \alpha - 1$. Die Rotation der Ellipse bei den Beispielen für $\beta \approx 2$ nennt man Präzesion.

Für $\beta = 3.00$ fällt auf, daß die Erde, nachdem sie in die Nähe der Sonne "geflogen" ist, sich mit hoher Geschwindigkeit aus dem System entfernt. Dies ist allerdings ein numerisches Problem. Für diesen Effekt ist die Wahl eines zu großen Δt verantwortlich, daß bei großen

als diskrete Fouriertransformation (DFT). Der die Frequenz kennzeichnende Index k läuft von $0, \ldots, \pm N/2$ entsprechend den Kreisfrequenzen $0, \pm\frac{2\pi}{T}, \pm\frac{2\pi}{T} \ldots, \pm\frac{2\pi}{2\Delta} = \pm\frac{2\pi}{f}_c$. Dies sind $N+1$ Koeffizienten im Gegensatz zu den N Abtastwerten, aber man sieht leicht, daß $G_{N/2} = G_{-N/2}$, so daß tatsächlich nur N unabhängige Zahlen resultieren.

Häufig ist es praktisch, die Indizes auch im Frequenzraum ab 0 laufen zu lassen, dann aber bei $N-1$ aufzuhören. Durch Einsetzen in die Definitionsgleichung kann man sich leicht überzeugen, daß für die negativen Frequenzen $G_{-k} = G_{N-k}$ ist.

Die inverse Transformation lautet

$$f(t_n) = f(n\Delta) = \frac{1}{N}\sum_{k=0}^{N-1} G_k e^{i\frac{2\pi}{N}kn} \tag{8.58}$$

Für eine mögliche Herleitung dieser Beziehung sei nur angedeutet, daß man zunächst die Darstellung des Kronecker-δ

$$\delta_{nn'} = \frac{1}{N}\sum_{k=0}^{N-1} e^{\frac{2\pi}{N}k(n-n')} \tag{8.59}$$

aus hintereinanderausgeführter Transformation und Rücktransformation einer Folge $f_n = \delta_n n'$ nachweisen kann. Damit folgt (8.58) durch Einsetzen der G_k aus (8.57) und Vertauschen der Summation über n' und k.

Beide Tranformationsrichtungen sind jedoch in dieser Form sehr rechenintensiv, denn jeder der N Datenpunkte erfordert N komplexe Multiplikationen und Additionen, also sind insgesamt für die Rechnung N^2 Operationen nötig. Zum gleichenResultat gelangt man, durch die Interpretation der DFT (8.57) als Matrixmultiplikation der Matrix $(e^{-i\frac{2\pi}{N}kn})_{k,n=0\ldots N-1}$ mit dem Spaltenvektor (f_n).

Im 2-dimensionalen wächst die Anzahl der Operationen wie N^4, in 3D wie N^6. Es ist daher klar, daß man ein schnelleres Verfahren braucht. Solch ein Verfahren existiert und wurde im Laufe der Geschichte der Numerik mehrfach erfunden. Es wurde bekannt durch die Arbeit von Coley and Tukey (IBM Yorktown Heights) um 1965, ist jedoch in einer eleganten Version bereits durch Danielson und Lanczos 1942 beschrieben worden.

8.4.2 Schnelle Fouriertransformation (FFT)

Bei der sogenannten schnellen Fourier-Transformation (engl. *Fast Fourier Transform* (FFT) geht man von der normalen DFT aus, zerlegt sie aber in eine Komponente mit geradem Laufindex m und in eine mit ungeradem m (s. unten). Man erhält so zwei Summen, die ihrerseits wieder DFT's eines reduzierten Problems sind. Wiederholte rekursive Anwendung dieser Technik führt auf die DFT einer einzelnen Zahl, die dann gar nicht mehr transformiert werden muß.

durch ihre Abtastwerte f_i. Eine explizite Darstellung für reelles $f(t)$ ist

$$f(t) = \Delta \sum_{n=-\infty}^{\infty} f_n \frac{\sin \omega_c (t - n\Delta)}{\pi (t - n\Delta)}. \tag{8.55}$$

Dem Quotienten muß für $t = m\Delta$ und $n = m$ mittels der d'Hopitalschen Regel als Grenzwert ein Sinn gegeben werden.

Störend ist natürlich, daß wir nicht vermeiden können, auch Signalanteile zu erfassen, deren Frequenz jenseits der Nyquist-Frequenz liegt, sofern sie denn im ursprünglichen Signal vorhanden sind. Wenn dies einmal passiert ist, dann gibt es keinen Weg zurück... In der Praxis muß man daher sicherstellen, daß vor dem Abtasten mit geeigneten Mitteln die Bandbreite des Signales auf die Nyquist Frequenz beschränkt wird.

8.4.1 Diskrete Fouriertransformation (DFT)

Wir haben oben gesehen, daß sich recht komplexe Rechenvorschriften im Zeitraum — wie beispielsweise die Berechnung von Faltungs- oder Korrelationsintegralen — auf recht einfache Vorschriften im Frequenzraum reduzieren — hier die punktweise Multiplikation der Transformierten. Nun ist die Bestimmung der Korrelationsfunktion im Zeitraum ein Algorithmus der Komplexität N^2, denn wir müssen für jedes mögliche τ (davon gibt es N) im wesentlichen N Zahlenpaare miteinander multiplizieren und aufsummieren. Falls es möglich ist, die Fouriertransformation mit einem Rechenaufwand von nicht wesentlich mehr als $\sim N$ Operationen durchzuführen, dann können wir tatsächlich vom geringeren Aufwand im Frequenzraum profitieren.

Wir wollen also zunächst eine diskrete Fouriertransformation einführen und dann deren Rechenaufwand abschätzen.

Wir betrachten die folgende Näherung der Fouriertransformation von $f(t)$

$$\begin{aligned} \sqrt{2\pi} g(\omega_k) &= \int_{-\infty}^{\infty} f_i e^{-i\omega_k t} dt \\ &\approx \sum_{n=0}^{N-1} \Delta f_k e^{-i\frac{2\pi}{N\Delta} kn\Delta} \\ &= \Delta \sum_{n=0}^{N-1} f_n e^{-i\frac{2\pi}{N} kn}. \end{aligned} \tag{8.56}$$

Wir bezeichnen $\frac{2\pi}{\Delta} g(\omega_k)$ als G_k und definieren damit die Rechenvorschrift (8.57)

$$G_k = \sum_{n=0}^{N-1} f_n e^{-i\frac{2\pi}{N} kn} \tag{8.57}$$

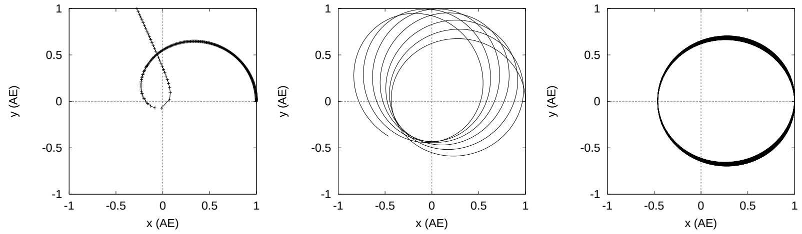


Abbildung 3.15: Bahnkurven bei versch. Gravitationsgesetzen $\beta = 3$ (links), $\beta = 2.1$ (mitte) und $\beta = 2.01$ (rechts).

Geschwindigkeitsänderungen kleiner sein sollte. Allerdings würde bei einem kleineren Δt die Simulation zu lange dauern. Man benötigt also ein automatisches Verfahren, das die Größe von Δt regelt. Eine Möglichkeit ist es, die Änderung der Energie für verschiedene Δt zu vergleichen und den Zeitschritt dementsprechend zu verändern (adaptives Verfahren). Dazu bietet sich z.B. das Runge-Kutta Verfahren an, da dort sowieso Werte zur Zeit $t + \Delta t/2$ berechnet werden. Die Energie der Erde ist

$$E_{Erde} = -\frac{G m_S m_E}{r} + \frac{m_E}{2} v^2 \tag{3.37}$$

Die Energie ergibt sich also aus den Koordinaten x und y für $r = \sqrt{x^2 + y^2}$ direkt und für $\mathbf{v} = \Delta\mathbf{r}/\Delta t$ indirekt. Wir berechnen zuerst die Energie E_1 aus

$$x_1 = x(t + \Delta t) = f_{\Delta t}(x(t), y(t)) \tag{3.38}$$
$$y_1 = y(t + \Delta t) = g_{\Delta t}(x(t), y(t)) \tag{3.39}$$

und die Energie E_2 mit halb so großem Δt durch

$$x_2 = f_{\frac{\Delta t}{2}}(x(t + \frac{\Delta t}{2}), y(t + \frac{\Delta t}{2})) \tag{3.40}$$
$$y_2 = g_{\frac{\Delta t}{2}}(x(t + \frac{\Delta t}{2}), y(t + \frac{\Delta t}{2})) \tag{3.41}$$

Die Funktionen f und g berechnen die neuen x- bzw. y-Werte für Δt oder $\Delta t/2$. Das entscheidende Kriterium ist die Differenz

$$\Delta E = |E_2 - E_1| \tag{3.42}$$

Ist ΔE groß, so ist Δt zu klein gewählt, ist ΔE sehr klein, so kann Δt größer gewählt werden. Im Programm könnte z.B. bei $\Delta E > 10^{-5} \to \Delta t' = \Delta t/2$ und bei $\Delta E < 10^{-7} \to \Delta t' = \Delta t * 2$ gewählt werden.

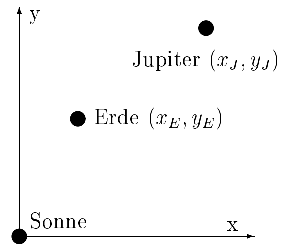


Abbildung 3.16: System Erde–Jupiter

3.3.1 Erde und Jupiter

Kommt zu den Körpern Sonne und Erde noch ein dritter (beispielsweise der Jupiter) hinzu, so ist dieses Problem nicht mehr analytisch lösbar. Die Keplerschen Gesetze zur Planetenbewegung gelten nicht mehr. Allgemein ist das beim N-Körperproblem ($N > 2$) so. Numerisch läßt sich dieses Problem immer noch lösen. Wir benötigen als erstes das Gravitationsgesetz für Erde-Jupiter:

$$f_{EJ} = -\frac{Gm_E m_J}{r_{EJ}^2} \text{ mit } r_{EJ} = \sqrt{(x_E - x_J)^2 + (y_E - y_J)^2} \tag{3.43}$$

Wieder stellen wir für die Erdbewegung (und auch für die Jupiterbewegung) eine komponentenweise Gleichung auf, die analog für y gilt.

$$\ddot{x}_E = -\frac{4\pi^2 x_E}{r_E^3} - \frac{Gm_J(x_E - x_J)}{r_{EJ}^3} \tag{3.44}$$

$$\ddot{x}_J = -\frac{4\pi^2 x_J}{r_E^3} - \frac{Gm_E(x_J - x_E)}{r_{EJ}^3} \tag{3.45}$$

Der rechte Ausdruck ist der Einfluß vom dritten Körper, also jeweils des Jupiters, bzw. der Erde. Für die Rechnung benötigen wir jetzt auch die Massen der einzelnen Körper.

$$\begin{aligned} m_E &= 6 * 10^{24}\text{kg} \\ m_S &= 2 * 10^{30}\text{kg} \\ m_J &= 2 * 10^{27}\text{kg} * M_J \end{aligned}$$

Den Faktor M_J benutzen wir später in der Simulation um auszuprobieren, wie sich unser System verhalten würde, wenn die Masse des Jupiters geändert würde. Der Radius der Jupiterbahn ist $r_J = 5.2\,\text{AE}$ und seine Bahngeschwindigkeit ist $v = 2.76\,\text{AE/a}$. Man kann nun die Konstanten

$$Gm_E = 12 * \pi^2 * 10^{-6} \tag{3.46}$$

$$Gm_J = 4 * \pi^2 * 10^{-3} * M_J \tag{3.47}$$

Um zu einer Aussage im Frequenzraum zu kommen, schreiben wir dieses jetzt unter Benutzung der Fourierdarstellungen von $f(t)$ und $\overline{f(t)}$ als dreifaches Integral

$$\int_{-\infty}^{\infty} dt\,|f(t)|^2 = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} d\omega' \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} d\omega \int_{-\infty}^{\infty} dt\,\mathcal{F}(f)(\omega')e^{i\omega' t}\mathcal{F}(\overline{f})(\omega)e^{i\omega t} \tag{8.51}$$

Unter Benutzung von $\mathcal{F}(\overline{f})(\omega) = \overline{\mathcal{F}(f)}(-\omega)$ machen wir im Integral über ω die Substitution $\omega \to -\omega$ und erhalten damit

$$\int_{-\infty}^{\infty} dt\,|f(t)|^2 = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} d\omega' \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} d\omega \int_{-\infty}^{\infty} dt\,|\mathcal{F}(f)(\omega')|^2 e^{i(\omega' - \omega)t} \tag{8.52}$$

Die Integration über t der komplexen Exponentialfunktion ist jetzt aber nicht anderes als $2\pi\delta(\omega - \omega')$ und damit finden wir nach Integration über ω' das *Parsevalsche Theorem*

$$\int_{\infty}^{\infty} dt\,|f(t)|^2 = \int_{\infty}^{\infty} d\omega\,|\mathcal{F}(f)(\omega)|^2. \tag{8.53}$$

Wenn wir $|\mathcal{F}(f)(\omega)|^2$ als Leistungsdichte im Frequenzraum benutzen, "verlieren" wir also keine Leistung durch die Fouriertransformation. Man betrachtet diese häufig nur für positve Frequenzen und definiert das *einseitige Leistungsspektrum* als

$$P(\omega) = |\mathcal{F}(f)(\omega)|^2 + |\mathcal{F}(f)(-\omega)|^2. \tag{8.54}$$

Für reelle Funktionen $f(t)$ ist $|\mathcal{F}(f)(\omega)|^2$ darüberhinaus symmetrisch, so daß $P(\omega) = 2|\mathcal{F}(f)(\omega)|^2$ gilt.

8.4 Abgetastete Signale

Von jetzt an wollen wir annehmen, daß die betrachteten Signale nur zu diskreten Zeitpunkten t_i, $i = 0 \ldots N-1$ vorliegen, die äquidistant mit Δ aufeinander folgen sollen. Wir werden N als durch 2 teilbar voraussetzen. Der Einfachheit halber bezeichne auch $f_i = f(t_i)$ den Wert des Signales zum Abtastzeitpunkt t_i. Die spezielle Frequenz (hier ist wirklich Frequenz und nicht Kreisfrequenz gemeint) $f_c = 1/2\Delta$ trägt den Namen *Nyquist* Frequenz. Ihre Bedeutung wird klar, wenn man sich überlegt wie die f_i einerseits eines harmonischen Signales von genau f_c und dem einer vielfachen Frequenz nf_c aussehen. Erhlt man beispielsweise bei der Frequenz f_c Abtastungen gerade an den Nullstellen des Signales, so ist das auch bei allen Vielfachen dieser Frequenz der Fall. Es ist also zwar möglich, von höherfrequenten Signalen Abtastungen zu sehen, sie lassen sich jedoch nicht von Signalen unterscheiden, deren Frequenzen im Bereich bis zu f_c liegen.

Jedoch gibt es das bemerkenswerte *Abtastungstheorem:* Eine kontinuierliche Funktion, deren Frequenzspektrum auf das Intervall $[-f_c, f_c]$ beschränkt ist, ist vollständig festgelegt

Ein System antworte auf einen δ-Impuls mit der *Impulsantwort* $h(t)$. Wegen der Zerlegung (8.46) von $f(t)$ gilt

$$F(t) = \int_{-\infty}^{\infty} dt'\, h(t-t')f(t') \tag{8.47}$$

für die Antwort auf $f(t)$. Aus dem Faltungssatz folgt jetzt die zentrale Aussage, daß die Fouriertransformierte der Antwort gleich dem Produkt der jeweiligen Transformierten von Impulsantwort und Eingangsfunktion ist:

$$\mathcal{F}(F)(\omega) = \mathcal{F}(h)(\omega)\mathcal{F}(f)(\omega) \tag{8.48}$$

Durch einen Vergleich mit (8.43) finden wir $\mathcal{F}(h)(\omega) = i\omega\mathcal{F}(\tilde{h})(\omega)$, im Zeitraum ist daher die Impulsantwort die Zeitableitung der Sprungantwort.

8.3.7 Fouriertransformation der δ-Funktion

Die Fouriertransformierte der δ-Funktion ist wegen (8.46) gleich $\mathcal{F}(\delta)(\omega) = 1/\sqrt{2\pi}$ also gleich einer Konstanten. Aus der Rücktransformation erhalten wir eine weitere Darstellung der δ-Funktion,

$$\delta(t) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} d\omega\, \frac{1}{\sqrt{2\pi}}e^{i\omega t} = \frac{1}{2\pi}\int_{-\infty}^{\infty} d\omega\, e^{i\omega t} \tag{8.49}$$

Multiplikation mit einer reellen Konstanten c, komplexe Konjugation und Umbenennung von t in ω und umgekehrt zeigt, daß wir damit im Prinzip eine Rechenregel für die Fouriertransformation einer Konstanten gefunden haben,

$$\sqrt{2\pi}c\delta(\omega) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} dt\, ce^{-i\omega t}. \tag{8.50}$$

Wie bereits früher erwähnt, muß solchen Ausdrücken natürlich mathematisch immer ein Sinn gegeben werden. Hier z.B. kann das dadurch geschehen, daß wir den Integranden auf das Interval $[-T/2\ldots T/2]$ einschränken, dann die Weiterrechnung ausführen (die ein Integral über die δ Darstellung beinhalten muß) und schließlich den Übergang $T \to \infty$ durchführen.

8.3.8 Spektrale Leistungsdichte

Häufig ist es von Wichtigkeit zu beurteilen, wieviel “Leistung” in einem bestimmten Bereich des Spektrums angesiedelt ist. Denken wir uns das Signal als abgegriffene Spannung, so ist die Leistung proportional zum Integral $\int_{-\infty}^{\infty} dt\, |f(t)|^2$.

berechnen. Als numerisches Verfahren verwenden wir wieder die Verlet-Methode. Die Gleichungen

$$\begin{aligned} x_E(t+\Delta t) &= -x_E(t-\Delta t) + 2x_E(t) - \\ &\quad -\Delta t^2 4\pi^2 \left(\frac{x_E(t)}{r_E^3(t)} + \frac{10^{-3}(x_E(t)-x_J(t)) * M_J}{r_{EJ}^3(t)}\right) \end{aligned} \tag{3.48}$$

$$\begin{aligned} x_J(t+\Delta t) &= -x_J(t-\Delta t) + 2x_J(t) - \\ &\quad -\Delta t^2 4\pi^2 \left(\frac{x_J(t)}{r_J^3(t)} + \frac{3 * 10^{-6}(x_J(t)-x_E(t))}{r_{EJ}^3(t)}\right) \end{aligned} \tag{3.49}$$

gelten für die x-Komponente, die y-Komponente ist analog dazu.

```
   double p0x, p1x, p2x, dp1x, t;
   double p0y, p1y, p2y, dp1y, r3e;
   double q0x, q1x, q2x, dq1x, r3j;
   double q0y, q1y, q2y, dq1y, r3ej;
   double alpha, massj, t_max, dt;
// ...
// loop from t=0 to t=t_max
   t=0.0;
   ofstream outfile("orbit.dat");
   for( int i=0; i<t_max/dt; i++ ) {
        r3e=1./pow(p1x*p1x+p1y*p1y,beta);
        r3j=1./pow(q1x*q1x+q1y*q1y,beta);
        r3ej=1./pow((p1x-q1x)*(p1x-q1x)+(p1y-q1y)*(p1y-q1y),beta);
// Verlet sums
        p2x=2.*p1x-p0x-pi24*dt2*
              (r3e*p1x+r3ej*1.e-3*massj*(p1x-q1x));  // Earth
        p2y=2.*p1y-p0y-pi24*dt2*
              (r3e*p1y+r3ej*1.e-3*massj*(p1y-q1y));
        q2x=2.*q1x-q0x-pi24*dt2*
              (r3j*q1x-r3ej*3.e-6*(p1x-q1x));        // Jupiter
        q2y=2.*q1y-q0y-pi24*dt2*
              (r3j*q1y-r3ej*3.e-6*(p1y-q1y));

        outfile << t << ' ' << p1x << ' ' << p1y <<
                         ' ' << q1x << ' ' << q1y << '\n';
        t=t+dt;
        p0x=p1x;                             // change new->old
        p0y=p1y;
        p1x=p2x;
        p1y=p2y;
        q0x=q1x;                             // change new->old
        q0y=q1y;
```

```
        q1x=q2x;
        q1y=q2y;
  }
```

Abbildung 3.17 zeigt die Bahnen für verschiedene Massen des Jupiters. Gerade in der letzten Grafik ist deutlich zu sehen, daß bei diesem Drei–Körper–Problem die Erde, die deutlich weniger Masse besitzt, als der Jupiter, einen chaotischen Verlauf hat. Ebenso ist das allgemeine N–Körper–Problem chaotisch!

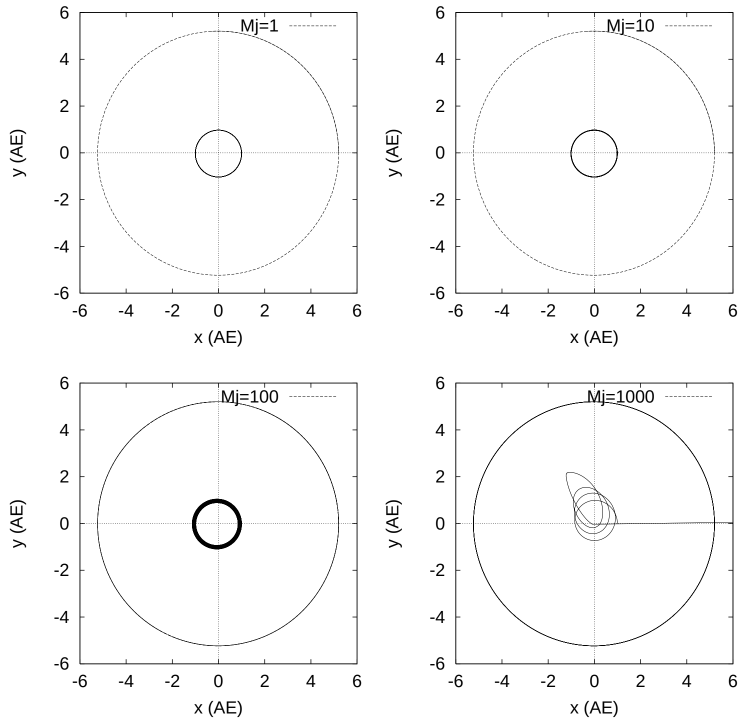


Abbildung 3.17: 2–Körper–Simulation mit verschiedenen Jupiter-Massen mit fixierter Sonne.

Wenn wir die Terme nach den $\theta(t-t_i)$ ordnen und die Summe von $-\infty$ bis ∞ erstrecken, dann wird

$$f(t) \approx \sum_{i=-\infty}^{\infty} \theta(t-t_i)\left(f(t_i) - f(t_{i-1})\right) \tag{8.40}$$

Durch Einfügen von Δt erhalten wir eine Summation, die für $\Delta t \to 0$ in ein Integral übergeht,

$$\begin{aligned} f(t) &\approx \sum_{i=-\infty}^{\infty} \Delta t\, \theta(t-t_i) \frac{f(t_i) - f(t_{i-1})}{\Delta t} \\ f(t) &= \int_{-\infty}^{\infty} dt'\, \theta(t-t')\, \frac{d}{dt'} f(t'). \end{aligned} \tag{8.41}$$

Aus dieser Darstellung erhalten wir die Systemantwort $F(t)$ auf $f(t)$ als Eingangssignal, indem wir unter Ausnutzung der Linearität die $\theta(t-t')$ durch $\tilde{h}(t-t')$ ersetzen:

$$F(t) = \int_{-\infty}^{\infty} dt'\, \tilde{h}(t-t') \frac{d}{dt'} f(t'). \tag{8.42}$$

Hier erkennt man unschwer eine Faltung wieder. Unter Ausnutzung des Faltungssatzes und des Verhaltens der Fouriertransformierten von Ableitungen findet man

$$\mathcal{F}(F)(\omega) = i\omega \mathcal{F}(\tilde{h})(\omega) \mathcal{F}(f)(\omega). \tag{8.43}$$

Noch einfacher wird diese Beziehung, wenn wir im Zeitraum die auf $f(t')$ wirkende Ableitung in (8.41) zunächst *rein formal* durch partielle Integration auf die θ Funktion überwälzen. Hierbei ist zu beachten, daß t' mit negativem Vorzeichen in $\theta(t-t')$ auftaucht, also

$$f(t) = \int_{-\infty}^{\infty} dt'\, \frac{d}{dt'} \theta(t-t') f(t'). \tag{8.44}$$

Diese formale Ableitung der θ Funktion hat also die Eigenschaft, bei Integration gerade den Wert des Koeffizienten an der Sprungstelle t' zu liefern. Diese Eigenschaft ist charakteristisch für die Dirac δ-“Funktion.” Somit haben wir eine *Darstellung (von vielen) der δ-Funktion* gefunden,

$$\delta(t) = \theta'(t), \quad \text{also} \tag{8.45}$$

$$f(t) = \int_{-\infty}^{\infty} dt'\, f(t')\delta(t-t'). \tag{8.46}$$

Mathematisch müssen wir implizit immer einen Rückbezug auf die Integraldarstellung (8.41) machen, um Ausdrücke auszuwerten, die δ Funktionen enthalten. In vielen Fällen reicht aber die Rechenregel (8.46) aus.

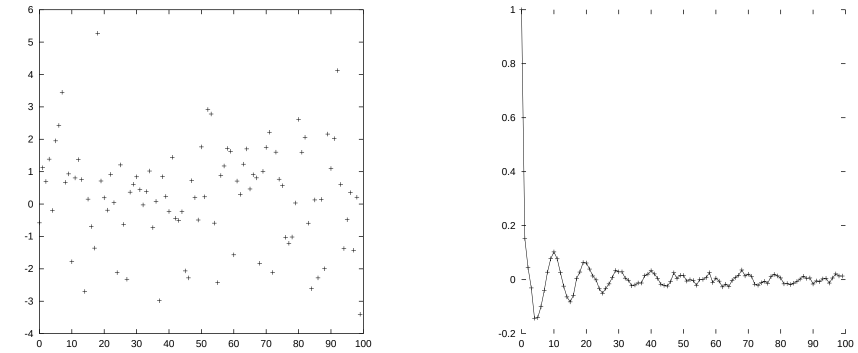


Abbildung 8.4: Links: Die ersten 100 Werte eines verrauschten Signales $s(t) + \eta(t)$. Rechts: Autokorrelationsfunktion dieses Signales über 10000 Abtastpunkte. Hier sind sowohl klare Rauschunterdrückungseffekte zu sehen als auch die vom Rauschen herrührende Überhöhung bei $\tau = 0$. Ebenso sieht man, daß das Ausgangssignal keine "langreichweitige" Korrelation hat, sondern seinen Anfangszustand nach etwa $t_c \approx 60 \ldots 80$ Abstastintervallen vergessen hat.

8.3.6 Faltungssatz und lineare Systemtheorie

Der Faltungsproduktes ist zentral für die Theorie linearer, zeitinvarianter Systeme. Diese sind dadurch charakterisiert, daß sie auf ein gegebenes Eingangssignal immer (Zeitinvarianz) mit einem bestimmten Signal am Ausgang "antworten." Summe und Vielfache von Eingangssignalen führen auf entsprechende Summen und Vielfache der Ausgangssignale (Linearität).

Relativ einfach zu messen ist die sogenannte Sprungantwort $\tilde{h}(t)$ des Systems auf eine $\theta(t)$ Funktion am Eingang,

$$\theta(t) = \begin{cases} 0, & t < 0, \\ 1, & t \geq 0, \end{cases} \tag{8.37}$$

sozusagen der "Einschaltknacks" des Systems. Um jetzt die Antwort des Systems auf beliebige Eingangssignale $f(t)$ zu bekommen, müssen wir $f(t)$ in eine Summe von θ-Funktionen zerlegen. Wir wollen dies durch einen Grenzübergang einer abgetasteten Version von $f(t_i)$ für $\Delta t = t_{i+1} - t_i \to 0$ erreichen. Im Zeitintervall $t_{i-1} < t \leq t_{i+1}$ etwa ist $f(t)$ die Summe zweier Rechteckimpulse, die wir aus θ-Funktionen zusammensetzen können,

$$\begin{aligned} f(t) \approx \theta(t - t_{i-1})f(t_{i-1}) &- \theta(t - t_i)f(t_{i-1}) & (8.38) \\ &+ \theta(t - t_i)f(t_i) - \theta(t - t_{i+1})f(t_i) & (8.39) \end{aligned}$$

3.3.2 Das Drei–Körper–Problem

Bei einer Masse des Jupiter von $M_j = 1000$ ist allerdings zu bedenken, das die Annahme einer unbeweglichen Sonne falsch wird. Simuliert man das wirkliche 3–Körper–Problem, mit beweglicher Sonne, so umkreisen Sonne und Jupiter den gemeinsamen Schwerpunkt und die Erde wird nach kurzer Zeit aus dem System geschleudert. In Abb. 3.18 ist die Simulation für verschiedene Zeitschritte dargestellt. Eigentlich entspricht diese Simulation einem Doppelsternsystem mit zwei großen Sonnen und einem Planeten. Schon bei einem Zeitschritt von $\Delta t \leq 0.01$ ist die Lösung beinahe nicht mehr von der Lösung für $\Delta t \leq 0.001$ zu unterscheiden. Als Kontrolle kann man die Energie im Planetensystem berechnen. Die kinetische Energie im N–Körper–System ist

$$E_k = \sum_{i=1}^{N} \frac{1}{2} m_i v_i^2 \ , \tag{3.50}$$

und die potentielle Energie errechnet sich aus allen paarweisen Wechselwirkungen zu

$$E_p = \sum_{i=1}^{N} \sum_{j=i+1}^{N} 4\pi^2 m_i m_j \frac{1}{r_{ij}} \ , \tag{3.51}$$

wobei wie oben alle Längen in AE, alle Geschwindigkeiten in v_0, alle Zeiten in Jahren, und alle Massen in Sonnenmassen m_S gemessen werden. In Abb. 3.18 erkennt man, daß die Energieerhaltung nur für große Zeitschritte verletzt wird.

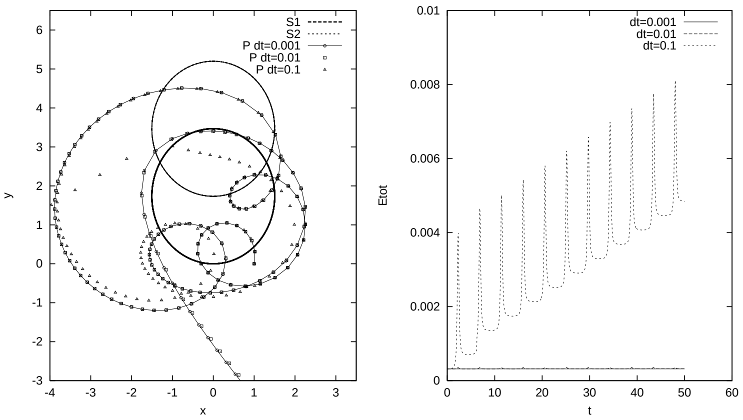


Abbildung 3.18: 3–Körper–Simulation mit Jupiter-Masse $M_j = 1000$ und beweglicher Sonne. Links: Trajektorien und rechts: Energie als Funktion der Zeit.

Die gleiche Methode, bei der jeder Körper mit jedem anderen wechselwirkt, verwendet man auch bei Simulation von Molekülen (Molekulardynamik), siehe Kapitel 3.4.

3.3.3 Perihelbewegung des Merkurs

Ein Beispiel für die Anwendung der Relaticitätstheorie in der Himmelsmechanik ist die Präzession des Perihels (der Umlaufbahn) des Merkurs, die aufgrund seiner Sonnennähe relativ groß ist. Die Bahn des Merkurs dreht sich um 566 Arksek/Jahrhundert (1 Arksek $= (1/3600)^\circ$). Für eine volle Umdrehung benötigt die Bahn insgesamt 230000 Jahre! Wird die Präzession allerdings selbst mit dem Einfluß des Jupiters errechnet, kommt man nur auf 523 Arksek/Jahrhundert! Albert Einstein erhält durch seine Allgemeine Relativitätstheorie (Raum wird durch Masse gekrümmt) ein erweitertes Bewegungsgesetz. Für die Kraft zwischen Sonne und Merkur gilt nun:

$$f \approx \frac{G m_S m_M}{r_{SM}^2}\left(1 + \frac{\alpha}{r_{SM}^2}\right), \text{ mit } \alpha \approx 1.1 * 10^{-8} \mathrm{AE}^2 \tag{3.52}$$

Der Unterschied in der Berechnung durch diese Formel (und mit dieser Konstanten α) ist 43 Arksek/Jahrhundert, genau die Differenz zwischen dem vorher berechneten Wert und dem beobachteten. Dieses Ergebnis bescherte der Allgemeinen Relativitätstheorie den Durchbruch, da sie ein bis dahin unlösbares Problem der Physik korrekt erklärt.

3.3.4 Starre Körper

Man kann auch die Drehung der Planeten um sich selbst als starren Körper berechnen. Zur Vereinfachung betrachten wir wieder nur eine Drehung im 2-Dimensionalen. Statt Ort $\mathbf{r}$ und Geschwindigkeit $\mathbf{v}$ wird nun auch der Drehwinkel φ und die Winkelgeschwindigkeit ω verwendet. Das Bewegungsgesetz lautet:

$$\ddot{\varphi} = \dot{\omega} = \frac{M}{J} \text{ , mit} \tag{3.53}$$

$$J = \sum_i m_i (\mathbf{r}_i - \mathbf{r}_M)^2 \text{ (Trägheitsmoment)} \tag{3.54}$$

$$M = \sum_i (\mathbf{r}_i - \mathbf{r}_M) \times \mathbf{f}_i \text{ (Drehmoment)} \tag{3.55}$$

m_i sind die Massenpunkte, aus denen sich der Körper zusammensetzt, $\mathbf{r}_i$ ihre jeweiligen Koordinaten, f_i ihre anliegenden Kräfte und $\mathbf{r}_M$ die Schwerpunktskoordinate. Ein Beispiel für diese Drehung ist die Bewegung des Saturnmondes Hyperion, dessen äußere Form der eines Eis gleicht. Man stellt fest, daß seine Bewegung chaotisch ist.

3.4 Molekulardynamik

Mit der sog. Molekulardynamik kann man nicht nur Moleküle wie z.B. das Helimatom simulieren, sondern ganz allgemeine N–Teilchen–Systeme wie Gase, Flüssigkeiten, Membranen oder Schüttgüter.

Beiträge wächst. Im Vergleich dazu wächst das Autokorrelationsintegral des Signales direkt proportional zur Anzahl der Beiträge. Relativ und für genügend lang andauernde Signale gesehen, kann daher die Kreuzkorrelation $\mathcal{C}(s,\eta)(\tau)$ gegen die Autokorrelation $\mathcal{C}(s,s)(\tau)$ vernachlässigt werden.

Gleichermaßen werden wir annehmen, daß die Autokorrelation des Rauschsignals verschwindet, außer an der Stelle $\tau = 0$, wo ein Wert proportional zum Quadrat der Amplitude des Rauschens angenommen wird. Das nötige Argument ist die Unabhängigkeit des Rauschsignals zu zwei verschiedenen Zeitpunkten, die erlaubt, das gleiche Argument wie im vorigen Abschnitt anzuwenden und die Autokorrelation für $\tau > 0$ gegen die bei $\tau = 0$ zu vernachlässigen, bei der alle Beiträge das gleiche positive Vorzeichen haben.

Bilden wir jetzt die Autokorrelation des Signales, so ergibt sich (Einsetzen in die Definition, Unterdrücken des Argumentes τ)

$$\begin{aligned} \mathcal{C}(s+\eta, s+\eta) &= \mathcal{C}(s,s) + \mathcal{C}(s,\eta) + \mathcal{C}(\eta,s) + \mathcal{C}(\eta,\eta) \\ &= \mathcal{C}(s,s) + \mathcal{C}(\eta,\eta). \end{aligned} \tag{8.34}$$

Das bedeutet, daß man außer bei $\tau = 0$, wo der zweite Term einen Beitrag liefert, die Autokorrelationsfunktion des ursprünglichen, unverrauschten Signales gewinnt.

Als ein Anwendungsbeispiel für das oben Gesagte zeigen wir in Abb. 8.4 die ersten 100 Werte einer Zeitreihe $f(t)$, die durch Abtasten gewonnen ist und deren Autokorrelation für $\tau \leq 100$, ermittelt über ein Zeitfenster von 10000 Abtastungen.

8.3.5 Faltung

In sehr enger Beziehung zur Definition der Kreuzkorrelation steht die Faltung, die sich nur im Vorzeichen des zweiten Integranden von der Korrelation unterscheidet,

$$f \odot g(\tau) = \int_{\infty}^{\infty} dt\, f(t) g(\tau - t). \tag{8.35}$$

Mit einer zu (8.32) exakt parallelen Begündung läßt sich zeigen, daß die Fouriertransformierte des Faltungsproduktes zweier Funktionen bis auf einen konstanten Vorfaktor gleich dem Produkt der Fouriertransformierten ist

$$\mathcal{F}(f \odot g)(\omega) = \sqrt{2\pi} \mathcal{F}(f)(\omega) \mathcal{F}(g)(\omega) \tag{8.36}$$

Im Gegensatz zur Beziehung für die Kreuzkorrelation taucht jetzt hier im Argument von $\mathcal{F}(g)$ kein negatives Vorzeichen mehr auf. Die Beziehung (8.36) trägt die Bezeichnung *Faltungssatz.*

2. *Varianz:* Der Funktionswert $\mathcal{C}(f,f)(0) = \int f^2(t)dt$ ist ein Maß für die Variabilität des Signals, in Analogie zur Varianz $\langle\zeta^2\rangle - \langle\zeta\rangle^2$ einer Zufallsvariablen ζ. Da für quadratintegrable Funktionen strenggenommen allerdings immer der Mittelwert $\lim_{T\to\infty}\int_{-T/2}^{T/2} dt\, f(t)$ verschwindet, bleibt nur der der Mittelwert des Quadrates $\langle\zeta^2\rangle$ übrig.
3. *Unkorreliertheit bei langen Zeiten:* Reale Signale verlieren für große Zeitzwischenräume τ ihre Kohärenz, das Signal "vergißt" sozusagen seine Vergangenheit. Außer im Falle von Modellsignalen wie beispielsweise $f(t) = \sin t$ strebt daher $\lim_{\tau\to\infty}\mathcal{C}(f,f)(\tau)$ gegen 0.

Korrelationszeit eines Signales

Häufig kann man eine Autokorrelationszeit τ_c definieren, die die Zeitkonstante des "Gedächtnisses" des Signals charakterisiert. Je nach Verlauf des Signales $f(t)$ kann sich dazu eignen

1. der Wert von τ, zu dem $\mathcal{C}(f,f)(\tau)$ auf einen Bruchteil $1/e$ des Wertes $\mathcal{C}(f,f)(0)$ abgefallen ist.
2. unter der Annahme, daß die Abnahme von $\mathcal{C}(f,f)(\tau)$ effektiv exponentiell erfolgt $= \mathcal{C}(f,f)(0)e^{-\tau/\tau_c}$ kann man das Integral $\int \mathcal{C}(f,f)(\tau)\,d\tau$ einmal aus dieser einfachen Beziehung zu $\tau_c\mathcal{C}(f,f)(0)$ bestimmen, andererseits aber auch (in der Regel numerisch) aus dem wirlichen Verlauf von $\mathcal{C}(f,f)(\tau)$. Der Vergleich liefert τ_c.
3. Für Funktionen mit "Durchschwinger" kann man ein τ_c manchmal schon durch Ermitteln der ersten Nullstelle von $\mathcal{C}(f,f)(\tau)$ gewinnen.

8.3.4 Autokorrelation zur Rauschunterdrückung

Stellen wir uns ein Signal vor, das aus einem Nutzanteil $s(t)$ und "Rauschen" $\eta(t)$ zusammengesetzt ist. Unabhängig von der physikalischen Ursache dieses Rauschens wollen wir es durch seine statistischen Eigenschaften definieren.

Statistisch gesehen verschwindet die Kreuzkorrelation von Signal und Rauschen, wenn es sich um zwei unabhängige Prozesse handelt. Diese Aussage setzt voraus, daß die Beobachtungszeit und die Dauer des Nutzsignales lang genug sind. Anschaulich benötigt man ein "Herausmitteln" der Rauschanteile zu verschiedenen Zeitpunkten. Das heißt, daß im Korrelationsintegral die Produkte $s(t)\eta(t-\tau)$ wegen des ständig wechselnden Vorzeichens von $\eta(t)$ als stochastische Beiträge auftreten. Wenn wir uns der Einfachheit halber das Kreuzkorrelationsintegral als *Summe* über einzelne Beiträge vorstellen, dann können wir unser Wissen aus dem Kapitel über stochastische Integration anwenden, daß nämlich eine Summe über solche Beiträge wie die Quadratwurzel aus der Anzahl der

Ein weiteres großes Gebiet der numerischen Physik ist die Molekulardynamik: eine Menge von Teilchen wird für eine bestimmte Zeit mit allen auf sie wirkenden Kräften simuliert. Dies ist ein typisches N–Körper-Problem. Dabei kann eine riesige Anzahl an Gleichungen auftreten, die alle berechnet werden müssen. Im November 1997 wurde durch den Zusammenschluß von zwei Supercomputern eine Molekularsimulation mit über einer Milliarde (10^9) Punkte eines Kristallgitters durchgeführt. Allerdings besitzt ein Kubikzentimeter Materie ungefähr 10^{22} Moleküle, also einen Faktor 10^{13} mehr. Man könnte also nur $1/10^{13}\,\mathrm{cm}^3$ simulieren.

In der Praxis ist es sowieso nur üblich bis zu 10^6 Teilchen zu simulieren. Weiterhin muß man wegen der großen auftretenen Kräfte die Zeitintervalle Δt sehr klein wählen, üblicherweise in der Größenordnung von 10^{-12}s= 1ps. Bei normalerweise 10^6 Iterationen kommt man dann auf eine Zeit von 1μs. Wegen dieser Einschränkungen werden nur kleine und kurze Vorgänge auf diese Weise simuliert wie Bruchspitzen, Schmelzen und Erstarren von Materie oder "head-crash", dem Aufprall des Lese-/Schreibkopfes auf die Magnetscheibe einer Festplatte.

3.4.1 Das Heliumatom

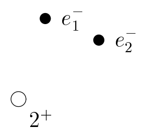


Abbildung 3.19: Elektronenbewegung im Heliumatom

Die elektrostatischen Kräfte ähneln den Gravitationskräften. So kann man auch die Bahn der beiden Elektronen in einem Heliumatom konstruieren. Neben der Anziehung von entgegengesetzten Ladungen kommt noch die Abstoßung durch gleiche Ladungen hinzu, die mit umgekehrtem Vorzeichen in die Berechnung eingehen. Wir setzen Masse und Ladung des Elektrons auf 1 ($m_e = q_e = 1$) und erhalten die beiden Kräfte für die Elektronen :

$$f_1 = -2\frac{\mathbf{r}_1}{r_1^3} + \frac{\mathbf{r}_1 - \mathbf{r}_2}{r_{12}^3} \tag{3.56}$$

$$f_2 = -2\frac{\mathbf{r}_2}{r_2^3} + \frac{\mathbf{r}_2 - \mathbf{r}_1}{r_{12}^3} \tag{3.57}$$

$$\tag{3.58}$$

Durch die möglichen starken Kräfte ist hierbei ein adaptiver Zeitschritt notwendig. Die entstehende Bewegung ist wieder chaotisch.

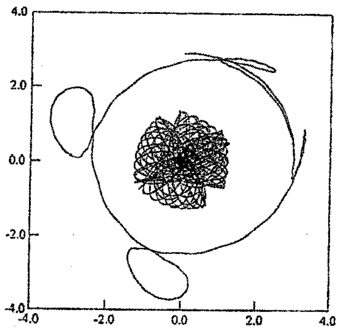


Abbildung 3.20: Bewegung der zwei Elektronen im Heliumatom.

3.4.2 Optimierung

Durch die gegenseitige Wechselwirkung ist der Aufwand bei N Teilchen $O(N^2)$, da der Einfluß von allen Teilchen auf jedes Teilchen berücksichtigt wird. Nun ist es aber so, daß die Stärke der Kraft von der Entfernung der Teilchen abhängt. Nur die Teilchen in näherer Umgebung haben einen wirklichen Einfluß auf die Bewegung. Um das auszunutzen gibt es verschiedene Ansätze:

1. Nachbarschaftstafeln
   Für jedes Teilchen i gibt es eine Liste von Teilchen die sich innerhalb einer Kugel (oder eines Kreises) mit dem Radius R um Teilchen i befinden. R muß dabei größer als die Reichweite der Kräfte sein. Für das Teilchen i werden nur die Kräfte aus den in der Liste stehenden Teilchen berechnet und die Liste wird regelmäßig erneuert. Der Erneuerungsprozeß kann zwar $O(N^2)$ sein, aber der Aufwand muß nur nach vielen Zeitschritten betrieben werden.
2. linked-cell (verbundene Zellen)
   Der Raum (die Ebene) wird in einzelne Zellen unterteilt. Jede Zelle besitzt eine Liste der Teilchen in ihr und jedes Teilchen in einer Zelle wechselwirkt nur mit Teilchen in umgebenden Zellen. Für die Berechnung werden nur die Teilchen aus der eigenen und den benachbarten Zellen herangezogen.

Durch Kombination dieser beiden Methoden kann man einen Aufwand von $O(N)$ erreichen. Außerdem lassen sich die Methoden sehr einfach parallelisieren und so auch auf Systemen mit mehreren Recheneinheiten effektiv realisieren.

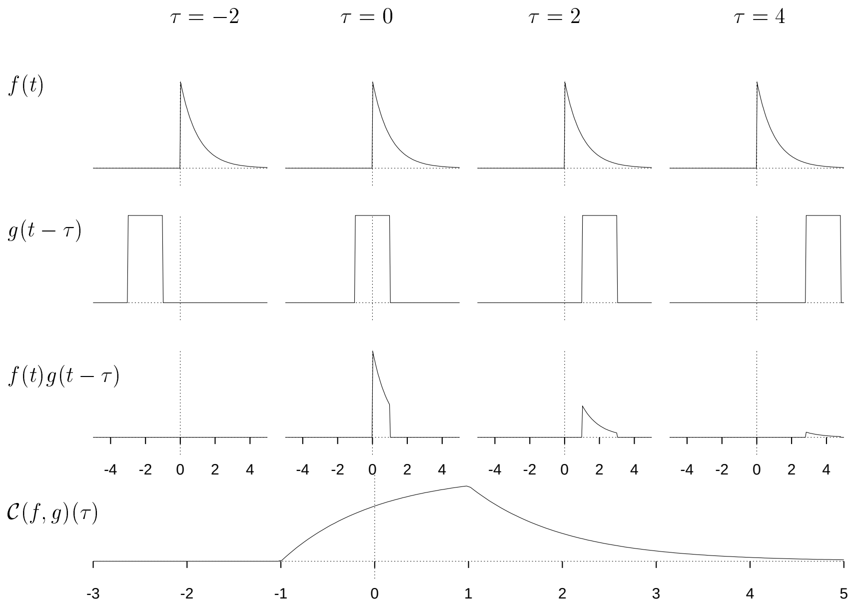


Abbildung 8.3: Die Kreuzkorrelation $\mathcal{C}(f,g)(\tau)$ zweier Funktionen läßt sich durch den Überlapp einer der Funktionen mit einer verschobenen Version der anderen verdeutlichen. Oben sind für verschiedene Werte des Parameters τ in der ersten Zeile die Funktionen $f(t) = \theta(t)e^{-t}$ und in der zweiten $g(t-\tau)$, $g(t) = \theta(t+1) - \theta(t-1)$ aufgetragen. Die dritte Zeile enthält das Produkt dieser beiden, dessen Integral schließlich den Wert der Kreuzkorrelation für diesen einen speziellen Wert von τ festlegt. Der Verlauf der Kreuzkorrelationsfunktion für alle τ, die sich in diesem Falle zu $\exp(x+1)\theta(-x-1) - \theta(-x-1) - \exp(x-1)\theta(-x+1) + \theta(-x+1)$ bestimmt, ist als Funktionsverlauf ganz unten angegeben.

Hierzu könnte er den Mittelwert des Produktes $\langle n_6 B \rangle$ auswerten und mit dem Produkt der Mittelwerte $\langle n_6 \rangle \langle B \rangle$ vergleichen. Sind diese beiden Größen gleich, so nennt man h_6 und B statistisch unabhängig/unkorreliert.

Für den *Zeitverlauf* zweier Größen $f(t)$ und $g(t)$ können wir fragen, ob die Werte von f zur Zeit t und g zu einem anderen, früheren Zeitpunkt $t-\tau$ miteinander korreliert sind. Wir ersetzen die Mittelwertbildung durch ein Integral und definieren die Kreuzkorrelationsfunktion durch

$$\mathcal{C}(f,g)(\tau) = \int_{-\infty}^{\infty} f(t)\, g(t-\tau)\, dt \tag{8.30}$$

$$= \int_{-\infty}^{\infty} f(t+\tau)\, g(t)\, dt. \tag{8.31}$$

Beide Definitionen sind äquivalent. Die Kreuzkorrelationsfunktion besitzt als Argument natürlich die gegenseitige Verschiebung τ der beiden Funktionen. Die Kreuzkorrelation zweier Funktionen kann man sich anschaulich als den Überlapp beider Funktionen vorstellen, wie das in der Abb. 8.3 verdeutlicht ist.

Die Fouriertransformierte der Korrelationsfunktion berechnen wir so:

$$\begin{aligned}
\mathcal{F}(\mathcal{C}(f,g))(\omega) &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} d\tau\, e^{-i\omega\tau} \int_{-\infty}^{\infty} dt\, f(t+\tau)\, g(t) \\
&= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} dt' \int_{-\infty}^{\infty} dt\, f(t')\, g(t) e^{-i\omega(t'-t)} \\
&= \sqrt{2\pi} \left(\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} dt' f(t') e^{-i\omega t'} \right) \left(\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} dt\, g(t) e^{i\omega t} \right) \\
&= \sqrt{2\pi} \mathcal{F}(f)(\omega) \mathcal{F}(g)(-\omega).
\end{aligned} \tag{8.32}$$

Sie ergibt sich also als das Produkt der Transformierten von f und g an den Stellen ω und $-\omega$. Für reelle Funktionen $g(t)$ gilt $\mathcal{F}(g)(-\omega) = \overline{\mathcal{F}(g)}(\omega)$, so daß $\mathcal{F}(\mathcal{C}(f,g))(\omega) = \sqrt{2\pi}\mathcal{F}(f)(\omega)\overline{\mathcal{F}(g)}(\omega)$.

Aufgrund der obigen Definition kann man auch eine *Autokorrelationsfunktion* $\mathcal{C}(f,f)$ einer Funktion $f(t)$ mit sich selbst einführen. Falls $f(t)$ reell ist, gilt aufgrund von (8.32)

$$\mathcal{F}(\mathcal{C}(f,f))(\omega) = \sqrt{2\pi} |\mathcal{F}(f)(\omega)|^2. \tag{8.33}$$

Diese Beziehung ist auch unter der Bezeichnung *Wiener-Khinchin-Theorem* bekannt.

Die Autokorrelationsfunktion hat folgende Eigenschaften, die aus ihrer Definition ersichtlich sind,

1. *Symmetrie:* $\mathcal{C}(f,f)(\tau) = \mathcal{C}(f,f)(-\tau)$, daher werden Autokorrelationsfunktionen nur für positive τ angegeben,

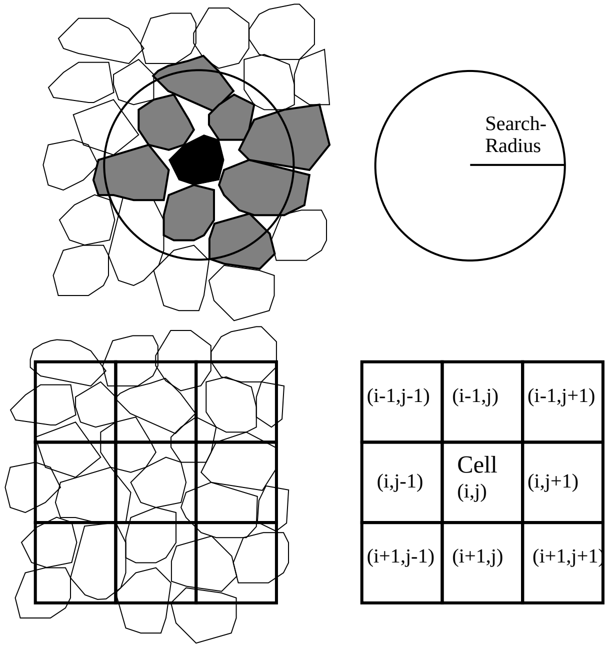


Abbildung 3.21: Verlet–Nachbarschaftstafeln (oben) und linked–cell (unten)

3.4.3 Weitere Vielteilchensysteme

Sind die Teilchen größer, wie Beispielsweise bei Sand, wo in einem Kubikzentimeter ungefähr 10^5 Körner sind, und durch kleinere auftretende Kräfte die Zeitintervalle größer, also $\Delta t \approx 10^{-5}s$, sein können, sind realistische Mengen und Zeitdauern von mehreren Minuten möglich.

Hierzu gehört beispielsweise die Simulation von Sand in einem Trichter, Fragmentation einer Wand mit dem Hammer, Teilchenvermischung in Flüssigkeiten oder die Zersplitterung zweier Kugeln nach einem Zusammenstoß.

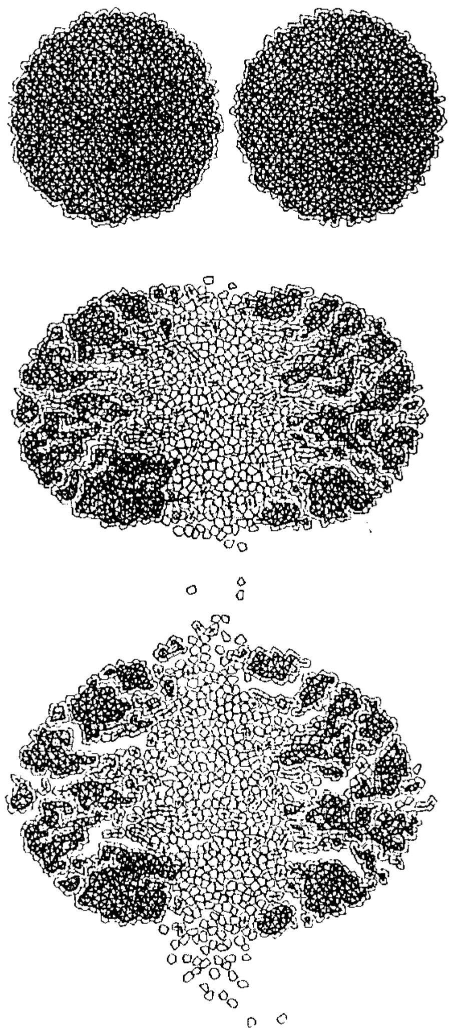


Abbildung 3.22: Zusammenstoß zweier Atomkerne

Je lokalisierter also die Funktion $f(t)$ ist, desto breiter ist $g(\omega)$, also das "Spektrum der Frequenzen," das man zu ihrer Darstellung braucht. In der Quantenmechanik wird sich dies in der *Unschärferelation* ausdrücken.

3. *Verschiebung im Zeitbereich* (Variablensubstitution)

$$\mathcal{F}(f(t-t_0))(\omega) = e^{-i\omega t_0}\mathcal{F}(f(t))(\omega) \tag{8.26}$$

4. *Modulationssatz* (Variablensubstitution)

$$\mathcal{F}(e^{i\omega_0 t}f(t))(\omega) = \mathcal{F}(f(t))(\omega-\omega_0) \tag{8.27}$$

Niederfrequente Nutzsignale $f(t)$ werden häufig hochfrequenten Trägersignalen $e^{i\omega_0 t}$ aufmoduliert. Das Nutzsignal wird dadurch in den hochfrequenten Bereich verschoben.

5. *Symmetrien*

		$f(t) = f(-t)$	$f(t) = -f(-t)$
$f(t)$ komplex		$\mathcal{F}(f)(\omega) = \mathcal{F}(f)(-\omega)$	$\mathcal{F}(f)(\omega) = -\mathcal{F}(f)(-\omega)$
$f(t)$ reell	$\mathcal{F}(f)(\omega) = \overline{\mathcal{F}(f)}(-\omega)$	$\mathcal{F}(f)(\omega)$ reell, gerade	$\mathcal{F}(f)(\omega)$ imaginär, ungerade

6. *Zeitumkehr*

$$\mathcal{F}(f(-t)) = (\mathcal{F}f)(-\omega) \tag{8.28}$$

7. *Differentiation im Zeitbereich*

$$\mathcal{F}(f'(t)) = i\omega(\mathcal{F}(f))(\omega) \tag{8.29}$$

Diese Beziehung spielt eine herausragend wichtige Rolle bei der Lösung partieller und gewöhnlicher Differentialgleichungen weil die Differentiation in eine einfache Multiplikation übergeht. Im Frequenzraum lassen sich so Lösungen sehr häufig zumindest formal finden, die dann nach Rücktransformation (falls möglich analytisch, sonst numerisch) die Lösung der Differentialgleichung bilden.

8.3.3 Kreuz- und Autokorrelationsfunktion

Bei der Analyse der Ergebnisse einer Messung tritt häufig die Frage auf, ob zwei Größen in Abhängigkeit voneinander stehen. Statistisch gesehen, möchten wir wissen, ob die Größen voneinander unabhängig sind. Ein Spielkasinobesitzer gewordener Physiker könnte sich beispielsweise fragen: Ist die relative Häufigkeit des Auftretens von Sechsen n_6 beim Werfen eines inhomogenen Metallwürfels korreliert mit der Stärke eines außen anliegenden Magnetfeldes B?

Setzt man in diesen Beziehungen $\omega = n\Delta\omega = n\frac{2\pi}{T}$, mit $\Delta\omega = \frac{2\pi}{T}$, so kann man zunächst die Bestimmung der Koeffizienten umschreiben zu

$$g(\omega) = c_n \frac{T}{\sqrt{2\pi}} = \frac{1}{\sqrt{2\pi}} \int_{-T/2}^{T/2} f(t)\, e^{-i\omega t}\, dt, \tag{8.19}$$

$$c_n = \frac{\sqrt{2\pi}}{T} g(\omega) = \frac{2\pi}{T} \frac{1}{\sqrt{2\pi}} g(\omega) = \Delta\omega \frac{1}{\sqrt{2\pi}} g(\omega). \tag{8.20}$$

Damit erhält man für die Entwicklung von $f(t)$

$$f(t) = \frac{1}{\sqrt{2\pi}} \sum_{n=-\infty}^{\infty} \Delta\omega\, g(\omega)\, e^{i\omega t} \tag{8.21}$$

Führt man jetzt den Grenzübergang $T \to \infty$ aus, dann verwandelt sich die Summe in ein Integral und der Wert $\Delta\omega$ geht über in ein Differential,

$$\mathcal{F}^{-1}(g)(t) = f(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(\omega)\, e^{i\omega t}\, d\omega \tag{8.22}$$

$$\mathcal{F}(f)(\omega) = g(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t)\, e^{-i\omega t}\, dt \tag{8.23}$$

Die erste dieser Beziehungen heißt Fourier-Rücktransformation, die zweite definiert die Fouriertransformierte der Funktion $f(t)$.

Damit die obigen Integrale wohldefiniert sind, fordert man in der Regel, daß das Signal $f(t)$ quadratintegrabel ist, d.h. $\int dt\, |f(t)|^2 < \infty$. Dies ist insbesondere für reale, zeitlich begrenzte Signale immer erfüllt. Für Modellsignale wie die im vorigen Abschnitt besprochenen periodischen Funktionen ist das jedoch nicht gegeben. Eine Möglichkeit, die obigen Beziehungen zu retten, besteht darin, die Signale einfach bei einem großen T abzuschneiden. Dann führt man alle Rechnungen wie gewohnt durch, mit einer expliziten T-Abhängigkeit während der gesamten Rechnung. Am Schluß führt man dann den Übergang $T \to \infty$ durch.

Wir werden auf diesen und ähnliche Fälle und geeignetere formale Mittel in späteren Abschnitten noch zurückkommen.

8.3.2 Eigenschaften der Fouriertransformation

1. *Linearität* (Eigenschaft der Definition über ein Integral)

$$\mathcal{F}(a_1 f_1(t) + a_2 f_2(t)) = a_1 \mathcal{F}(f_1) + a_2 \mathcal{F}(f_2) \tag{8.24}$$

2. *Reziprozität* (Nachweis durch Variablensubstitution)

$$\mathcal{F}(f(\alpha t))(\omega) = \frac{1}{|\alpha|} (\mathcal{F} f) \left(\frac{\omega}{\alpha}\right) \tag{8.25}$$

Kapitel 4

Populationsdynamik

- Quetelet (1835): “Essai de Physique Sociale”, Abhandlung über Populationsphysik
- Verhulst (1838): Modell für das dichteabhängige Wachstum einer Population (Deterministisch)
- Lotka, Volterrra, Kostizin (1923-1940): Grundlagen der theoretischen Ökologie (Deterministisch)
- Heute: Durch Einsatz von Computern ist die Einbeziehung stochastischer Faktoren in das Modell möglich

Die Populationsdynamik besteht aus der Modellierung eines Zustandes und seiner Dynamik und nicht aus einem Gesetz (wie das Newtonsche Gravitationsgesetz).

4.1 Betrachtung einer einzelnen Spezies

Der einfachste Fall ist eine einzelne Spezies, die sich ausbreitet oder, je nach Randbedingungen auch wieder schrumpfen kann. Ein Beispiel sind Bakterien, die sich vermehren solange Nahrung vorhanden ist.

4.1.1 Iterative Modelle

Zuerst kann man ein iteratives Modell formulieren, in dem die neue Population direkt und in einem Schritt aus der vorherigen hervorgeht.

Exponentielles Wachstum

Die gewöhnlichen Wachstumsvorgänge in der Natur werden meistens durch eine exponentielle Iteration gelöst, da zum Beispiel jedes Tier wieder mehrere Nachkommen besitzt. Jede Folgegeneration P_{n+1} ist das Produkt der Vorgängergeneration mit einem konstanten Faktor a:

$$P_{n+1} = aP_n \ . \tag{4.1}$$

Beginnt man mit einer Anfangspopulation P_0 ergibt sich $P_n = a^n P_0$. Hierbei würde allerdings die Population im Laufe der Zeit bis ins unendliche wachsen, so daß diese Form der Iteration nicht sinnvoll ist.

Logistische Gleichung

Bei Überbevölkerung geht die Population wegen Nahrungsmittelmangel und zu wenig Platz wieder zurück. Um dies zu berücksichtigen, wird in die Formel eine "umweltbedingte Kapazität" k ("Carrier Capacity") eingefügt, die die Population wieder verkleinert:

$$P_{n+1} = (a - k * P_n) * P_n \tag{4.2}$$

Durch die Normierung von $a = \mu$ und $y_n = (k/a)P_n$ entsteht die logistische Gleichung:

$$y_{n+1} = \mu y_n(1 - y_n) \tag{4.3}$$

μ ist in diesem Fall die Reproduktionsrate. Durch Anwenden der Gleichung entsteht in der Population ein Gleichgewicht. Trägt man das Gleichgewicht über der Reproduktionsrate auf, entsteht das Bild einer Bifurkation. Das heißt, wenn die Reproduktionsrate höher wird, entstehen mehrere verschiedene Gleichgewichte. Es folgt eine Kaskade von Bifurkationen von Populationen, bis diese in Chaos übergehen. Dieser Zustand ist für eine Populationssimulation nicht sinnvoll.

4.1.2 Gewöhnliche Differentialgleichungen

Wird die Population durch Differentialgleichungen simuliert, entsteht ein kontinuierliches Zeitverhalten. Die Zustandsvariable ändert sich stetig. Bei hoher Individuenanzahl ist gewährleistet, daß die ganzen Zahlen (die Individuen) problemlos als reele Zahlen genähert werden können. Allerdings ist auch bei dieser Methode keine Simulation statischer Fluktuationen durch Änderung der Bedingungen während der Simulation möglich, sondern nur eine Variation der Anfangsbedingungen.

Formt man diese Beziehungen zu den komplexen Koeffizienten um, so wird

$$\begin{aligned} c_0 &= a_0/2 \\ c_n &= (a_n - ib_n)/2 \\ c_{-n} &= (a_n + ib_n)/2. \end{aligned}$$

Für reelle Funktionen sind natürlich die a_n und b_n reell und man liest ab, daß die Koeffizienten zu positiven und negativen Frequenzen zueinander konjugiert komplex sind: $c_{-n} = \overline{c_n}$. Weiterhin sieht man, daß die c_n aus dem Integral

$$c_n = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(t) e^{-int} \tag{8.16}$$

bestimmt werden können.

Wir listen jetzt noch einige Eigenschaften der c_n für spezielle Funktionsverläufe $f(t)$ auf:

1. *für gerade Funktionen $f(t)$ gilt $c_n = c_{-n}$.* Mit Hilfe der Substitution $t \to -t$ findet man
$$c_n = \frac{1}{2\pi} \int_{-\pi}^{\pi} dt\, f(t) e^{-int} = \frac{1}{2\pi} \int_{-\pi}^{\pi} dt\, f(t) e^{int} = c_{-n}.$$
2. *für ungerade Funktionen $f(t) = -f(-t)$ gilt $c_n = -c_{-n}$.* Der Nachweis verläuft wie für gerade Funktionen.
3. Wie bereits oben erwähnt, gilt für reelle Funktionen $f(t) = \overline{f}(t)$ auch $c_n = \overline{c_{-n}}$, wie man auch direkt nachweisen kann,
$$\overline{c_{-n}} = \frac{1}{2\pi} \overline{\int_{-\pi}^{\pi} dt\, f(t) e^{int}} = \frac{1}{2\pi} \int_{-\pi}^{\pi} dt\, f(t) e^{-int} = c_n.$$

8.3 Fouriertransformation

8.3.1 Herleitung/Definition

Wir verallgemeinern zunächst die komplexen Fourierreihen auf das Intervall $[-T/2, T/2]$:

$$f(t) = \sum_{n=-\infty}^{\infty} c_n\, e^{in\frac{2\pi}{T}t} \tag{8.17}$$

$$c_n = \frac{1}{T} \int_{-T/2}^{T/2} f(t)\, e^{-in\frac{2\pi}{T}t}\, dt \tag{8.18}$$

8.2.1 Fourier-Approximation vs. Interpolation

Bricht man wie im Beispiel eine Fourier-Entwicklung nach einer bestimmten Anzahl von Gliedern ab, so erhält man eine *Approximation* der vorgegebenen Funktion $f(t)$. Im Gegensatz zu den *Interpolationen*, die im vorigen Kapitel beschrieben wurden, kann nicht garantiert werden, daß die Funktionswerte wirklich an einigen Stellen angenommen werden. Häufig ist jedoch gar keine exakte Repräsentation gefragt, sondern nur eine gute Annäherung in einem bestimmten Intervall. Insbesondere ist bei Fourierentwicklungen kein Punkt des Intervalles gegenüber einem anderen ausgezeichnet. Dies ist ein besonderer Vorteil gegenüber Taylor-Approximationen, die nur lokale Entwicklungen in einem einzelnen Punkt sind. Weiterhin verhält sich die Fourierreihe sehr gutmütig in der Gegenwart von Unstetigkeitsstellen, was häufig numerisch von Vorteil sein kann.

8.2.2 Komplexe Fourierreihen

Es liegt nahe, die cos und sin Terme der reellen Fourierreihen zu komplexen Exponentialfunktionen zusammenzufassen. Um zu erreichen, daß Linearkombinationen von solchen komplexen Exponentialfunktionen reell werden, müssen wir in der Regel zulassen, daß die Koeffizienten komplex sind. Weiterhin müssen wir negative Frequenzen berücksichtigen, denn z.B. ist

$$\sin t = \frac{1}{2i}(e^{it} - e^{-it}) \tag{8.13}$$

nur durch Kombination von Frequenz 1 und -1 zu erreichen. Wir wollen im weiteren auch allgemein komplexe Funktionen $f(t)$ zulassen. Der Ansatz für die komplexe Fourierreihe lautet daher

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{int}. \tag{8.14}$$

Durch Ausschreiben der Terme nach der Euler-Beziehung und Zusammenfassen der Terme für n und $-n$ findet man

$$f(t) = c_0 + \sum_{n=1}^{\infty}(c_n + c_{-n})\cos nt + i(c_n - c_{-n})\sin nt. \tag{8.15}$$

Damit ergibt sich für die Beziehungen der Koeffizienten der Fourier-Sinus- und -Cosinus-Reihe

$$\begin{aligned} a_0 &= 2c_0 \\ a_n &= c_n + c_{-n} \\ b_n &= i(c_n - c_{-n}). \end{aligned}$$

Exponentielles Wachstum

Die Änderung des Systems ist hier die Differenz der Zahl der Geburten und der Zahl der Todesfälle, die beide proportional zu der Gesamtpopulation sind. Die Zahl der Geburten ist $b * y$ und die Zahl der Todesfälle $m * y$, wobei y die Individuenanzahl ist.

$$\frac{dy}{dt} = by - my = (b-m)y \tag{4.4}$$

Die Lösung dieser DGL besitzt wieder ein Exponentialverhalten, diesmal aber für kontinuierliche Zeiten:

$$y(t) = y_0 * e^{(b-m)*t} \tag{4.5}$$

Die Lösung ist entweder konstant (bei $y_0 = 0$ immer $y(t) = 0$), wächst ins unendliche (für $b - m > 0$), oder sie verschwindet exponentiell (für $b - m < 0$). Gleichung 4.5 ist also auch keine sinnvolle Beschreibung der meisten Populationen.

Logistisches Wachstum

Wird die Kapazität berücksichtigt (bei Verhulst 1838 und Schäfer 1954), entsteht die nichtlineare DGL

$$\frac{dy}{dt} = \mu y(1 - \frac{y}{K}) \tag{4.6}$$

Die Lösung lautet hierfür:

$$y_1(t) = \frac{K}{1 + (\frac{K}{y_0} - 1)e^{-\mu t}} \tag{4.7}$$

sowie $y_2 = 0$ und $y_3 = K$. Ohne Rauschen ist die DGL stabil, d.h. wenn in Gleichung 4.6 für y der Wert 0 eingegeben wird, bleibt das Ergebnis 0 und ist $y > 0$, ergibt sich als oberer Grenzwert die Kapazität K. Wird aber zu Glg. 4.6 ein Rauschterm (z.B. $A*\cos(t)$, mit sehr kleiner Amplitude A) hinzuaddiert, zeigt sich, daß die Lösung y_2 instabil ist: nach kurzer Zeit steigt die Funktion trotz des Anfangswertes 0 bis zu einem gewissen Wert exponentiell an (Abbildung 1, links). Der obere Grenzwert (y_3) ist auch inklusive Rauschterm stabil: bei einer Störung kehrt die Funktion exponentiell zur Ausgangslösung zurück (Abb. 4.1, rechts).

Es gibt noch weitere logistische Gleichungen, zum Beispiel mit konstanter Ernterate v:

$$\frac{dy}{dt} = \mu y(1 - \frac{y}{K}) - v\ , \tag{4.8}$$

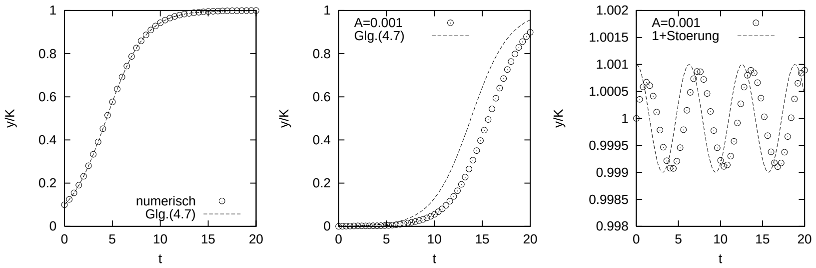


Abbildung 4.1: Numerische Lösung der Populationsgleichung 4.6 mit $\mu = 0.5$ und $K = 1$. Links: $y_0 = 0.1$, $A = 0$, Mitte: $y_0 = 0$, $A = 0.001$, Rechts: $y_0 = K$, $A = 0.001$.

wobei von einer Population immer ein Teil abgezogen ("geerntet") wird. Eine andere Möglichkeit zur Einbeziehung der Kapazität ist das "Mitscherliche Ertragsgesetz" von 1908:

$$\frac{dy}{dt} = \mu(K - y) \tag{4.9}$$

Die Kapazität K ist hier der maximal mögliche Ertrag. Die Gleichung hat den Vorteil, daß sie nicht quadratisch ist und somit am Anfang auch nicht so stark ansteigt, wie die anderen logistischen Gleichungen.

Vergleich logistisches -und exponentielles Wachstum

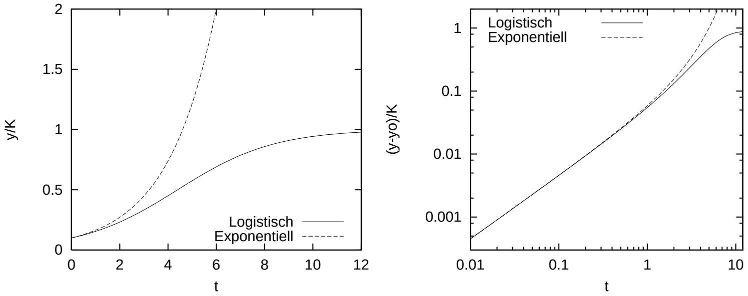


Abbildung 4.2: Vergleich von exponentiellem und logistischem Wachstum, Links: linear, Rechts: Halblogarithmisch. Gezeigt ist $(y(t) - y_0)/K$ aus Gleichung 4.6 (durchgezogene Linie) und exponentielles Wachstum $(1 - y_0/K)y_0(\exp(\mu t) - 1)$ (gestrichelte Linie).

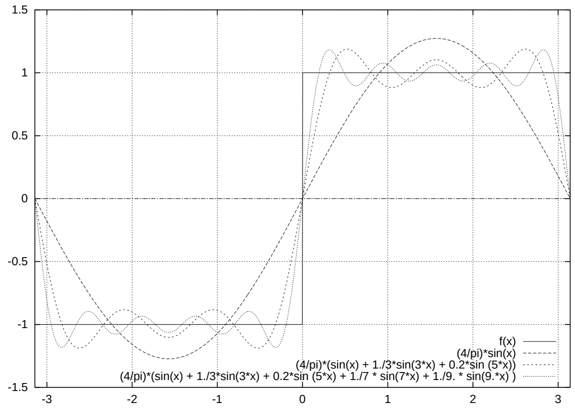


Abbildung 8.1: Rechteckfunktion $f(t)$ und Fourierreihen-Partialsummen gemäß (8.10) für $n = 1, 3, 5$. Man sieht ein Überschwingen der Entwicklung an der Sprungstelle $t = 0$ von $f(t)$. Gemäß des Satzes von Dirichlet nimmt die Entwicklung dort den Mittelwert der beiderseitigen Grenzwerte an, hier ist sogar für alle N der Wert $f_N(t = 0) = 0$.

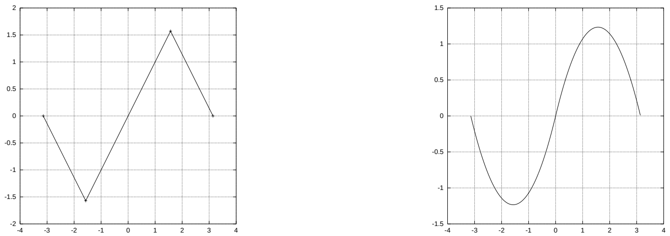


Abbildung 8.2: Ungerade fortgesetzte (a) Dreiecksfunktion $f(t) = t, 0 \leq t < \pi/2,\ \ f(t) = \pi - t, \pi/2 \leq t \leq \pi$ und (b) Parabelbogen $(1/2)t(\pi - t), 0 \leq t \leq \pi$

so daß $f(t) = f(-t)$, dann gilt:

$$\begin{aligned} a_n &= (1/\pi)\int_{-\pi}^{\pi} dt\, f(t)\cos nt \\ &= (1/\pi)\int_{-\pi}^{0} f(t)\cos nt + (1/\pi)\int_{0}^{\pi} f(t)\cos nt \\ &= (1/\pi)\int_{0}^{\pi} f(-t)\cos(-nt) + (1/\pi)\int_{0}^{\pi} f(t)\cos nt = \\ &= (2/\pi)\int_{0}^{\pi} f(t)\cos nt, \qquad (8.7) \\ b_n &= (1/\pi)\int_{-\pi}^{\pi} dt\, f(t)\sin nt \\ &= (1/\pi)\int_{0}^{\pi} f(-t)\sin(-nt) + (1/\pi)\int_{0}^{\pi} f(t)\sin nt \\ &= 0. \qquad (8.8) \end{aligned}$$

Mit anderen Worten verschwinden alle b_n als Koeffizienten ungerader Terme. Analog findet man für ungerade $f(t) = -f(-t)$, daß alle a_n gleich 0 sind, während jetzt die $b_n = (2/\pi)\int dt\, f(t)\sin nt$ nicht mehr verschwinden.

Als Beispiel soll hier die Konvergenz der Entwicklung eines ungeraden Rechteckimpulses $f(t)$

$$f(t) = \begin{cases} -1, & -\pi \le t \le 0 \\ +1, & 0 < t \le \pi \end{cases} \qquad (8.9)$$

gezeigt werden. Dessen Fourierentwicklung beinhaltet nur die b_n für ungerades n und lautet

$$f(t) = \frac{4}{\pi}\sum_{n=1}^{\infty}\frac{1}{2n-1}\sin(2n-1)t. \qquad (8.10)$$

Die Funktion $f(t)$ und einige der ersten Partialsummen der Fourierreihe sind in Abb. 8.1 dargestellt.

Als weitere Beispiele seien die Entwicklungen für die ungerade fortgesetzte Dreiecksfunktion $f(t) = t, 0 \le t < \pi/2$, $f(t) = \pi - t, \pi/2 \le t \le \pi$ genannt (Abb. 8.2a),

$$f(t) = \frac{4}{\pi}\left(\sin t - \frac{sin3t}{3^2} + \frac{sin5t}{5^2} - \ldots\right), \qquad (8.11)$$

sowie für diejenige Funktion, die aus der ungeraden Fortsetzung des Parabelbogens $(1/2)t(\pi - t), 0 \le t \le \pi$ entsteht (Abb. 8.2b):

$$f(t) = \frac{4}{\pi}\sum_{n=1}^{\infty}\frac{sin(2n-1)t}{(2n-1)^3}. \qquad (8.12)$$

An diesen Entwicklungen kann man gut sehen, daß die Fourierreihen umso besser konvergieren, je "glatter" die zu entwickelnde Funktion ist und je näher sie bereits an einer harmonischen Funktion liegt.

In Abbildung 4.2 sieht man, daß die Exponentialfunktion und die Logistische Funktion anfangs gleichschnell wachsen. Man kann also am Anfang nicht unterscheiden, ob ein Wachstumsprozess exponentiell ist und ins unendliche wächst, oder logistisch und sich nach einer Zeit einem Grenzwert nähert. Derartige Prognosen sind bei Phänomenen wie beispielsweise der Verbreitung von AIDS, die es erst seit relativ kurzer Zeit gibt, bedeutungslos.

4.2 Zwei und mehr Spezies

In der Natur gibt es normalerweise mehr als nur eine Art von Lebewesen. Man unterscheidet zwischen verschiedenen Systemen wie z.B. *Konkurrenten* oder *Räuber–Beute–Systeme*. Die Räuber–Beute–Systeme ergeben meistens oszillierende Populationen, wobei der Räuber der Beute "hinterherhinkt", da die Population der Räuber von derjenigen der Beute abhängt. In Abbildung 3 ist dies teilweise nicht der Fall, da sie die Zahl der gefangenen Tiere zeigt, die auch abhängig vom Absatzmarkt sind (Preisentwicklung, Nachfrage).

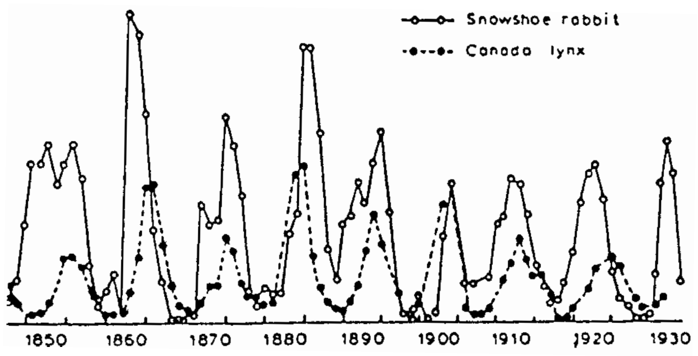


Abbildung 4.3: Räuber–Beute–System

4.2.1 Volterra–Gleichung (1925)

In der Volterra–Gleichung besitzen die Populationen y_i $(i = 1, 2)$ eine unbegrenzte Kapazität. Ausfälle geschehen bei der Beute durch "gefressen" werden, sowie bei den Räubern durch Todesfälle. Die Geburten sind bei der Beute proportional zur Population y_1, und bei den Räubern sowohl proportional zur eigenen Population y_2 als auch zur Population der Beutetiere y_1.

Damit ergibt sich das Gleichungssystem:

$$\begin{aligned}\frac{dy_1}{dt} &= (b_1 - r_{21}y_2)y_1 \\ \frac{dy_2}{dt} &= (-m_2 + r_{12}y_1)y_2\end{aligned} \tag{4.10}$$

b_1 ist die Geburtenrate der Beute, m_2 die Sterberate der Räuber, r_{21} die "gefressene" Beute und r_{12} die "satten" Räuber.

Die in Abb. 4.4 abgebildete Lösung zeigt, das die Räuberpopulation zeitlich nach der Beutepopulation reagiert. Dieses Gleichungssystem besitzt immer periodische Lösungen, und es existieren deshalb keine Gleichgewichtszustände, d.h. auch das "Aussterben" einer Art ist nicht möglich.

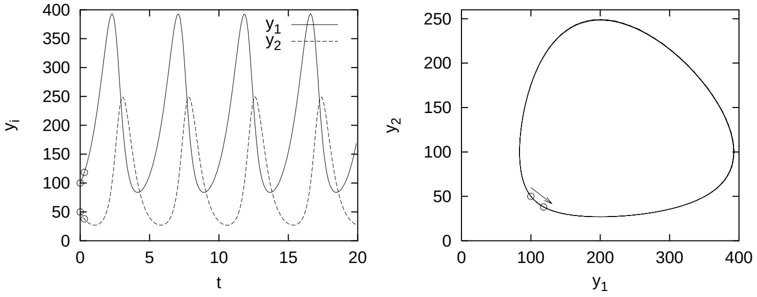


Abbildung 4.4: Räuber–Beute als Funktion der Zeit (links) und im Phasenraum (rechts). Gleichung 4.10 wurde mit den Parametern $b_1 = 1$, $m_2 = 2$, $r_{21} = r_{12} = 0.01$ und den Anfangswerten $y_1(0) = 100$ und $y_2(0) = 50$ numerisch gelöst. Die Kreise zeigen die Populationen zur Zeit $t = 0$ und $t = 0.3$ an, und der Pfeil im rechten Bild deutet die Orientierung im Phasenraum an.

4.2.2 Lotka–Volterra–Gleichung (1926)

Bei der Lottka–Volterra–Gleichung hat die Beutepopulation zusätzlich noch eine begrenzte Kapazität K, sonst sind die Gleichungen unverändert.

$$\begin{aligned}\frac{dy_1}{dt} &= (b_1 - r_{21}y_2 - \frac{y_1}{K})y_1 \\ \frac{dy_2}{dt} &= (-m_2 + r_{12}y_1)y_2\end{aligned} \tag{4.11}$$

definiert sind. Dies wird in der Funktionalanalysis (siehe Quantenmechanik-Vorlesungen) in allgemeinerem Zusammenhang gezeigt. Die genaue Aussage ist dort, daß die Partialsummen $f_N(t)$,

$$f_N(t) = \frac{a_0}{2} + \sum_{n=1}^{N} a_n \cos nt + \sum_{n=1}^{N} b_n \sin nt \tag{8.4}$$

im quadratischen Mittel gegen $f(t)$ konvergieren, d.h., daß die "Summe der Fehlerquadrate" (die sogenannte L_2-Norm der Differenz) für $N \to \infty$ gegen 0 konvergiert:

$$\lim_{N\to\infty} \int_{-\pi}^{\pi} |f(t) - f_N(t)|^2 \, dt = 0. \tag{8.5}$$

Ein wenig schärfer gefaßt ist der *Satz von Dirichlet:* Sei $f(t)$ eine Funktion solcherart, daß sich das Intervall $[-\pi, \pi]$ zerlegen läßt in endlich viele Teilintervalle, in denen $f(t)$ stetig und monoton ist und existieren weiterhin an jeder Unstetigkeitsstelle x_0 die links- und rechtsseitigen Grenzwerte $f(t-0)$, $f(t-0)$, dann konvergiert die Fourierreihe $f_N(t)$ gegen $f(t)$, wo $f(t)$ stetig ist und gegen $(1/2)(f(t+0) + f(t-0))$ an den Unstetigkeitsstellen. Denkt man sich $f(t)$ außerhalb $[-\pi, \pi]$ fortgesetzt, so gilt die Aussage des Dirichlet'schen Satzes für alle t.

Die Fourierkoeffizienten haben anschauliche Bedeutungen. So ist z.B. $a_0/2$ die "Gleichkomponente," der zeitliche Mittelwert des Signals. Sei etwa $f(t) = f_0 = const$, dann ist

$$\frac{a_0}{2} = \frac{1}{2\pi} \int_{-\pi}^{\pi} dt \, f_0 = f_0. \tag{8.6}$$

Der Koeffizient a_1 beschreibt den symmetrischen/geraden Anteil der "Grundschwingung" $\cos t$, b_1 den ungeraden Anteil $\sin t$, usw.

Die Fourierreihe (8.1) vereinfacht sich stark, wenn die zu entwickelnden Funktionen bestimmte Symmetriekriterien erfüllen. Ist z.B. $f(t)$ gerade, d.h. symmetrisch zur y-Achse,

reale Signale	Modelle von Signalverläufen
endliche Signaldauer	unendlich andauernd
endliche Signalamplituden	unbegrenzte Signalamplituden (δ-Impulse)
endliche Signalansstiegszeiten	senkrechte Signalflanken
	periodische Signale
endlicher Energieinhalt	unbegrenzter Energieinhalt

Tabelle 8.1: Häufig anzutreffende Annahmen bei Signalmodellierungen gegenübergestellt den Eigenschaften realer Signale.

(Rauschen). Solche Signale können nicht durch ihren Zeitverlauf, sondern müssen durch allgemeinere Eigenschaften charakterisiert werden.

Für die theoretische Betrachtung muß man häufig vereinfachende Annahmen machen, die auf reale Signale nur in begrenzter Weise zutreffen. Die Tabelle 8.1 listet einige der häufigsten Annahmen auf.

8.2 Kontinuierliche, periodische Signale

Eine auf ganz $\mathbb{R}$ definierte Funktion $f(t)$ mit der Eigenschaft, daß $f(t+T) = f(t)$ für alle t, heißt periodisch mit der Periode T. Wir wollen in der Regel immer das kleinstmögliche positive T verwenden. Ohne Beschränkung der Allgemeinheit können wir dann Funktionen betrachten, die im Intervall $[-\pi, \pi]$ definiert und 2π-periodisch sind. Jede periodische Funktion $f(t)$ läßt sich durch reines Skalieren ihres Argumentes auf diesen Bereich einschränken $\tilde{f}(t') = f(\frac{T}{2\pi}t')$.

Jede periodische Funktion $f(t)$ gegeben auf $[-\pi, \pi]$ kann als eine *Fourierreihe* dargestellt werden:

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos nt + \sum_{n=1}^{\infty} b_n \sin nt. \tag{8.1}$$

Die Variablen a_n und b_n und heißen *Fourierkoeffizienten.*

Die Konvergenz der Fourier-Entwicklung im quadratischen Mittel ist für alle Funktionen gesichert, für die die bestimmenden Integrale der Fourierkoeffizienten

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(t) \cos nt \, dt \tag{8.2}$$

$$b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(t) \sin nt \, dt \tag{8.3}$$

Die begrenzte Kapazität führt dazu, daß nach einer Übergangszeit (“Transiente”) immer ein Gleichgewicht erreicht wird. Es gibt also keine periodischen Lösungen mehr und es ist auch möglich, daß eine Art ausstirbt, wie in Abb. 4.5 für $K = 200$ zu sehen.

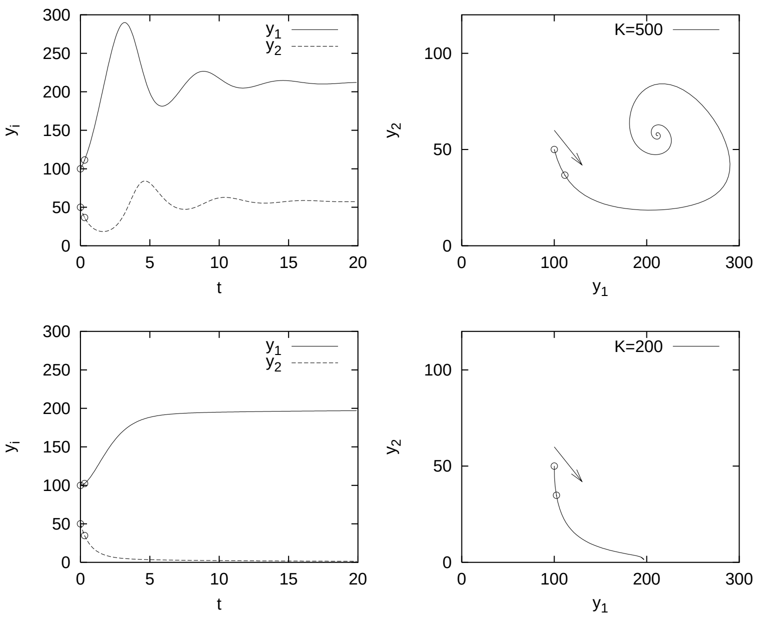


Abbildung 4.5: Räuber–Beute als Funktion der Zeit (links) und im Phasenraum (rechts). Gleichung 4.11 wurde mit den Parametern $b_1 = 1$, $m_2 = 2$, $r_{21} = r_{12} = 0.01$ und den Anfangswerten $y_1(0) = 100$ und $y_2(0) = 50$ numerisch gelöst. Oben hatte die Beutepopulation eine hohe ($K = 500$), unten eine relativ niedrige ($K = 200$) Kapazität. Die Kreise zeigen die Populationen zur Zeit $t = 0$ und $t = 0.3$ an, und der Pfeil im rechten Bild deutet die Orientierung im Phasenraum an.

4.2.3 Folgerungen

Vorteile von Differentialgleichungen

Die Modellierung der Populationsdynamik durch DGL’en ist beliebig allgemein möglich. Bestimmte Prozesse sind sehr gut nachvollziehbar und auch die Zeitabhängigkeit von Parametern (jahreszeitabhängige Schwankungen der Kapazität o.ä.) bei der numerischen

Lösung ist einfach zu implementieren.

Nachteile der differentiellen Modellierung

Bereits kleine Modifikationen können das Verhalten der DGL auf unvorhergesehene (und unerwünschte) Weise beeinflussen, wobei auch ein Erzwingen eines bestimmten Verhaltens sehr schwierig ist. Auswirkungen von Fluktuationen können nicht berücksichtigt werden und die differentiellen Amplituden sind nicht immer sinnvoll (z.B. 0.02 Elefanten).

Hauptsächlich fehlt jedoch die räumliche Auflösung, die im Folgenden genauer betrachtet werden soll.

4.3 Ausblick: Diskrete Modelle

Um die Nachteile der differentiellen Modellierung auszugleichen und die "unendliche" Reichweite der Räuber (jeder Räuber kann jedes Beutetier fressen) einzuschränken, kann man als Alternative auf ein diskretes System ausweichen. Der Algorithmus für ein Räuber–Beute System sieht folgendermaßen aus:

1. Das biologische System wird auf einem Gitter konstruiert.
2. Die Zahl der Individuen pro Gitterplatz wird abgespeichert.
3. *Migrationsschritt*: Räuber und Beute bewegen sich mit einer bestimmten Wahrscheinlichkeit auf einen anderen Gitterplatz.
4. *Jagdschritt*: Die Räuber versuchen in ihrer Umgebung, also auf ihrem Gitterplatz, Beute zu machen.
5. *Geburts/Todesrate*: Mit einer gewissen Wahrscheinlichkeit stirbt der Räuber / das Beutetier, oder bringt Nachkommen hervor. "Satte" Räuber bringen mit höherer Wahrscheinlichkeit Nachkommen hervor und sterben mit niedriger Wahrscheinlichkeit als "hungrige" Räuber.
6. Zurück zu Schritt 2.

Unterhalb einer gewissen Systemgröße passiert es nun, daß die Räuber die Beute ausrotten und anschließend selbst verhungern. Bei mittlerer Systemgröße zeigen sich Oszillationen in der Gesamtpopulation und für große Systeme bleibt sie konstant und fluktuiert nur noch lokal.

Kapitel 8

Analyse von Meßsignalen

Der Inhalt der folgenden Abschnitte über Integraltransformationen läßt sich in ähnlicher oder gleicher Darstellung im Skript *Elektronische Meßtechnik II* von H.-J. Bauer, Fassung vom Sommersemester 1994, im *Taschenbuch der Mathematik*, Verfasser Bronstein und Semendjajew, in *Numerical Recipes* und weiteren Werken nachlesen.

8.1 Klassifikation

In der Praxis vorkommende Signale lassen sich für unsere Zwecke durch einige Eigenschaftspaare charakterisieren:

analog — digital: der Zeitverlauf einer elektrischen Spannung oder eines Stromes in einer elektronischen Schaltung ist im Rahmen der üblichen Meßtechnik eine Größe, die für alle Punkte in einem bestimmten Zeitintervall definiert ist und kontinuierlich die Werte eines bestimmten "dichten" Wertebereiches $\subset \mathbb{R}$ annimmt. Das Vorkommen aller Werte aus diesem Bereich kennzeichnet das Signal als analog. Ein digitales Signal hingegen nimmt, bedingt durch die Repräsentation durch Zahlen mit endlicher Stellenzahl, nur bestimmte diskrete Werte an (häufig sind das nur zwei verschiedene Werte).

kontinuierlich — abgetastet: Bei *kontinuierlichen* Signalen ist für beliebige Zeitpunkte t ein Wert gegeben, bei *abgetasteten* nur für die Abtastzeitpunkte. Digitale Signale sind stets auch abgetastet.

deterministisch — stochastisch: Bei deterministischen Signalen ist der Signalverlauf $f(t)$ "genau" bekannt (d.h. in der Regel in Rahmen der Meßgenauigkeit reproduzierbar), während bei stochastischen Signalen ein statistischer Prozeß mit im Spiel ist

Kapitel 5

Gitter- oder Stochastische Modelle

Wie am Ende des letzten Kapitels angedeutet, stehen in diesem Kapitel nicht-deterministische Methoden im Vordergrund. Im Rahmen diskreter Modelle werden Ereignisse mit einer Wahrscheinlichkeit versehen und sollen dementsprechend häufig vorkommen. Dazu ist es zuerst nötig, Zufallszahlen zu definieren und "herzustellen", mit denen die Ereignisse stochastisch ausgewählt werden können.

5.1 Zufallszahlen

Zufallszahlen sind Folgen von Zahlen in zufälliger (unkorrelierter) Reihenfolge. Die Wahrscheinlichkeit, daß als nächstes eine bestimmte Zahl auftritt ist immer gleich. Das Problem bei Zufallszahlengeneratoren (RNG = random number generator) von Computern ist, daß eine deterministische Maschine nicht-deterministische Zahlenfolgen erstellen soll. Die Lösung ist in der Programmierung von besonders "wildem" Chaos, das selbst im Poincaréschnitt homogen verteilt ist. Besonders wichtig ist hierbei die Modulofunktion, da auf hohe Werte auch genauso wieder niedrigere Werte entstehen können. Bei den Generatoren werden die additiven und die multiplikativen Generatoren unterschieden. Die multiplikativen Generatoren sind einfacher, liefern aber keine so gute Folgen. Die neueren additiven Generatoren sind schwieriger, aber liefert dafür viel bessere Ergebnisse.

5.1.1 Kongruentieller RNG "IBM"

Der für den PC gebräuchlichsten Zufallszahlengenerator ist der multiplikative Generator. Er basiert auf der einfachen Formel :

$$x_{i+1} = (c * x_i) \bmod p \tag{5.1}$$

Der neue Wert errechnet sich dadurch, daß der alte Wert mit einer Konstanten c multipliziert und anschließend modulo einer weiteren Konstanten p genommen. Hierbei wird ein Keim x_0 benötigt, mit dem die Folge begonnen wird. Ein Keim hat den Vorteil, daß bei dem gleichen Wert immer die gleiche Folge entsteht. Dies ist für die Überprüfung von Berechnungen mittels Zufallszahlen sehr wichtig, damit man sie auch nachvollziehen und reproduzieren kann.

Da die Folge nur von der Vorgängerzahl abhängt, wiederholt sie sich beim erneuten Auftauchen einer Zahl periodisch. Für einen guten Generator muß diese Periode möglichst groß sein. Die maximal mögliche Periode ist dann erreicht, wenn alle Zahlen im Bereich bis p einmal durchlaufen wurden, bevor sich eine von ihnen wiederholte. Um die maximale Periode zu erhalten, muß p eine Mersennsche Primzahl, sein. Für sie gilt:

$$p = 2^{\alpha} - 1 \tag{5.2}$$

Hierbei ist α ganzzahlig. Durch die Zahlentheorie wurde auch bewiesen, daß c^p und p teilerfremd sein müssen und es keine andere kleinere Zahl geben sollte, für die das gilt. Formal heißt das :

$$c^p \mod p = 1 \quad \text{und} \quad \nexists q < p : c^q \mod p = 1 \tag{5.3}$$

Für die ersten 32-bit Maschinen von IBM wurden die beiden Konstanten $p = 2^{31} - 1$ (1 Bit Vorzeichen und 31 Ziffern) und die zugehörige Konstante $c = 65539$ verwendet. Für diese Konfiguration ist $x = 0$ ein Fixpunkt und ändert sich nicht. Dafür hat die restliche Folge die Periode $2^{31} - 1$, jede Zahl kommt also nur einmal vor und wiederholt sich erst nach $2^{31} \approx 10^9$ Schritten. Für die Berechnung der Zufallszahl multipliziert man die vorherige mit c und läßt eventuell einen Überlauf zu. Bei einem Überlauf wird die Zahl negativ. Ist sie negativ wandelt man sie einfach wieder in eine positive Zahl um, indem der Wert p addiert wird. Diese Berechnung entspricht dann der Modulofunktion. Dadurch ist dieser Generator sehr schnell. Bei der Implementierung benutzen wir zwei Funktionen: `rng0` zur Initialisierung des Generators (mit einer Zahl "seed") und `rng` für den Generator selbst.

```
#include<iostream>
using std::cout;

const static int p=1U << 32 - 1;        // p=2^31-1
const static int c=65539;
const static int romax=p;
static int x1;

void rng0(int x0);                      // function prototypes
 int rng();

int main()
{
```


4. (mit Einschränkungen:) "periodische" Randbedingungen können gefordert werden, falls $y_n = y_0$ gilt. Dann möchte man üblicherweise auch $y_0' = y_n'$ und $y_0'' = y_n''$ erreichen.

Die ersten drei Zusatzbedingungen führen auf ein sogenanntes tridiagonales System von Gleichungen, das sich durch einmaliges Vorwärtseinsetzen und dann Rückwärtseinsetzen mit geringem Aufwand lösen läßt, siehe z.B. [7.2].

Die letzte Bedingung führt nicht auf ein solch einfaches System und ist daher mit zusätzlichem Aufwand bei der Bestimmung der der y_i'' verbunden. Für Routinen zur Ausführung der Spline-Interpolation und zur Bestimmung von Zwischenwerten verweisen wir ebenfalls auf *Numerical Recipes* [7.2].

7.4 Weiterführende Literatur

[7.1] Hans Rudolf Schwarz. *Numerische Mathematik.* B. G. Teubner, Stuttgart, Germany, 1986.
[7.2] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, Cambridge, 2nd edition 1992.

wobei die Koeffizienten von x_i, x_{i+1} und x abhängen und folgende Form haben,

$$\begin{aligned} A &= (x_{i+1}-x)/(x_{i+1}-x_i), \\ B &= 1-A = (x-x_i)/x_{i+1}-x_i), \\ C &= (1/6)A(A^2-1)(x_{i+1}-x_i)^2, \\ D &= (1/6)B(B^2-1)(x_{i+1}-x_i)^2. \end{aligned}$$

Weil A und B die Werte 0 und 1 bzw. 1 und 0 bei $x = x_i$ und $x = x_{i+1}$ annehmen, sieht man direkt, daß C und D an diesen beiden Punkten verschwinden und das Polynom (7.16) auch weiterhin die Interpolationseigenschaft besitzt. Für die erste und zweite Ableitung von (7.16) nach x errechnet man

$$\begin{aligned} \frac{d}{dx}y &= \frac{y_{i+1}-y_i}{x_{i+1}-x_i} - \frac{3A^2-1}{6}(x_{i+1}-x_i)y_i'' \\ &\quad + \frac{3B^2-1}{6}(x_{i+1}-x_i)y_{i+1}'' \qquad (7.17) \\ \frac{d^2}{dx^2}y &= Ay_i'' + By_{i+1}'' \qquad (7.18) \end{aligned}$$

Wir können jetzt auch sehen, daß (7.16) an den Intervallgrenzen die richtigen Werte für die zweiten Ableitungen annimmt. Wir haben damit einen eindeutigen Ausdruck für das Interpolationspolynom gefunden, ausgedrückt durch die Funktionswerte und die zweiten Ableitungen an den Stützstellen x_i. Im eigentlichen Interpolationsproblem sind die y_i'' allerdings nicht festgelegt (wir hatten das nur der Einfachheit zuliebe eingangs für unsere Argumentation angenommen), wir können daher zwei Bedingungen an die Interpolation stellen. Wir fordern (i) daß an den Verbindungsstellen sowohl die ersten als auch (ii) die zweiten Ableitungen stetig sein sollen.

Formulieren wir diese Bedingung unter Zuhilfenahme von (7.17) und Einsetzen von A bzw. B an den inneren Punkten x_i für die beiden Splines links und rechts von x_i, dann erhalten wir ein Gleichungssystem mit $n-2$ Gleichungen, eine für jeden inneren Stützpunkt x_i, $i = 2, \ldots, n-1$. In diesem System tauchen die y_i'' als Unbekannte auf,

$$\frac{x_i - x_{i-1}}{6}y_{j-1}'' + \frac{x_{i+1}-x_{i-1}}{3}y_i'' + \frac{x_{i+1}-x_i}{6}y_{i+1}'' = \frac{y_{i+1}-y_i}{x_{i+1}-x_i} - \frac{y_i - y_{i-1}}{x_i - x_{i-1}}. \qquad (7.19)$$

Über die restlichen zwei Gleichungen, die für die eindeutige Bestimmung der y_i'' benötigt werden, verfügt man jetzt üblicherweise im Rahmen einer der folgenden Strategien,

1. die *natural spline* Bedingung: festgelegte Werte (üblicherweise = 0) für die zweiten Ableitungen y_1'' und y_n'',
2. feste Werte für die ersten Ableitungen bei x_1 und x_n. Die Werte für die zweite Ableitung an diesen Stellen bestimmt sich dann direkt aus Gl. (7.17),
3. Randbedingungen gemischt aus 1. und 2.,


```
   int r0, r1;

   rng0( 234567 );                // set seed 234567 only once !
   r0=rng();

   for( int i=0; i<20; i++ ){
      r1=rng();
      cout << r0 << ' ' << r1 << '\n';   // print x_n and x_n+1
      r0=r1;
   }
   return 0;
}

// set seed -- initialization function
void rng0(int x0){
    x1=x0;
}

// random number generator
int rng(){
    x1 = (c*x1);                         // allow overflow
    if(x1 < 0) x1-=p;                    // add p if negative
    return x1;
}
```

Anstatt `integers` mit Überlauf zu verwenden kann man auf 64-bit Maschinen `long integers` verwenden, und nur die 32 kleinen Bits verwenden.

Der Algorithmus erzeugt ganze Zahlen zwischen 1 und $2^{31}-1$. Um jetzt auf beliebige (gleichverteilte) reele Zahlenbereiche (z.B. von 0 bis a) zu kommen, multipliziert man a mit der Zufallszahl und teilt durch die höchste mögliche Zahl p.

$$y = \text{double}(x1 * a)/p \quad \Rightarrow \quad y \in]0, a] \qquad (5.4)$$

Trägt man von der Zahlenfolge immer die benachbarten Zahlen in einen Poincaréschnitt so ein (in diesem Fall dann Squaretest genannt), daß nach rechts immer die Zahlen x_n und nach oben die Zahlen x_{n+1} liegen, dann fällt auf, daß alle Zahlen auf parallelen Linien liegen, die zwar dicht beieinander liegen, aber dennoch nicht gleichmäßig homogen verteilt sind Abb. 5.1. Um dieses Defizit auszuschalten und um längere Perioden zu bekommen, wurden additive Generatoren entwickelt.

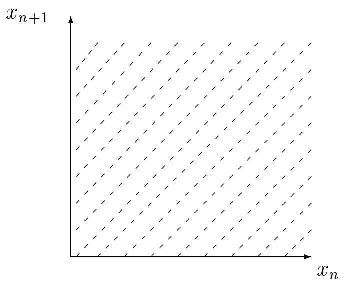


Abbildung 5.1: Squaretest (schematisch) für einen Zufallszahlengenerator.

5.1.2 Lagged Fibbonacci Sequenzen

Die Lagged Fibbonacci Sequenz ist ein additiver Generator, bei dem sich das neue Folgenglied aus der Summe zweier früheren Zufallszahlen ergibt. Die Zahl wird wieder durch die Modulofunktion "zurechtgestutzt". Wir benutzen in diesem Fall einen Generator für Dualzahlen, so daß unsere Berechnung Modulo 2 ist).

$$x_{n+1} = (x_{n-a} + x_{n-b})\mathrm{mod}2 \quad \hat{=}\ x_{n-a} \oplus x_{n-b}\ . \tag{5.5}$$

Ohne Beschränkung der Allgemeinheit sei $a > b$. Das bedeutet, daß wir mindestens eine Teilsequenz von a Zufallszahlen benötigen. Unser Keim ist also die Zahlenfolge $x_1 \cdots x_a$. Um diese Zahlenfolge zu erzeugen, wird meistens ein multiplikativer Generator benutzt. Wiederum sollten die Zahlen a und b gewissen Bedingungen genügen. In diesem Fall sind sie ein "Zierlertrinom", d.h. die Gleichung

$$T_{a,b}(z) = 1 + z^a + z^b \tag{5.6}$$

läßt sich nicht in Unterpolynome faktorisieren, wobei z eine binäre Zahl ist. Die Konstanten a und b werden bis zu der Größe von 100000 gewählt. Die maximale Periode der Zufallszahlen ist $2^a - 1$ und damit um einiges größer als beim IBM-Generator.

Ein einfacher Generator ist der nach seinen Erfindern genannte "Stoll & Kirkpatrick"-Generator. Er benutzt die Konstanten $a = 250$ und $b = 103$. Das folgende Programm benutzt für die Generierung des Keims die beiden Prozeduren von oben. Für die Zufallszahlenserie benutzen wir eine Tafel mit periodischen Rändern. Dabei wird die Speicherplatz für 250 Zahlen reserviert und belegt. Die neue Zufallszahl überschreibt dabei jeweils die letzte mit dem Index $n - a$, die zu ihrer eigenen Berechnung noch gebraucht wurde. Anstatt der Modulofunktion könnte man mit einer `UND`-Verknüpfung einfach das letzte Bit abschneiden, um die Berechnung binärer Zufallszahlen zu beschleunigen.

```
#include<iostream>
```


Eine weitere unangenehme Eigenschaft der polynomialen Interpolation ist auch, daß das Interpolationspolynom insbesondere bei äquidistanter Wahl der Stützstellen dazu neigt, an den Intervallenden stark zu "oszillieren."

Einen Ausweg aus dieser Situation bieten die sogenannten Spline-Funktionen, die an den Anschlußpunkten zwischen den Intervallen garantierte Anschlußbedingungen erfüllen und weiterhin auch eine globale Optimierungseigenschaft haben: Man kann zeigen, daß sie von allen möglichen interpolierenden Funktionen die insgesamt kleinste mittlere Krümmung (Betrag der zweiten Ableitung) aufweisen.

7.3 Spline-Funktionen

Diese Funktionen sind nach dem englischen Begriff für dünne Latten *splines* benannt, die man sich ohne äußere Krafteinwirkung durch die gegebenen Stützpunkte gelegt denkt und die dann eine "glatte" Funktion darstellen. Mit Spline-Funktionen ist es möglich, als bestmögliche Approximation durch n Datenpunkte eine global zweimal stetig differenzierbare Funktion zu legen. Die folgende Darstellung ist i.w. dem Werk *Numerical Recipes* [7.2] entnommen, in dem weitere Details zu finden sind.

Gegeben sei der Datensatz (x_i, y_i), $i = 1, \ldots, n$. Betrachten wir zunächst $x_i < x < x_{i+1}$ und eine lineare Interpolation der Daten in diesem Intervall:

$$f(x) = Ay_i + By_{i+1} = \frac{(x_{i+1} - x)y_i + (x_i - x)y_{i+1}}{x_{i+1} - x_i} \tag{7.15}$$

Die in x linearen Gewichte A und B kann man von der rechten Seite als Vorfaktoren der y_i ablesen, die ansonsten die bekannte Form des interpolierenden Lagrange-Polynomes hat.

Diese Interpolation hat eine verschwindende zweite Ableitung in diesem Interval, jedoch haben die linearen Interpolationen in den Nachbarintervallen im allgemeinen verschiedene erste Ableitungen, so daß diese an den Stützstellen x_i unstetig ist.

Für die weitere Diskussion nehmen wir für einen Moment an, daß wir die Werte y_i'' für die zweiten Ableitungen an den Stellen x_i kennen. Wir addieren jetzt zu unserer linearen Interpolation oben ein kubisches Polynom, dessen zweite Ableitung linear zwischen y_i'' und y_{i+1}'' variiert und das an den Stellen x_i den Wert 0 annimmt. Dies sind vier Bedingungen an ein Polynom dritten Grades, das daher eindeutig bestimmt ist. Wir schreiben das Summenpolynom in der Form

$$y = Ay_i + By_{i+1} + Cy_i'' + Dy_{i+1}'', \tag{7.16}$$

Eingabe		Interpolation			
x	$\tan x$	x	exakt	rational	polynomial
1.1	1.9647597				
1.2	2.5721516	1.15	2.2344969	$2.2344921(5 \times 10^{-6})$	$2.0980797(1 \times 10^{-1})$
1.3	3.6021024	1.25	3.0095697	$3.0095730(1 \times 10^{-6})$	$3.0862589(1 \times 10^{-1})$
1.4	5.7978837	1.35	4.4552218	$4.4552164(6 \times 10^{-6})$	$4.3438010(1 \times 10^{-1})$
1.5	14.1014200	1.45	8.2380928	$8.2381449(5 \times 10^{-5})$	$8.7133064(5 \times 10^{-1})$

Tabelle 7.1: Werte des $\tan(x)$ und Interpolationswerte, die durch polynomiale bzw. gebrochen-rationale Interpolation erhalten wurden. Die Werte in Klammern sind die absoluten Fehler der jeweiligen Interpolation and dieser Stelle

groß. In diesen Fällen ist es besser, durch eine rationale Funktion (Quotient von Polynomen) zu approximieren. Dazu ist die Methode der *Padé-Approximanten* entwickelt worden: Die rationale Funktion $R(x)$ sei

$$R(x) = \frac{P(x)}{Q(x)} = \frac{p_0 + p_1 x + p_2 x^2 + ... + p_\mu x^\mu}{1 + q_1 x + ... + q_\nu x^\nu} \tag{7.13}$$

Es seien n Werte y_i, $i = 1...n$ gegeben. Die Anzahl der Unbekannten ist also $\mu + \nu + 1$, so daß $n = \mu + \nu + 1$ gelten muß, damit wir die Unbekannten bestimmen können. Üblicherweise wählt man $\mu = \nu$ oder $\mu = \nu - 1$, da sich gezeigt hat, daß die Approximation für diese Wahl häufig gut ist. Um die Unbekannten $p_0, ..., p_\mu,\ q_1, ..., q_\nu$ zu bestimmen, stellt man die N Gleichungen

$$R(x_i) = f_i\,, \quad i = 1...N \tag{7.14}$$

auf und bestimmt daraus die Unbekannten. Man kann so zwar immer eine Lösung erhalten (unter Umständen auch mit verschwindenden Koeffizienten für die führenden Potenzen), jedoch ist nicht immer die Interpolationseigenschaft garantiert. Insbesondere kann eine Nullstelle sowohl im Nenner– als auch im Zählerpolynom auftreten und die nach Herauskürzen übrigbleibende interpolierende Funktion nimmt in der Regel dann an diesen Stellen den Funktionswert nicht mehr an. Man muß dann die Ordnungen des Zähler– und Nennerpolynoms ändern, um ggf. eine gültigen Interpolation zu erhalten.

Die Effizienz dieser Methode im Vergleich zur polynomialen Interpolation für Funktionen mit Polstellen kann man an der Funktion $\tan(x)$ gut sehen. Die Ergebnisse kann man in Tabelle 7.1 miteinander vergleichen. Besonders zu beachten ist der Fehler für Werte nahe $\frac{\pi}{2} \approx 1.57$.

Die bisher behandelten Interpolationsverfahren beschränken sich auf ein Intervall. Will man eine globale Funktion erhalten, so kann man in entsprechend vielen benachbarten Intervallen interpolieren und die Ergebnisse aneinanderreihen. Man erhält zwar eine stetige Funktion, die aber an den Berührstellen der einzelnen Intervalle in der Regel nicht differenzierbar ist, da sie dort einen Knick besitzt, was eigentlich immer unerwünscht ist.


```
using std::cout;

// Include from header file ...
const static int p=1U << 32 - 1;            // p=2^31-1
const static int c=65539;
static int x1;
void rng0(int x0);                        // function prototypes
int rng();                                // IBM kongruential

const static int kps250=250;
const static int kps103=103;
static long int kps[kps250];              // use long int for kps[]
static int ikps1, ikps2;
void rkps0(int x0);                       // function prototypes
int rkps();                               // Kirckpatrick & Stoll

int main()
{
  int r0, r1;

   rkps0( 234567 );
   r0=rkps();

   for(long int i=0; i<20; i++ ){
      r1=rkps();
      cout << r0 << ' ' << r1 << '\n';
      r0=r1;
   }
   return 0;
}

// add also the rng functions ...

// Kirkpatrick & Stoll initialization
void rkps0(int x0){
    rng0( x0 );
    rng(); rng(); rng();                  // disregard the first 3
    for( int i=0; i<kps250; i++){
        kps[i]=rng();                     // initialize the field
    }
    ikps1=0;                              // initialize counters
    ikps2=kps250-kps103;
}

// Kirkpatrick & Stoll random number generator
```

```
int rkps(){
    ikps1++;   if(ikps1 >= kps250) ikps1=0;
    ikps2++;   if(ikps2 >= kps250) ikps2=0;
    long int kpsnew=(kps[ikps1]+kps[ikps2]);
    kps[ikps1]=kpsnew%p;
    return int(kpsnew%p);
}
```

5.1.3 Testen von Zufallszahlen

Es gibt viele Möglichkeiten Zufallszahlen auf ihre "Zufälligkeit" zu testen. Im Folgenden kommen Beispiele ohne die mathematischen Grundlagen weiter zu vertiefen.

1. Squaretest
   Der schon vorher erwähnte Test mittels des Poincaréschnitts in zwei Dimensionen (deswegen: Square=Quadrat). Die Zufallszahlen sollten ein homogenes Bild ergeben.

2. Cubetest
   Der Cubetest entspricht dem Squaretest, nur daß er im Dreidimensionalen stattfindet. Die drei aufeinanderfolgenden Zahlen werden dreidimensional aufgetragen und sollten ebenfalls homogen im Raum verteilt sein.

3. Mittelwert
   Das arithmetische Mittel der Zufallszahlen sollte dem rechnerischen Mittelwert entsprechen. Ist $x \in]0,1]$, dann sollte das arithmetische Mittel sein:

$$\bar{x} = \lim_{x\to\infty} \frac{1}{N} \sum_{x=1}^{N} x_n = \frac{1}{2}\ . \tag{5.7}$$

   Mit weiteren Folgengliedern soll dem rechnerischen Mittel also immer weiter genähert werden.

4. Fluktuation des Mittelwertes
   Dies ist der sogenannte χ^2-Test. Die Verteilung um den Mittelwert sollte ein Gaußsche Glockenkurve ergeben.

5. Fourieranalyse
   Die Zufallszahlenfolge kann als Funktion angesehen werden. Durch eine Schnelle–Fourier–Transformation (FFT) wird der sogenannte Spektraltest durchgeführt. Dabei wird darauf geachtet, ob als Ergebnis ein "weißes Rauschen" (gute Zufälligkeit) oder irgendwelche Spitzen entstehen (Resonanzen).

An der oberen Diagonale lesen wir das Interpolationspolynom

$$P(x) = -37 + (x+2)(19 + x(-6 + (x-1)3)) = 1 + x - 3x^2 + 3x^3 \tag{7.12}$$

ab.

Als Algorithmus läßt sich unser Vorgehen etwa so formulieren:

```
template<class T>
void pts_neville(int n, const T& x, const T& y, T &result ){
  // x contains the arguments, y the values of the function to be interpolated
  // result will contain the upper diagonal of the Neville scheme,
  // i.e. the coefficients of the interpolating polynomial
  // P(x) = result[0] + result[1]*(x-x0) + result[2]*(x-x1)(x-x0) +...

  // initialize the result with the f(x_i)/y_i
  for (int i=0; i < n; ++i )
    result[i] = y[i];

  // increasing 'order' of the interpolation polynomial
  for (int k=0; k < n; ++k )
    for (int i=n; i> k ; --i){
      // compute new column of divided differences
      result[i] = (result[i]-result[i-1])/(x[i] - x[i-k-1]);
    }
}
```

Eine abgewandelte Form dieser Berechnungsvorschrift läßt sich auch vorteilhaft verwenden, wenn nur ein einzelner zusätzlicher Wert und nicht allgemeiner die Koeffizienten des Polynoms berechnet werden sollen. Dazu wird wieder die rekursive Vorschrift (7.9) angewandt, aber jetzt die Faktoren $(x - x_i)$, die auf der rechten Seite auftreten, bei der Berechnung gleich mit berücksichtigt. Ein derart abgewandeltes Schema läßt sich hervorragend zur Extrapolation nutzen. So wird z.B. beim Romberg-Verfahren der Integration der Wert eines für verschiedene Schrittweiten h berechneten Integrals nach $h \to 0$ extrapoliert [7.1].

7.2 Rationale Interpolation

Polynomiale Interpolationsverfahren eignen sich gut für glatte Funktionen. Hat aber die zu interpolierende Funktion Pole (oder auch weniger starke Singularitäten, wie z.B. $1/\sqrt{(x)}$), so können diese prinzipiell nicht durch eine polynomiale Interpolation erfaßt werden und in der Nähe der Polstellen sind häufig die auftretenden Fehler unakzeptabel

den Nenner kürzt. Nun ist aber $P_{i_0 i_1 \ldots i_{k-1}}(x_{i_0}) = y_{i_0}$, so daß auch der Gesamtausdruck diesen Wert annimmt. Ähnlich geht man für die übrigen Werte vor.

Auf der linken Seite tauchen nur Polynome k-ten Grades auf, das Gesamtpolynom seinerseits hat $k+1$-ten Grad. Daher kann man man sowohl das Interpolationspolynom, aber auch Werte an bestimmten Stellen x, durch sukzessive Kombination niedergradiger Polynome (bzw. deren Funktionswerte bei x) berechnen. Hierauf beruht das Verfahren von Neville.

Zunächst werden als "Spezialfälle" von (7.9) die Interpolationspolynome zwischen zwei benachbarten Punkten x_i und x_{i+1} bestimmt. Dies ist analog zur Lagrange-Interpolation 1. Ordnung und man erhält eine Gerade:

$$P_{i,i+1}(x) = \frac{1}{x_i - x_{i+1}}((x - x_{i+1})y_i - (x - x_i)y_{i+1}) \tag{7.10}$$

Ein solches Polynom wird für alle benachbarten Punkte x_i, x_{i+1} und die dazugehörigen Werte y_i, y_{i+1} aufgestellt. Die n resultierenden Polynome ersten Grades sind dann $P_{0,1}$, $P_{1,2}$, $\ldots$, $P_{n-1,n}$. "Benachbarte" Polynome werden jetzt als Ausgangsbasis in (7.9) für die nächste Generation von Polynomen, jetzt zweiten Grades benutzt. Das Ergebnis sind die $n-1$ Polynome $P_{0,1,2}$, $P_{1,2,3}, \ldots, P_{n-2,n-1,n}$. Dieses Verfahren des paarweisen Kombinierens führen wir weiter, bis wir am Interpolationspolynom n-ten Grades angelangt sind.

Dieses rekursive Berechnungsschema läßt sich noch weiter vereinfachen, indem wir nur die jeweils führenden Koeffizienten der linken und rechten Seite in (7.9) miteinander vergleichen. Nennen wir den Vorfaktor von $(x - x_{i_0})\ldots(x - x_{i_k})$ des Polynomes $P_{i_0 i_1 \ldots i_k}(x)$ beispielsweise $f[i_0, i_1, \ldots, i_k]$, dann ist

$$f[i_0, i_1, \ldots, i_k] = \frac{f[i_1, \ldots, i_k] - f[i_0, i_1, \ldots, i_{k-1}]}{x_{i_k} - x_{i_0}}. \tag{7.11}$$

Dies bedeutet, daß wir die Koeffizienten sukzessive als dividierte Differenzen bestimmen können. Als Beispiel ermitteln wir das Interpolationspolynom durch die Punkte $(-2, -37), (0, 1), (1, 2), (2, 15)$. Man beachte, daß immer durch die Differenz der zu den nicht gemeinsamen Indizes gehörenden x_i dividiert wird:

x_i	y_i			
-2	$\boxed{f[0] = -37}$			
0	$f[1] = 1$	$\boxed{f[0,1] = \frac{1-(-37)}{0-(-2)} = 19}$		
1	$f[2] = 2$	$f[1,2] = (2-1)/(1-0) = 1$	$\boxed{f[0,1,2] = \cdots = -6}$	
2	$f[3] = 15$	$f[2,3] = (15-2)/(2-1) = 13$	$f[1,2,3] = \cdots = 6$	$\boxed{\text{f[0,1,2,3]= 3}}$

6. Korrelationsfunktionen
   Untersuchung von Korrelationsfunktionen wie zum Beispiel

$$< x_n * x_{n+t} > - < x_n^2 > \tag{5.8}$$

Außerdem gibt es noch Tests mit den schönen Namen "Christmas-Test" und "Butterfly-Test", sowie viele andere.

5.2 Zellularautomaten

Bereits am Ende der Populationsdynamik gibt es ein Beispiel für ein Gittermodell. Ohne Benutzug von Zufallszahlen hat man das einfachste Gittermodell, einen Zellularautomaten. Ein Zellularautomat ist die (zeitliche) Entwicklung auf einem diskreten Gitter von D Dimensionen und einem System von Freiheitsgraden nach einer bestimmten Regel (Dynamik). Jede Zelle des Gitters wird durch eine Variable repräsentiert. Bei jedem Zeitschritt wird die Regel auf alle Variablen gleichzeitig angewendet.

Ein einfaches Beispiel ist eine Gruppe Kinder, die sich im Kreis aufstellen und immer dann ein Signal geben, wenn genau ein Nachbar ein Signal gibt. In Abbildung 5.2 sind die Kinder in zwei aufeinanderfolgenden Zeitschritten zu sehen.

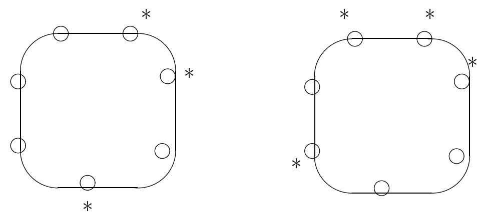


Abbildung 5.2: Beispiel der Anwendung der Regel 90.

Der Name der Regel 90 stammt aus einer Klassifizierung des Engländers Wolfram von 1992. Sie bedeutet im Prinzip die Exklusiv-Oder-Verknüpfung der beiden Nachbarn. Die Klassifizierung wird klar wenn man die sog. Wahrheitstafel für die oben beschriebene Regel aufstellt.

Vorgänger	111	110	101	100	011	010	001	000
Nachfolger i	0	1	0	1	1	0	1	0

In der ersten Zeile sind alle möglichen Kombinationen der Vorgänger $i-1$, i und $i+1$ aufgelistet, und in der zweiten Zeile ist der neue Wert gezeigt, den i nach Anwendung der Regel einnimmt. Zusammengenommen ist die Nachfolger-Zeile die binäre Repräsentation der Zahl $90 = 2^6 + 2^4 + 2^3 + 2^1 = 01011010$. Jede andere Regel läßt sich nun mit der Wahrheitstafel bestimmen.

Um den Kreis zu simulieren benötigen wir ein Feld für jede Position, mit den Einträgen 0 oder 1, die anzeigen ob im Moment ein Signal gegeben wird, oder ob nicht. Formal geschrieben ist dies: $f(i) = \{0, 1\}$ mit $i = 1, \cdots, N$, für N Positionen. Durch die Regel 90 werden ist die neue Variable: $f'(i) = [f(i+1)$ und $\{$nicht $f(i-1)\}]$ oder $[\{$nicht $f(i+1)\}$ und $f(i-1)]$, bzw. in symbolischer Schreibweise

$$f'(i) = (f(i+1) \wedge \neg f(i-1)) \vee (\neg f(i+1) \wedge (f(i-1)) \ , \tag{5.9}$$

oder ganz einfach

$$f'(i) = f(i+1) \oplus f(i-1) \quad \text{für } i = 2, \cdots, N-1, \tag{5.10}$$

wobei $\oplus$ die Exklusiv-Oder-Verknüpfung mit folgender Funktionstabelle ist:

$\oplus$	0	1
0	0	1
1	1	0

Bisher konnten nur die Feldeinträge $i = 2, \cdots, N-1$ behandelt werden, da für $i = 1$ und $i = N$ kein linker bzw. rechter Vorgänger existiert. Die Ränder können einzeln für die Berechnung abgefragt werden:

$$\begin{aligned} f'(1) &= f(2) \oplus f(N) \ , \\ f'(N) &= f(1) \oplus f(N-1) \ , \end{aligned} \tag{5.11}$$

Eleganter, aber auch aufwendiger, sind periodische Randbedingungen im Rahmen einer verketteten Liste. Um die Liste nach links und nach rechts periodisch fortzuführen werden zwei neue Felder f_l und f_r angelegt, die jeweils auf den rechten, bzw. auf den linken Nachbarn zeigen. Die Initialisierung dieser Felder erfolgt mit:

$$\begin{aligned} f_l(i) = i-1 \ \ (i \geq 2), &\qquad \text{und} \quad f_l(1) = N \ , \\ f_r(i) = i+1 \ \ (i \leq N-1), &\qquad \text{und} \quad f_r(N) = 1 \ . \end{aligned} \tag{5.12}$$

Die neue Regel in Glg. 5.10 lautet somit

$$f'(i) = f(f_r(i)) \oplus f(f_l(i)) \ . \tag{5.13}$$

die Interpolationsbedingungen erfüllt, denn für den speziellen Fall, daß x_k eine Stützstelle ist, gilt

$$P(x_k) = \sum_{i=0}^{n} y_i L_i(x_k) = \sum_{i=0}^{n} y_i \delta_{ik} = y_k. \tag{7.7}$$

Dieses Polynom ist eindeutig, denn gäbe es ein zweites mit den gleichen Eigenschaften, dann wäre auch die Differenz ein Polynom n-ten Grades und müßte an den $n+1$ Stützstellen x_i insgesamt $n+1$ Nullstellen aufweisen. Polynome n-ten Grades können aber maximal n Nullstellen aufweisen, und wir erhalten einen Widerspruch zu unserer Annahme.

Bei dieser Methode werden alle Stützstellen x_i gleich behandelt. Allerdings ist sie für numerische Berechnungen ungeschickt, da mehr als $(n+2)(n+1)$ wesentliche Operationen durchgeführt werden müssen: n Multiplikationen für den Zähler jedes Lagrange-Polynoms sowie eine Division, dann Bestimmung von insgesamt $n+1$ solcher Polynome, und $(n+1)$ weitere Multiplikationen mit den y_i. Ohne weitere Herleitung geben wir hier daher die sogenannte *baryzentrische* Darstellung der Lagrange-Interpolation an, die sich in einigen Schritten aus (7.6) gewinnen läßt:

$$P(x) = \sum_{i=0}^{n} \mu_i y_i / \sum_{i=0}^{n} \mu_i, \quad \mu_i = \frac{1}{x - x_i} \prod_{i \neq j}^{n} \frac{1}{x_i - x_j} \tag{7.8}$$

Bis auf die in den μ_i vorkommenden Faktoren $x - x_i$ sind die vorkommenden Größen konstant und können berechnet werden, sobald die Lage der Stützstellen bekannt ist. Der Berechnungsaufwand für eine Interpolation für $n+1$ Stützstellen beträgt dann noch $2(n+1)$ Multiplikationen zur Bestimmung von Zähler und Nenner, sowie eine Division. Diese Darstellung hat die Form eines gewichteten Mittels der Funktionswerte y_i. Tatsächlich geht aus der Gleichung für die μ_i hervor, daß das Gewicht umso größer wird, je näher x and der Stützstelle x_i liegt.

7.1.2 Neville Schema

Weitere nützliche Darstellungen des Interpolationspolynoms lassen sich aufgrund einer Rekursionsformel gewinnen. Bezeichne $P_{i_0 i_1 \ldots i_k}(x)$ das Interpolationspolynom durch die Punkte (x_i, y_i), $i = i_0, i_1, \ldots i_k$, das also durch eine Teilmenge der Stützstellen x_i festgelegt wird. Dann gilt

$$P_{i_0 i_1 \ldots i_k}(x) = \frac{(x - x_{i_0}) P_{i_1 i_2 \ldots i_k}(x) - (x - x_{i_k}) P_{i_0 i_1 \ldots i_{k-1}}(x)}{(x_{i_k} - x_{i_0})}. \tag{7.9}$$

Dies kann man sehen, indem man einfach die Stützstellen einsetzt und zeigt, daß das Resultat tatsächlich der Funktionswert an der jeweiligen Stelle ist. Beispielsweise überlebt für x_{i_0} nur der zweite Term im Zähler, dessen Vorfaktor sich bis auf das Vorzeichen gegen

7.1 Polynomiale Interpolation

Die einfachsten Interpolationsverfahren beruhen auf der Idee, $f(x)$ als ein Polynom Nter Ordnung mit $N+1$ unbekannten Koeffizienten a_i anzusetzen

$$P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_N x^N. \tag{7.2}$$

In der Regel werden dann auch $N+1$ Bedingungen zur Bestimmung der Koeffizienten benötigt, d.h., daß das Interpolationspolynom durch eine an $n+1$ Stellen gegebene Funktion im allgemeinen $N = n$ter Ordnung ist: Durch zwei Punkte läßt sich eine Gerade legen, durch drei Punkte eine Parabel, usw.

Der obige Ansatz führt unter Auswertung der Interpolationsbedingungen (7.1) auf ein *lineares* Gleichungssystem

$$\begin{aligned} P(x_1) = y_0 &= a_0 + a_1 x_0 + a_2 x_0^2 + \ldots a_n x_0^n \\ P(x_1) = y_1 &= a_0 + a_1 x_1 + a_2 x_1^2 + \ldots a_n x_1^n \\ P(x_2) = y_2 &= a_2 + a_1 x_2 + a_2 x_2^2 + \ldots a_n x_2^n \\ \ldots & \\ P(x_n) = y_n &= a_0 + a_1 x_n + a_2 x_n^2 + \ldots a_n . x_n^n \end{aligned} \tag{7.3}$$

Wenn die x_i paarweise verschieden sind, sind die Zeilengleichungen linear unabhängig und die Lösung dieses Systems legt eindeutig das Interpolationspolynom fest (die Eindeutigkeit wird auch aus dem folgenden Abschnitt klar werden). Dieses Verfahren ist auch für allgemeinere als polynomiale Interpolationsansätze anwendbar. Für Polynome hingegen müssen wir es gar nicht lösen, sondern können auf eine explizite Darstellung der Lösung zurückgreifen, die wir jetzt besprechen wollen.

7.1.1 Lagrange-Interpolation

Betrachten wir zunächst die sogenannten Lagrange-Polynome [7.1], denen wir schon einmal im Zusammenhang mit der Gauß-Qaudratur begegnet sind (siehe Gl. 6.45),

$$L_i(x) = \prod_{k=1}^{n} \frac{(x-x_0)(x-x_1)\ldots(x-x_{i-1})(x-x_{i+1})\ldots(x-x_n)}{(x_i-x_0)(x_i-x_1)\ldots(x_i-x_{i-1})\ldots(x_i-x_{i+1})\ldots(x_i-x_n)}. \tag{7.4}$$

Diese Polynome sind vom Grad n und besitzen die Eigenschaft

$$L_i(x_k) = \delta_{ik} = \begin{cases} 1, & \text{falls } i = k \\ 0, & \text{falls } i \neq k. \end{cases} \tag{7.5}$$

Dann ist klar, daß das Polynom

$$P(x) = \sum_{i=0}^{n} y_i L_i(x) \tag{7.6}$$

Ebenso brauchbar und dabei weniger aufwendig sind sog. Schatten oder Bildfelder. Im Fall der linearen Kette werden einfach neue Feldelemente $f(0)$ und $f(N+1)$ eingeführt, die immer die Werte

$$\begin{aligned} f(0) &= f(N) \\ f(N+1) &= f(1) \end{aligned} \tag{5.14}$$

enthalten. Damit ist die Regel in Gleichung 5.10 ohne weiteres auf alle $i = 1, \cdots, N$ anwendbar. Ein entsprechendes einfaches Programm verwendet **rkps** und das C++ Exklusiv-Oder: `^`.

```
int main()
{
   const int NF=150;               // field length 150
   int f[NF+2], f1[NF+2];          // field + shadow elements

   rkps0( 1 );                     // Initialize rkps

   for(int i=1; i<=NF; i++ ){
       f[i]=rkps();                // or f[i]=0;
   }
   f[0]=f[NF];                     // shadow left
   f[NF+1]=f[1];                   // shadow right
   // f[NF/2]=1;                   // to set only one 1

   for(int t=0; t<150; t++ ){      // 150 timesteps
      for(int i=1; i<=NF; i++ )
         if( f[i] == 1 ) cout << t << ' ' << i << '\n';
      cout << '\n';                // output of (t,i) if f(i)==1

      for(int i=1; i<=NF; i++){    // set ALL new fields f1
         if(( f[i-1] == 1 ) ^ ( f[i+1] == 1 ))
            f1[i]=1;
         else
            f1[i]=0;
      }
      for(int i=1; i<=NF; i++){    // write new values to f
         f[i]=f1[i];
      }
      f[0]=f[NF];                  // shadow left
      f[NF+1]=f[1];                // shadow right
   }
   return 0;
}
```

Werden die einzelnen Zeilen grafisch untereinander ausgegeben (schwarz für eins und weiß für Null) ergibt sich für die Regel 90 ein unregelmäßiges Bild, siehe Abbildung 5.3 links. Regel 90 ist also eine chaotische Regel. Andere Regeln ergeben nach kurzer Zeit ein ganz weißes Bild (Regel 40), eine örtlich fixierte Struktur (Regel 78) oder eine wandernde Struktur (Regel 138). Die Anfangskonfiguration der Grafiken ist hier zufällig gewählt.

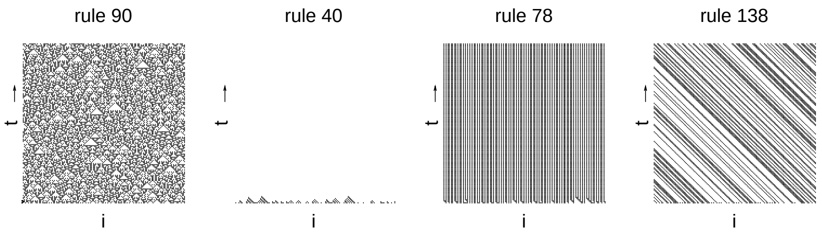


Abbildung 5.3: Jeweils 150 Zeit- und Ort-Schritte für die Regeln 90, 40, 78 und 138 bei zufälligen Anfangsbedinungen.

Besitzt die Anfangskonfiguration bei der Regel 90 nur eine einzige Eins, dann entsteht als Bild das sogenannte Sierpinski-Sieb (auch Sierpinski-Dreieck genannt). Eine fraktale (also selbstähnliche) Figur, die sich dadurch auszeichnet, daß sie Dilatationsinvariant (Dehnungsinvariant) ist und bei Vergrößerung selbstähnliche Figuren entstehen.

Schematisch dargestellt entwickelt sich die erste Eins wie folgt:

$$\begin{array}{ccccccccccccccc}
 & ... & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & ... & \\
 & & & & & & \mathbf{1} & 0 & \mathbf{1} & & & & & & \\
 & & & & & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & & & & & \\
 & & & & \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} & & & & \\
 & & & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & & & \\
 & & \mathbf{1} & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & \mathbf{1} & & \\
 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & \\
\mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1}
\end{array}$$

und entsprechend der Darstellung in Abb. 5.3 ergibt sich Abbildung 5.4.

5.3 Game of life

Ein sehr bekanntes Beispiel für Zellularautomaten ist das 1970 von John Conway entwickelte Populationssystem: Game of life. Es besteht aus einem zweidimensionalen Sy-

Kapitel 7

Interpolation und Approximation

Interpolation bedeutet, zu einer Funktion, die nur an $n+1$ diskreten vorgegebenen Stützstellen x_i, $i = 0 \dots n$, mit den zugehörigen Funktionswerten y_i bekannt ist, eine Interpolationsfunktion $f(x)$ zu finden, für die einerseits die Interpolationsbedingungen

$$f(x_i) = y_i \tag{7.1}$$

gelten, die aber andererseits auch zwischen den x_i definiert ist, im allgemeinen für $x_0 \leq x \leq x_n$.

Vor Einführung des Computers als wichtigstes Rechenhilfsmittel wurden zur Berechnung von Funktionen Rechentafeln (z.B. die berühmten Logarithmentafeln) benutzt. Da in diesen Tabellen nur bestimmte Werte der Funktion aufgelistet waren, war es wichtig, gute Interpolationsverfahren zu kennen, um Zwischenwerte gewinnen zu können. Heute ist diese Anwendnung zwar zurückgedrängt worden, aber weil eine Interpolation häufig numerisch nicht sehr aufwendig durchzuführen ist, findet sie Anwendung bei

1. der Elimination teurer Funktionsberechnungen in zeitkritischen Teilen eines Algorithmus (Kraftberechnung aus Tabellen),
2. bei der numerischen Differentiation,
3. bei der Interpolation zwischen zwei verschiedenen Diskretisierungen,
4. bei der Datenreduktion: eine Bezier-Kurve (Vorsicht: ist keine eigentliche Interpolation, da sie nicht durch die Stützpunkte verläuft) ist bereits durch die Lage weniger Kontrollpunkte festgelegt,
5. bei der *Extrapolation*. Bestimmte Integrationsverfahren (Romberg) beruhen darauf, die Genauigkeit eines Verfahrens dadurch zu verbessern, daß eine Interpolation des Resultates als Funktion des Stützstellenabstandes h für $h \to 0$ durchgeführt wird.

Deshalb sollen im folgenden verschiedene Interpolationsverfahren vorgestellt werden.

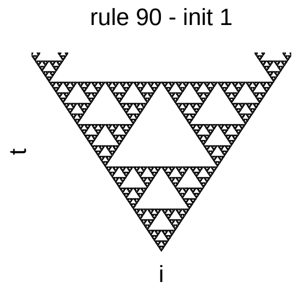


Abbildung 5.4: 300 Zeit-Schritte für die Regel 90 mit einer Eins als Anfangsbedinung.

stem von Zellen, wobei jede Zelle entweder ein Wesen besitzt oder leer ist. Ob die Zelle nach dem nächsten Zeitschritt besetzt ist, hängt von der Summe der besetzten Felder aus den acht Nachbarzellen ab. Gibt es zu wenige Nachbarn, dann geht ein eventuell vorhandenes Wesen durch Vereinsamung ein. Sind zu viele Nachbarn vorhanden, dann stirbt es durch Überbevölkerung. Nur für wenige Nachbar-Häufigkeiten überlebt es, oder entsteht es überhaupt. Im Folgenden sind die Fälle beschrieben, wobei k die Summe der Wesen in den acht Nachbarfeldern ist und $n(i,j)$ die entsprechende betrachtete Zelle ist:

1. $k > 3 : f'(i,j) = 0$ (Überbevolkerung)
2. $k = 3 : f'(i,j) = 1$ (Idealfall)
3. $k = 2 : f'(i,j) = f(i,j)$ (Wert bleibt gleich)
4. $k < 2 : f'(i,j) = 0$ (Vereinsamung)

Im folgenden Programmbeispiel benutzen wir wieder ein Schattenfelder für die Realisierung der periodischen Randbedingung. Die Initialisierung des Feldes erfolgt genauso wie beim letzten Beispiel, nur in zwei Dimensionen. Die Zellen werden durch ein zweidimensionales Feld von ganzen Zahlen dargestellt, das die Werte Eins und Null enthält, wobei Eins bedeutet, daß das Feld lebt. In jedem Schritt wird für alle Felder die Summe der acht Nachbarfelder der Variablen `k` zugewiesen. Die neuen Werte f' werden erst einmal auf Null gesetzt und nur in den Fällen $k = 3$ bzw. eventuell bei $k = 2$ auf Eins gesetzt.

```
int main()
{
   const int NF=200;              // field length 200
   int f[NF+2][NF+2];
```

```
int f1[NF+2][NF+2];            // field + shadow elements
int k=0, fsum=0;
rkps0( 1 );                    // Initialize rkps

for(int i=1; i<=NF; i++ ){
  for(int j=1; j<=NF; j++ ){
    f[i][j]=rkps();
    if( f[i][j] == 1 ) fsum++; // Random initial condition
  }
}
for(int j=1; j<=NF; j++ ){
   f[0][j]   =f[NF][j];        // shadow bottom
   f[NF+1][j]=f[1][j];         // shadow top
}
for(int i=1; i<=NF; i++ ){
   f[i][0]   =f[i][NF];        // shadow right
   f[i][NF+1]=f[i][1];         // shadow right
}
f[0][0]      =f[NF][NF];       // shadow bottom left
f[NF+1][0]   =f[1][NF];        // shadow top left
f[0][NF+1]   =f[NF][1];        // shadow bottom right
f[NF+1][NF+1]=f[1][1];         // shadow top right

for(int t=0; t<2000; t++ ){    // 2000 timesteps
   cout << "# time t=" << t << ' ' << fsum << '\n';
   for(int i=1; i<=NF; i++){   // set ALL new f1 fields
     for(int j=1; j<=NF; j++){
       k=f[i-1][j-1]+f[i][j-1]+f[i+1][j-1]+
         f[i-1][j]            +f[i+1][j]   +
         f[i-1][j+1]+f[i][j+1]+f[i+1][j+1];
       if( k>3 )  f1[i][j]=0;
       if( k==3 ) f1[i][j]=1;
       if( k==2 ) f1[i][j]=f[i][j];
       if( k<2 )  f1[i][j]=0;
     }
   }
   fsum=0;
   for(int i=1; i<=NF; i++){
     for(int j=1; j<=NF; j++){ // write new values to f
       f[i][j]=f1[i][j];
       if( f[i][j] == 1 ) fsum++;
     }
   }
   for(int j=1; j<=NF; j++ ){
      f[0][j]   =f[NF][j];        // shadow bottom
```


Der Teil des Integrals über x_i ist dann:

$$I_i = \int_{a_i}^{b_i} f(x_i')dx_i' = (b_i - a_i) \int_0^1 f(x_i)\, dx_i \approx \frac{b_i - a_i}{N} \sum_{j=1}^{N} f_j \ , \tag{6.54}$$

wobei natürlich auch die Funktion $f(x_i')$ durch die transformierte Funktion $f(x_i)$ ersetzt werden muß. Als Beispiel wird $f(x_i') = x_i'^2 + 1$ durch $f(x_i) = [(b_i - a_i)x_i + a_i]^2 + 1$ ersetzt. Die Summe bezeichnet dabei N zufällige Funktionsauswertungen f_j an N Stellen $x_j \in [0, 1]$.

In D Dimensionen sieht ein Integral über einen Hyperkubus dann folgendermaßen aus:

$$\begin{aligned} I &= \int_{a_1}^{b_1} \int_{a_2}^{b_2} ... \int_{a_D}^{b_D} f(\mathbf{x})\, dx_1 dx_2 ... dx_D \\ &= \frac{(b_1 - a_1)(b_2 - a_2)...(b_D - a_D)}{N} \sum_{j=1}^{N} f(\mathbf{x}_j) \ . \end{aligned}$$

Hat man Funktionen, die stark schwanken oder nicht im gesamten Gebiet “glatt” sind (z. B. weil sie eine Singularität besitzen), so ist das sogenannte “Importance sampling”-Verfahren zu empfehlen. Dabei geht man wie bei der normalen Monte-Carlo-Methode vor, gewichtet allerdings die Verteilung der Zufallszahlen so, daß die “Unebenheiten” der Funktion berücksichtigt werden. Würde man das z.B. bei einer schmalen Gaußkurve nicht tun, so würden die hohen Werte unverhältnismäßig selten besucht, was einen starken Verlust an Genauigkeit zur Folge hätte. Man definiere eine Gewichtungsfunktion $w(x)$, die so normiert ist, daß $\int_0^1 w(x)\, dx = 1$ ist. Man kann das Integral nun umschreiben:

$$I = \int_0^1 w(x) \frac{f(x)}{w(x)}\, dx = \int_0^1 \frac{f(x(y))}{w(x(y))}\, dy = \frac{1}{N} \sum_{j=1}^{N} \frac{f(x(y_j))}{w(x(y_j))} \ , \tag{6.55}$$

wobei die Transformation $dy = w(x)dx$ mit $y(0) = 0$ und $y(1) = 1$ verwendet wurde. Diese Methode bedarf allerdings einer Funktion $w(x)$, die so integriert werden kann, daß man die Werte $x(y_j)$ einfach berechnen kann. Gleichzeitig sollte die Funktion $w(x)$ sich ähnlich verhalten wie $f(x)$, damit der Quotient $f(x)/w(x)$ im gesamten Integrationsbereich möglichst konstant bleibt.

Mit der Monte-Carlo-Methode sind damit auch die Integration von Funktionalintegralen, Pfadintegralen und Zustandssummen möglich, so daß sie ein breites Anwendungsgebiet in der Physik besitzt.

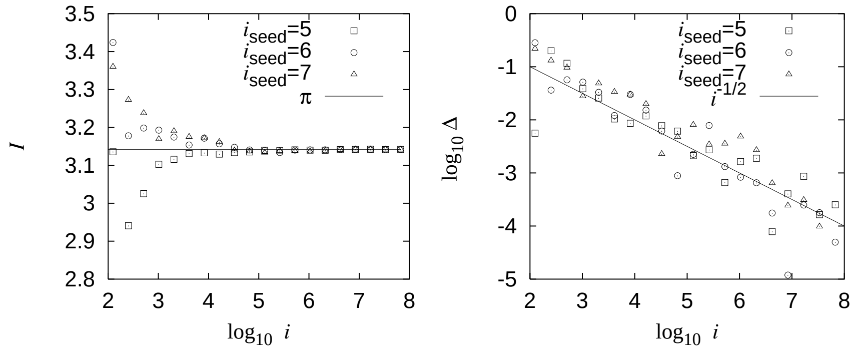


Abbildung 6.6: (Links) Näherungswert I für π, ermittelt mit dem Programm im Text für verschiedene i und drei verschiedene Folgen von Zufallszahlen. Zum Vergleich ist π als horizontale Line eingezeichnet. Die horizontale Achse ist logarithmisch. (Rechts) Fehler $\Delta = \sqrt{(I_i - \pi)^2}$ als Funktion der Iterationen i. Hier sind beide Achsen logarithmisch. Die durchgezogene Linie entspricht der Funktion $\sqrt{i}$.

was auch in Abbildung 6.6 zu sehen ist. Anders ausgedrückt muß man mit dieser Methode vier mal länger integrieren, um den statistischen Fehler auf die Hälfte zu reduzieren. Man beachte, daß Δ_i unabhängig von der Dimension D des Integrals ist.

Bei der Trapezregel findet man z.B. durch Aufteilung des Integrationsvolumens in N Hyperwürfelchen mit der Seitenlänge $h \propto N^{-(1/D)}$, in D Dimensionen, daß sich der Fehler wie $\Delta \propto N^{-(2/D)}$ verhält. In einer Dimension erzielt man also durch Intervallhalbierung eine Reduzierung des Fehlers um den Faktor vier! In Dimensionen $D > 4$ erzielt man jedoch mit der Monte-Carlo-Methode asymptotisch ein besseres Ergebnis! Entsprechende Aussagen mit jeweils anderen Dimensionszahlen lassen sich für alle Verfahren mit fester Fehlerordnung und mit äquidistanten Stützstellen finden.

6.4.2 Berechnung höherdimensionaler Integrale

Bei allgemeinen Funktionen $f(\mathbf{x})$ führt man am besten zunächst eine Variablensubstitution durch, um die Zufallszahlen direkt verwenden zu können, d.h. man ersetzt die Variable x'_i, über die im Intervall $[a_i, b_i]$ inegriert werden soll, durch eine Variable

$$x_i = \frac{x'_i - a_i}{b_i - a_i} \tag{6.53}$$


```
      f[NF+1][j]=f[1][j];           // shadow top
    }
    for(int i=1; i<=NF; i++ ){
      f[i][0]    =f[i][NF];         // shadow right
      f[i][NF+1]=f[i][1];           // shadow right
    }
    f[0][0]       =f[NF][NF];       // shadow bottom left
    f[NF+1][0]    =f[1][NF];        // shadow top left
    f[0][NF+1]    =f[NF][1];        // shadow bottom right
    f[NF+1][NF+1]=f[1][1];          // shadow top right
  }
  for(int i=1; i<=NF; i++ )         // Output data
    for(int j=1; j<=NF; j++ )
      if( f[i][j] == 1 ) cout << i << ' ' << j << '\n';
  return 0;
}
```

Je nach Anfangskonfiguration ergeben sich hierbei interessante Figuren. Einige Figuren sterben nach einer gewissen Zeit aus, da sie vereinsamen, siehe Abbildung 5.5 oben. Andere bleiben nach einer gewissen Zeit für jeden weiteren Zeitschritt stabil. Diese Figuren haben dann auch alle einen eigenen Namen bekommen, wie "Tube", "Block" und "Snake", in Abbildung 5.5 mitte. Weiterhin gibt es auch noch periodische Figuren, wie z.B. den "Blinker" mit der Periode zwei in Abb. 5.5 unten.

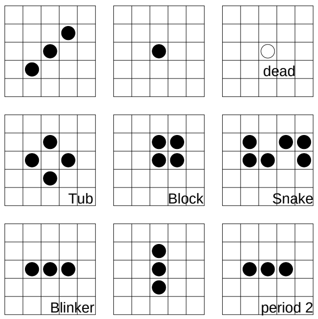


Abbildung 5.5: Aussterbende (oben), stabile (mitte) und periodische (unten) Figuren im Game of Life.

Eine besondere Figur ist der "Glider" (Gleiter). Diese Figur hat eine Periode von vier, d.h. nach vier Schritten hat sie wieder das vorherige Aussehen. Das Besondere dabei ist, daß der Gleiter dabei um eine Position weiter wandert. Bei der in Abb. 5.6 abgebildeten

Orientierung erfolgt die Wanderung nach rechts und unten gleichzeitig. Es gibt auch noch andere Gleiter, Pulsare und sogar Figuren, die Gleiter auf den Weg schicken, also eine Gleiter-Kanone (glider-gun).

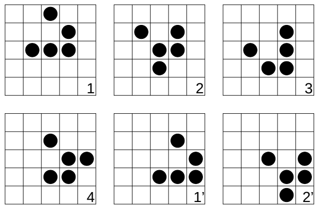


Abbildung 5.6: Die vier Phasen des Gleiters – verbunden mit einer Bewegung nach rechts unten.

5.4 Random walks (RW)

Eine Anwendung von Zufallszahlen sind sog. Zufallswege, engl. random walks (RW). Diese Methode kann z.B. für Brownsche Bewegungen, Molekulares Chaos und vor allem Diffusion benutzt werden. Bei einem Tintentropfen in Wasser, der sich langsam verteilt, ist es wegen der vielen Moleküle unmöglich, die Trajektorien (mit Molekulardynamik) zu berechnen. Hier hilft ein stochastischer Ansatz: ein typisches Molekül stößt mit anderen zusammen und ändert dadurch seine Richtung zufällig.

Das einfachste Modell ist ein diskretisierter eindimensionaler Raum (Gitter), wo ein Teilchen jeweils um eine Einheit und mit der gleichen Wahrscheinlichkeit nach links oder nach rechts springen kann. t sei hierbei der Zeitpunkt (diskret) und x_t der Ort des Teilchens zur Zeit t. Es gilt :

$$x_{t+1} = x_t + \begin{cases} 1 & \text{Wahrscheinlichkeit } p = 0.5 \\ -1 & \text{Wahrscheinlichkeit } p = 0.5 \end{cases} \tag{5.15}$$

Ein Programm zur Simulation mehrerer RW kann wie folgt aussehen:

```
int main()
{
  int xm=0, x2m=0, nstep=100, nwalk=300;
  int x1ave[nstep+1], x2ave[nstep+1], xn;

  for(int i=1; i<=nstep; i++ ){     // set fields to zero
```



```
   int iout=2, ic=iout;           // output delay
   int iprec=16;                  // precision
   int iseed=5;
   srand( iseed );                // set seed only once !

   for( int i=1; i<=100000000; i++ ){
      ic+=1;
      r0=float(rand())/romax;     // draw x, first random number
      r1=float(rand())/romax;     // draw y, second random number
                                  // check if inside the circle
      if( r1*r1+r0*r0 <= 1.E0 ) pint+=1;
      if( ic > iout ){
         cout << setprecision(iprec);
         cout << i << ' ' << M_PI << ' ' << 4.*double(pint)/i << '\n';
         ic=1;
         iout=iout*2;
      }
   }
   return 0;
}
```

Der Anteil der Punkte, die in den Kreis fallen, sollte gleich dem Anteil der gesuchten Fläche an dem umgebenden Quadrat sein. Für den oben betrachteten Viertel-Kreis mit Radius $R = 1$, sollte der Anteil gleich $I/4 = \pi/4 (\approx 0.785)$ sein. Das Programmbeispiel gibt sowohl π als auch den numerisch berechneten Näherungswert I aus.

Das Ergebnis I eines typischen Programmdurchlaufes ist in Abbildung 6.6 als Funktion der Iterationen i für verschiedene Zufallszahlen-Initialisierungen dargestellt.

Der Fehler der Monte-Carlo Methode ist rein statistisch; das Resultat I_i nach i Iterationen wurde als Mittelwert von i Versuchen mit Ergebnis f_j ermittelt als $I_i = (1/i)\sum_{j=1}^{i} f_j$. Die f_j sind um den Mittelwert $\bar{I} = \pi$ zufällig verteilt mit einer Standardabweichung von

$$\sigma_i = \sqrt{(1/i)\sum_{j=1}^{i}(f_j - \bar{I})^2} \; . \tag{6.50}$$

Der statistische Fehler nach i Iterationen ist dann

$$\Delta_i = \sigma_i/\sqrt{i-1} \; . \tag{6.51}$$

Nimmt man nun an, daß die Standardabweichung nicht von der Anzahl der Versuche i abhängt, wie es in der Regel für große i der Fall ist, so folgt sofort daß sich der Fehler verhält wie:

$$\Delta_i \propto i^{-1/2} \tag{6.52}$$

6.4.1 Berechnung der Zahl π

Das einfachste Anwendungsbeispiel ist die Berechnung der Fläche eines Kreises vom Radius $R = 1$, d.h. anders ausgedrückt, mit:

$$I = \int_{r \leq R} d\mathbf{r} = \pi \; , \tag{6.49}$$

die Berechnung von π. Hierzu werfen wir zufällig und homogen verteilt Punkte in ein Quadrat, welches das zu integrierende Gebiet (hier einen Kreis) enthält. Wenn der Punkt $\mathbf{r} = (x, y)$ innerhalb des Kreises liegt wird er zum Integral gezählt, liegt er außerhalb wird der Versuch verworfen.

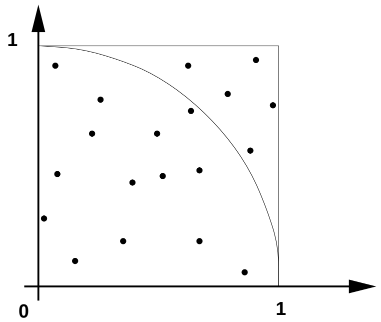


Abbildung 6.5: Geometrie zur Bestimmung von π durch Monte-Carlo Integration.

Der Zufallszahlengenerator sei `rand()`, der Werte im Intervall $[0, 1[$ liefert wenn man die `int`-Zufallszahl durch die Periode `RAND_MAX` teilt. Dann ist eine mögliche Implementierung:

```
#include<iostream>
#include<iomanip>
using std::cout;
using std::cin;
using std::setprecision;

const static long int romax=RAND_MAX;

int main()
{
   double r0, r1;
   int pint=0;                  // integration sum
```

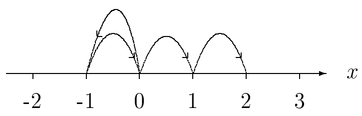



Abbildung 5.7: Ein-dimensionaler Zufallsweg: Start bei $x = 0$ in der Folge: l-r-r-r

```
    x1ave[i]=0;
    x2ave[i]=0;
  }

  for( int iw=1; iw<=nwalk; iw++ )  // perform nwalk random walks
  {
   rkps0( 234567+iw );              // initialize KPS RNG anew
   xn=0;                            // initial position is at zero
   for(int i=1; i<=nstep; i++ ){    // perform nstep steps
    if( rkps() ) xn++;              // step right
    else         xn--;              // step left
    x1ave[i] += xn;                 // averages
    x2ave[i] += xn*xn;
   }
  }
  for(int i=1; i<=nstep; i++ )      // output loop
   cout << i << ' ' << x1ave[i] << ' ' << x2ave[i] << '\n';

  return 0;
}
```

In Abbildung 5.8 (links) sind zwei verschiedene random walks dargestellt. Der Mittelwert bzw. das erste Moment der Position zur Zeit t ist:

$$\langle x_t \rangle = \frac{1}{N} \sum_{i=1}^{N} x_t(i) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{t} \gamma_{ij} \; , \tag{5.16}$$

wobei γ_{ij} die j-te Zufallsvariable im i-ten Lauf, mit $\gamma_{ij} = \pm 1$ ist. N ist die Anzahl der RW und $x_t(i)$ die Position des RW im i-ten Lauf zur Zeit t. Bildet man den Mittelwert aller Zufallswege, so ist $\langle x_t \rangle = 0$, da sich die Summe aller γ_{ij}, die ja mit gleicher Wahrscheinlichkeit -1 oder +1 sind, aufheben.

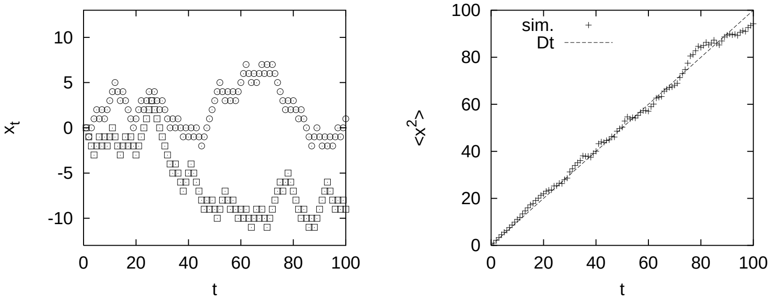


Abbildung 5.8: Position x_t von zwei verschiedenden random walks (links) und mittleres Verschiebungsquadrat $\langle x^2 \rangle$ im Mittel über 300 RW (rechts) als Funktion der Zeit.

Für die Bewegung im Raum ist der mittlere quadratische Abstand

$$\langle x_t^2 \rangle = \frac{1}{N} \sum_{n=1}^{N} x_t^2 \tag{5.17}$$

wichtig und vor allem aussagekräftiger.

In Abbildung 5.8 (rechts) ist dieser Abstand über die Zeit aufgetragen. Es ergibt sich eine Ursprungsgerade, es gilt also:

$$\langle x_t^2 \rangle = Dt \ , \tag{5.18}$$

mit der Diffusionskonstante D. Aus der Grafik ist ersichtlich, daß sie in diesem Fall $D \approx 1$ ist. Im Gegensatz zu ballistischen Teilchen (geradlinige, gleichförmige Bewegung), deren zurückgelegter Weg proportional zur Zeit ist ($r \sim t$), besitzen die RW-Teilchen ein diffusives Verhalten und ihr im Mittel zurückgelegter Weg ist proportional zur Wurzel aus der Zeit:

$$r \mathrel{\hat{=}} \sqrt{\langle x^2 \rangle} \ \sim \sqrt{t} \tag{5.19}$$

Auf den Tintentropfen bezogen, wäre die benötigte Zeit, um sich im Glas zu verteilen, bei doppelt so großem Glas viermal so hoch.

Zur analytischen Berechnung von D benutzen wir die Summendarstellung von linearen und quadratischem Abstand bei einem Lauf $N = 1$:

$$x_t = \sum_{i=1}^{t} \gamma_i \tag{5.20}$$

$$\Rightarrow x_t^2 = \left(\sum_{i=1}^{n} \gamma_i \right)^2 = \sum_{i,j=1}^{n} \gamma_i \gamma_j \tag{5.21}$$

$N = 2$	$x_{1,2} = \pm\frac{1}{\sqrt{3}}$	$w_{1,2} = 1$
$N = 3$	$x_{1,3} = \pm\sqrt{\frac{3}{5}}$ $x_2 = 0$	$w_{1,2} = \frac{5}{9}$ $w_0 = \frac{8}{9}$
$N = 4$	$x_{1,4} = \pm 0.8611363116$ $x_{2,3} = \pm 0.3399810436$	$w_{1,4} = 0.3478548451$ $w_{2,3} = 0.6521451549$
$N = 5$	$x_{1,5} = \pm 0.9061798459$ $x_{2,4} = \pm 0.5384693101$ $x_3 = 0$	$w_{1,5} = 0.2369268851$ $w_{2,4} = 0.4786286705$ $w_3 = 0.5688888889$

Tabelle 6.1: Stützstellen x_i und Gewichte w_i der Gaußschen Quadratur.

6.3.5 Weiterführende Literatur

[6.1] Hans Rudolf Schwarz. *Numerische Mathematik*. B. G. Teubner, Stuttgart, Germany, 1986.

[6.2] M. Abramovitz and I. Stegun. *Handbook of Mathematical Functions*. Dover, New York, 1972.

6.4 Monte-Carlo Integration

Der Name Monte-Carlo Integration kommt von dem Zufallscharakter der im Folgenden beschriebenen Methode, bzw. vom berühmten Casino in Monacco. Es handelt sich um ein Verfahren, bei dem die Zufallszahlen aus Abschnitt 5.1 Verwendung finden.

Der einfachste Fall eines höherdimensionalen Integrals ist:

$$\int_a^b \int_c^d f(x,y) dx dy = \int_{V^D} f(\mathbf{r}) d\mathbf{r} \ , \tag{6.48}$$

die Integration über ein Rechteck. Bei erwünschter Genauigkeit ist die Anzahl der Stützstellen z.B. N^D (d.h. $N = 100$ Stützstellen in einer Dimension, $D = 1$, 10^4 Stützstellen für $D = 2$ und 10^6 in 3D! Der numerische Aufwand wird also mit zunehmender Dimension enorm groß. Dies trifft umsomehr zu wenn der Rand des Integrationsbereichs kein Rechteck sondern ein beliebiger Körper ist. Für $D > 1$ existiert auch ein Gauß-Legendre-Verfahren, aber wenigstens für $D \geq 4$ ist die Monte-Carlo-Methode zu bevorzugen.

$$I = \underbrace{\int_{-1}^{1} q_{2N-1}(x)\,dx}_{(I)} = \underbrace{\int_{-1}^{1} q_{N-1}(x)\mathcal{P}_N(x)\,dx}_{(II)} + \underbrace{\int_{-1}^{1} r_{N-1}(x)\,dx}_{(III)} \tag{6.44}$$

Der Ausdruck (II) ist zufolge Gleichung (6.42) identisch Null:

$$\int_{-1}^{1} q_{N-1}(x)\mathcal{P}_N(x)\,dx \quad = \quad \sum_{i=1}^{N} w_i q_{N-1}(x_i)\mathcal{P}_N(x_i) = 0$$

Damit das auch für die diskrete Integrationsformel noch so ist, müssen die x_i die Nullstellen des Polynoms $\mathcal{P}_N(x_i) = 0$ sein. Diese Nullstellen sind in Werken wie Abramovitz and Stegun [6.2] aufgelistet oder lassen sich mit Standardprogrammen (Maple) bestimmen. Die x_i sind außerdem paarweise symmetrisch zu Null.

Hat man die Stützstellen x_i bestimmt, so kann man das sog. *Lagrange-Polynom*

$$L_k(x) = \Pi_{j=1,j\neq k}^{N}\left(\frac{x - x_j}{x_k - x_j}\right) \ , \tag{6.45}$$

vom Grad $N-1$ ansetzen, das die Bedingung $L_k(x_j) = \delta_{kj}$ erfüllt; d.h. das Lagrange-Polynom $L_k(x_j)$ verschwindet an allen Stützstellen x_j mit $j \neq k$, an der Stützstelle x_k hat es den Wert $L_k(x_k) = 1$.

Setzt man an die Stelle von $q_{2N-1}(x)$ eine der N Funktionen $L_k^2(x)$, so folgt aus den Gleichungen (6.39) und (6.44) das Gewicht:

$$w_k = \sum_{i=1}^{N} w_i L_k^2(x_i) = \int_{-1}^{1} L_k^2(x)dx > 0 \ , \tag{6.46}$$

welches mit bekannten Stützstellen analytisch zu lösen ist.

Zusammenfassend gilt für allgemeines $f(x)$:

$$I = \int_{-1}^{1} f(x)\,dx \approx \sum_{i=1}^{N} w_i f(x_i) \ , \tag{6.47}$$

wobei die x_i die Nullstellen des *Legendre-Polynoms* $\mathcal{P}_N(x)$ sind, und sich die Gewichte w_k aus Gleichung (6.46) als Integral über das quadrierte *Lagrange-Polynom* $L_k(x)$ ergeben. Das "Ungefähr Gleich" $\approx$-Zeichen wird zur Gleichheit, wenn $f(x)$ ein Polynom maximal $2N$-ten Grades ist.

In Tabelle 6.1 sind die Stützstellen und Gewichte bis zur Ordnung $N = 5$ nach Ref. [6.1] aufgeführt. Der Fall $N = 2$ entspricht der im vorigen Abschnitt durchgeführten Berechnung, siehe Gleichung (6.36).

Für $i \neq j$, ist $\gamma_i\gamma_j \pm 1$ und zwar wieder jeweils mit gleicher Wahrscheinlichkeit $p = 0.5$. Bei Addition löschen sich alle diese Terme gegenseitig aus, und es bleiben nur noch die Summanden mit $i = j$ übrig:

$$x_t^2 = \sum_{i=1}^{t} \gamma_i^2 = t \ , \ \text{da}\ \gamma_i^2 \quad = \quad 1 \tag{5.22}$$

$$\Rightarrow D = \frac{x_t^2}{t} \quad = \quad 1 \ . \tag{5.23}$$

Die ganze Bewegung kann auch dreidimensional realisiert werden. Dafür werden alle Berechnungen für x nochmals unabhängig für die beiden anderen Richtungen y und z durchgeführt. Die Summe der Quadrate wird zur quadratischen Entfernung vom Ursprung

$$r^2 = x^2 + y^2 + z^2 \tag{5.24}$$

Diese Entfernung ist immer noch proportional zur Wurzel der Zeit.

Eine andere Variationsmöglichkeit ist die Implementierung einer zufälligen Weglänge mit maximaler Sprungweite a. Anstatt einer dualen Zufallszahl für den random walk benutzen wir eine reelle Zufallszahl, beispielsweise zwischen $-a$ und a. Abbildung 5.9 (links) zeigt diesen random walk mit zufälliger Weglänge und $a = 2$. In der rechten Grafik ist das mittlere Quadrat der Entfernung aufgetragen und es ist zu erkennen, daß sie wieder proportional zur Zeit ist, aber diesmal mit kleinerem Faktor.

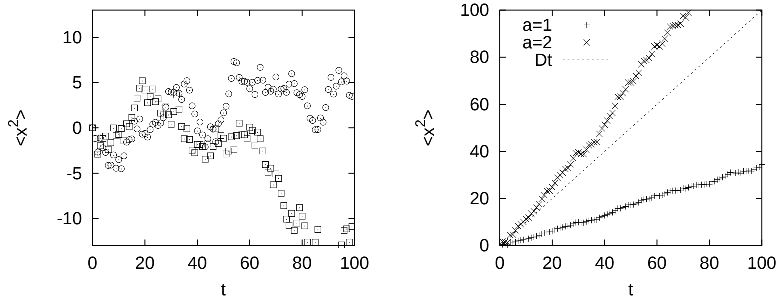


Abbildung 5.9: RW mit zufälliger Sprungweite und -richtung. Zwei Realisierungen (links) und $\langle x^2 \rangle$ im Mittel über 300 Läufe (rechts).

Trägt man zur Zeit t die Anzahl $N_t(x)$ der Teilchen (von verschiedenen RW) an einem Ort x auf, so entsteht eine Gaußsche Glockenkurve, siehe Abbildung 5.10. Das bedeutet daß $N_t(x)$ normalverteilt ist:

$$n_t(x) = \frac{N_t(x)}{\sum_x N_t(x)} = \frac{1}{\sqrt{\pi}\sigma} \mathrm{e}^{-\frac{(x-\mu)^2}{\sigma^2}} \ . \tag{5.25}$$

Hierbei ist der Mittelwert $\mu = \langle x_t \rangle = 0$. Für die Standardabweichung σ, also die Breite der Normalverteilung, gilt:

$$\sigma^2 = 2Dt = 2\langle x_t^2 \rangle \tag{5.26}$$

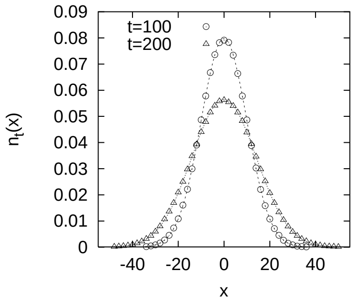


Abbildung 5.10: Gaußsche Glockenkurve der Verteilungen $2n_t(x)$. Symbole sind Daten der Simulation in Abb. 5.8, und die Linien stellen die entsprechenden Funktionen in Gleichung 5.25 dar.

Da die diskrete Simulation zur Gewinnung der Daten verwendet wurde, können bei geraden Zeiten t auch nur gerade Orte x besetzt sein. Zur Zeit $t = 0$ hat die Funktion $n_t(x)$ nur bei $x = 0$ den Wert $n_t(0) = 1$, überall sonst ist verschwindet die Funktion. Im Laufe der Zeit läuft die Funktion auseinander und wird flacher, die Fläche unter $n_t(x)$ beleibt durch die Definition erhalten.

Das mittlere Abstandquadrat eines RW hat also die Bedeutung der Breite der Verteilungsfunktion, die ebenso proportional zur Zeit ist:

$$\sigma \sim t \ . \tag{5.27}$$

Ein RW entfernt sich also mit der Zeit immer weiter von seinem Ausgangspunkt, kehrt jedoch immer wieder zurück, so daß die mittlere Position immer der Ausgangspunkt ist. Der RW unternimmt also immer längere Reisen zur Erkundung des Raumes.

Bemerkung:
Ein Legendre-Polynom $\mathcal{P}_l(x)$ von l-ter Ordnung besitzt l Nullstellen in $[-1, 1]$, siehe Ref. [6.1].

Wie oben soll das Integral einer Funktion $f(x)$ durch Auswertung an bestimmten Stützstellen x_i und Aufsummation mit Gewichten w_i approximiert werden:

$$I = \int_{-1}^{1} f(x)\,dx = \sum_{i=1}^{N} w_i f(x_i) \ . \tag{6.39}$$

Man hat dabei $2N$ Unbekannte: $w_i,\ x_i,\ (i = 1...N)$.

Ein allgemeines Lösungsverfahren für dieses Problems läßt sich folgendermaßen beschreiben: Die Gleichung sei exakt für ein beliebiges Polynom $f(x) = q_{2N-1}(x)$ der Ordnung $2N - 1$ und somit $2N$ freien Parametern. Man kann dies umschreiben zu:

$$q_{2N-1}(x) = q_{N-1}(x)\mathcal{P}_N(x) + r_{N-1}(x) \ . \tag{6.40}$$

Dabei ist P_N ein Legendre-Polynom von Grad N und q_{N-1} bzw. der Rest r_{N-1} sind Polynome der Ordnung $N - 1$. Diese Darstellung erhält man durch Polynomdivision von $q_{2N-1}(x)/\mathcal{P}_N(x)$. Es folgt nun aus den Eigenschaften der Legendre-Polynome:

1. Da sich jedes Polynom k-ter Ordnung durch eine Linearkombination von Legendre-Polynome bis zu einschließlich k-ter Ordnung darstellen läßt,

$$q_k(x) = \sum_{i=0}^{k} a_i \mathcal{P}_i(x) \ , \tag{6.41}$$

folgt direkt:

$$\int_{-1}^{1} \mathcal{P}_l(x) q_k(x)\,dx = \sum_{i=0}^{k} a_i \int_{-1}^{1} \mathcal{P}_l(x)\mathcal{P}_i(x)\,dx = 0 \ , \tag{6.42}$$

denn $i < l$, siehe Gleichung (6.37).

2. Weiterhin ist auch:

$$\int_{-1}^{1} 1 \cdot \mathcal{P}_l(x)\,dx = 0, \tag{6.43}$$

falls $l > 0$, siehe Gleichung (6.37).

3. Da $\mathcal{P}_l(x)$ im Intervall $[-1, 1]$ genau l Nullstellen hat (Beweis siehe z.B. Ref. [6.1]) folgt aus Glg. (6.40):

3. Zur Bestimmung dieser vier Unbekannten setzen wir in (6.31) $f(x) = 1,\ x,\ x^2$, und x^3 ein. Das heißt, wir benutzen vier Polynome verschiedenen Grades, um die vier Bestimmungsgleichungen zu erhalten (*beachte* : $a = -1$, $b = 1$):

$$(b-a) \quad = 2 = \quad w_1 + w_2 \tag{6.32}$$
$$\frac{1}{2}(b^2-a^2) \quad = 0 = \quad w_1 x_1 + w_2 x_2 \tag{6.33}$$
$$\frac{1}{3}(b^3-a^3) \quad = \tfrac{2}{3} = \quad w_1 x_1^2 + w_2 x_2^2 \tag{6.34}$$
$$\frac{1}{4}(b^4-a^4) \quad = 0 = \quad w_1 x_1^3 + w_2 x_2^3 \tag{6.35}$$

Einsetzen von (6.33) in (6.35) liefert $x_1^2 = x_2^2$ und damit $x_1 = -x_2$, wenn man fordert, daß die Stützstellen verschieden sein sollen. Unter Benutzung von (6.32), (6.33) und (6.34) ergibt sich dann:

$$w_1 = w_2 = 1 \quad \text{und :} \quad x_1^2 = \frac{1}{3} \Rightarrow x_{1,2} = \pm\frac{1}{\sqrt{3}} \tag{6.36}$$

Die $x_{1,2}$ sind die gesuchten Stützstellen – sie liegen symmetrisch zum Ursprung; jedes solches Stützstellenpaar besitzt auch nur ein zugehöriges Integrationsgewicht. Diese Symmetrie ist letztlich eine Folge der Tatsache, daß $\int_{-1}^{1} f(x)\,dx = \int_{-1}^{1} f(-x)\,dx$, siehe Glg. (6.31) für $f(x)$ und $f(-x)$.

6.3.4 Legendre-Polynome

Zurück zu den Polynomen $\mathcal{P}_l(x) = a_0 + a_1 x + ... + a_l x^l$ (Polynome l-ter Ordnung). Wir wollen jetzt fordern, daß diese im Intervall $[-1, 1]$ orthogonal zueinander sein sollen, d.h.:

$$\int_{-1}^{1} \mathcal{P}_k(x)\mathcal{P}_l(x)\,dx = \delta_{kl} \ . \tag{6.37}$$

Die einfachsten Polynome 1, x, x^2 und x^3 erfüllen diese Forderung jedoch nicht; stattdessen benötigt man die sog. *Legendre-Polynome*:

$$\frac{1}{\sqrt{2}},\ \sqrt{\frac{3}{2}}x,\ \sqrt{\frac{5}{2}}\frac{3x^2-1}{2}, ... \tag{6.38}$$

Die Frage ist: wie beschafft man sich die $\mathcal{P}_l(x)$?

Seien die $\mathcal{P}_k(x)$ mit $k \leq l-1$ bekannt, benutzt man den Ansatz $\mathcal{P}_l(x) = a_0 + a_1 x + ... + a_l x^l$. Die Orthogonalitätsbedingungen liefern zusammen mit der Normierungsforderung die Gleichung (6.37). Damit hat man $l+1$ Bestimmungsgleichungen für ebensoviele Koeffizienten $a_0 \ldots a_l$ ("Schmidtsches Orthogonalisierungsverfahren").

5.5 Random Walk und Diffusion

Wie schon erwähnt besteht ein enger Zusammenhang zwischen einem RW und der Diffusion. Die Verbindung zwischen beiden Prozessen soll kurz genauer erläutert werden.

Ausgehend von einem random walk kann die zeitliche Änderung der Aufenthaltswahrscheinlichkeit $p(x,t)$, den RW zur Zeit t am Ort x zu finden, durch die sog. *Master-Gleichung*

$$\Delta p(x, t+\Delta t) = p(x+\Delta x, t)W_- + p(x-\Delta x, t)W_+ - p(x,t)(W_+ + W_-) \tag{5.28}$$

ausgedrückt werden. Dabei sind W_+ und W_- jeweils die Wahrscheinlichkeiten, daß ein Teilchen in der Zeit Δt um die Entfernung $\pm\Delta x$ springt. Im Falle eines einfachen RW sind $W_+ = W_- = 1/2$.

Im Grenzfall $\Delta t \to 0$ und $\Delta x \to 0$ sind

$$\partial p/\partial t = \Delta p(x, t+\Delta t)/\Delta t \tag{5.29}$$

und

$$\partial^2 p/\partial x^2 = [p(x+\Delta x, t) - 2p(x+\Delta x, t) + p(x-\Delta x, t)]/\Delta x^2 \ , \tag{5.30}$$

woraus direkt die Diffusionsgleichung

$$\frac{\partial p}{\partial t} = \mathcal{D}\frac{\partial^2 p}{\partial x^2} \tag{5.31}$$

mit der Diffusionskonstante

$$\mathcal{D} = \frac{\Delta x^2}{2\Delta t} \tag{5.32}$$

folgt. Durch Einsetzen verifiziert man leicht die Lösung

$$p(x,t) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left[\frac{-(x-\mu)^2}{2\sigma^2}\right] \ , \tag{5.33}$$

mit

$$\sigma = \sqrt{2\mathcal{D}t} \ . \tag{5.34}$$

Der Unterschied zwischen Gleichung 5.33 und Gleichung 5.25 liegt nur in der unterschiedlichen Definition der Diffusionskonstanten $D = 2\mathcal{D}$, beide Gleichungen sind normiert, d.h. $\int p(x)dx = 1$. Die Definition in Glg. 5.33 ist allerdings leichter zu rekapitulieren, da sie genau so auf dem 10 DM Schein abgedruckt ist.

Anmerkungen

Das statistische Verhalten eines RW mit diskreter Zeit und diskretem Ort ist (bei hinreichend großer Schrittanzahl) gleich dem Verhalten eines RW mit kontinuierlicher Schrittweite, kontinuierlicher Wartezeitverteilung, oder beidem – solange die Diffusionskonstanten gleich sind.

Verbietet man die Orte, die ein RW bereits besucht hat, so erhält man einen self-avoiding RW (SAW), ein einfaches Modell für eine Polymerkette.

5.6 Fehler, Varianz und lineare Regression

Um den Fehlerbereich einer Meßreihe abzuschätzen, wird die Varianz (=Schwankung) der Meßreihe benutzt, die die Abweichung vom Mittelwert

$$\langle x\rangle_N = \frac{1}{N}\sum_{n=1}^{N} x_n \tag{5.35}$$

beschreibt. Für die Varianz σ gilt:

$$\begin{aligned}\sigma_N^2 &= \langle (x-\langle x\rangle_N)^2\rangle_N \\ &= \langle x^2 - 2x\langle x\rangle_N + \langle x\rangle_N^2\rangle_N \\ &= \langle x^2\rangle_N - 2\langle x\rangle_N\langle x\rangle_N + \langle x\rangle_N^2 \\ &= \langle x^2\rangle_N - \langle x\rangle_N^2\end{aligned} \tag{5.36}$$

5.6.1 Fehler

Der statistische Fehler des Mittelwerts wird typischerweise definiert als $\Delta_N = \sigma_N/\sqrt{N-1}$. Diese Wahl soll im Folgenden begründet werden. Geschickt ist eine Wahl, die unabhängig von der Meßprozedur ist. Wird dieselbe Meßreihe auf mehrere kleine Meßreihen aufgeteilt, so sollte der Fehler bei gleichen Werten auch konstant bleiben. Wir unterteilen unsere Meßreihe x_n mit N Punkten in k Einheiten der Länge m. Es gilt also $N = km$ und

$$x_n = x_{ij} \text{ mit } n = (i-1)m + j;\ i = 1,\cdots,k;\ j = 1,\cdots,m \tag{5.37}$$

Desweiteren haben die Untermeßreihen die Mittelwerte:

$$M_i = \frac{1}{m}\sum_{j=1}^{m} x_{ij} \tag{5.38}$$

Der gesamte Mittelwert ist dann:

$$\langle x\rangle_N = \langle M_i\rangle_k = \frac{1}{k}\sum_{i=1}^{k}\frac{1}{m}\sum_{j=1}^{m} x_{ij} \tag{5.39}$$

6.3.2 Quadratische Näherung

Analog ergibt sich für $L = 3$:

$$\begin{aligned}x_i - x_{i-1} &= w_0 + w_1 + w_2 \\ \frac{1}{2}(x_i^2 - x_{i-1}^2) &= w_0 x_{i-1} + w_1\bar{x}_i + w_2 x_i \\ \frac{1}{3}(x_i^3 - x_{i-1}^3) &= \frac{1}{6}\left[(x_i + x_{i-1})(x_i^2 - x_{i-1}^2) + (x_i - x_{i-1})(x_i^2 + x_{i-1}^2)\right] \\ &= w_0 x_{i-1}^2 + w_1\bar{x}_i^2 + w_2 x_i^2\end{aligned}$$

Lösung des Gleichungssystems ergibt:

$$\begin{aligned}w_0 = w_2 &= \frac{1}{6}(x_i - x_{i-1}) \\ w_1 &= \frac{4}{6}(x_i - x_{i-1})\end{aligned}$$

und somit die Simpsonregel, siehe Gleichung (6.20):

$$I_{x_{i-1},x_i} = (x_i - x_{i-1})\frac{1}{6}\left[f(x_i) + 4f(\bar{x}_i) + f(x_{i-1})\right] \ . \tag{6.29}$$

6.3.3 Beliebige Stützstellen

Bei dem Verfahren nach Gauß geht man ähnlich vor, allerdings werden neben den Gewichten w_i auch die Stützstellen x_i im Intervall $[a, b]$ als Parameter frei wählbar gelassen.

Die Vorgehensweise ist folgende:

1. Forme ein Integral über x' durch Substitution $x' = \frac{b+a}{2} + \frac{b-a}{2}y$ auf ein im Intervall $[-1, 1]$ definiertes, bestimmtes Integral um. Das Integral ist dann:

$$I_{ab} = \int_a^b f(x')\,dx' = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{b+a}{2} + \frac{b-a}{2}y\right)dy = \int_{-1}^{1} f(x)\,dx. \tag{6.30}$$

   Es reicht also aus, unsere Betrachtungen im Folgenden auf Integrale über das Intervall $[-1, 1]$ zu beschränken.

2. Betrachten wir nun eine fest vorgegebene Anzahl von Stützstellen, etwa $L = 2$, dann ist

$$I = I_{-1\ 1} = w_1 f(x_1) + w_2 f(x_2) \ , \tag{6.31}$$

   wobei sowohl die Stützstellen x_1 und x_2 als auch die Gewichte w_1 und w_2 als Unbekannte zu betrachten sind.

Nun sollen allgemein Funktionen im Intervall $[a,b]$ durch ein Polynom der Ordnung L approximiert werden,

$$f(x) = \sum_{l=0}^{L} \alpha_l \mathcal{P}_l(x) \ , \tag{6.22}$$

wobei die $\mathcal{P}_l(x)$ einen Satz linear unabhängiger Polynome der maximalen Ordnung l (höchste Potenz von x) bilden. Mit dieser Idee wollen wir nun zunächst alle Regeln im Intervall $[x_{i-1}, x_i]$ herleiten. Es sei

$$\begin{aligned} I_{ab} &= \sum_{i=1}^{N} I_{x_{i-1}\,x_i} && (6.23) \\ I = I_{x_{i-1}\,x_i} &= \sum_{k=0}^{L-1} w_i f_i \end{aligned}$$

Den Ausdruck (6.23) nennt man, wie eingangs erwähnt, "Quadratur" (Summe von Produkten). Die x_i nennt man *Integrationsstützstellen* mit den Funktionswerten $f_i = f(x_i)$ und die w_i sind die zugehörigen *Gewichte*.

6.3.1 Lineare Näherung

Für den einfachen Fall $L = 2$ ist also insbesondere

$$I = w_0 f_0 + w_1 f_1 \ . \tag{6.24}$$

Die Quadratur sei nun für die speziellen Funktionen $f(x) = 1$ und $f(x) = x$ exakt. Hieraus folgen zwei Gleichungen, aus denen sich die Gewichte w_i eindeutig bestimmen lassen:

$$\int_{x_{i-1}}^{x_i} 1\,dx = x_i - x_{i-1} \stackrel{!}{=} w_0 + w_1 \tag{6.25}$$

$$\int_{x_{i-1}}^{x_i} x\,dx = \frac{1}{2}\left(x_i^2 - x_{i-1}^2\right) \stackrel{!}{=} w_0 x_{i-1} + w_1 x_i \ . \tag{6.26}$$

Koeffizientenvergleich von (6.25) und (6.26) ergibt

$$w_0 = w_1 = \frac{x_i - x_{i-1}}{2} \ , \tag{6.27}$$

und somit das Integral über das Intervall:

$$I_{x_{i-1}\,x_i} = (x_i - x_{i-1}) \frac{f(x_i) + f(x_{i-1})}{2} \ , \tag{6.28}$$

also die bereits bekannte Trapezregel, siehe Gleichung (6.18).

Würde man die Varianz der M_i als Fehler nehmen, so bekäme man:

$$\begin{aligned} \sigma_k^2 &= \langle (M_i - \langle M_i \rangle_k)^2 \rangle_k \\ &= \tfrac{1}{k} \textstyle\sum_{i=1}^{k} (M_i - \langle x \rangle_N)^2 \\ &= \tfrac{1}{k} \textstyle\sum_{i=1}^{k} \tfrac{1}{m^2} \sum_{j,n=1}^{m} (x_{ij} - \langle x \rangle_N)(x_{in} - \langle x \rangle_N) \\ &= \tfrac{1}{k} \textstyle\sum_{i=1}^{k} \tfrac{1}{m^2} \sum_{j=1}^{m} (x_{ij} - \langle x \rangle_N)^2 \\ &= \tfrac{1}{m} \tfrac{1}{N} \textstyle\sum_{i=1}^{k} \sum_{i=1}^{m} (x_{ij} - \langle x \rangle_N)^2 \\ &= \tfrac{1}{m} \sigma_N^2 \end{aligned} \tag{5.40}$$

Die Vereinfachung der Doppelsumme gelingt durch die Auslöschung bei fluktuierenden Vorzeichen, wenn die Zufallsvariablen x_{ij} symmetrisch verteilt sind. Das Ergebnis zeigt, daß die Varianz von der Aufteilung der Meßreihe abhängt und somit nicht direkt für eine Fehlerberechnung verwendet werden darf. Aufteilungsunabhängig ist dagegen die Definition

$$\tilde{\Delta}_N = \frac{\sigma_N}{\sqrt{N}} \ , \tag{5.41}$$

wie durch die folgende Rechnung gezeigt wird:

$$\tilde{\Delta}_k^2 = \frac{\sigma_k^2}{k} = \frac{\sigma_N^2}{km} = \frac{\sigma_N^2}{N} = \tilde{\Delta}_N^2 \tag{5.42}$$

Die endgültige Definition für den Fehler lautet:

$$\Delta_N = \frac{\sigma}{\sqrt{N-1}} \ , \tag{5.43}$$

da bei $N = 1$ Messungen keine Aussage über den Fehler getroffen werden kann.

Sind die Meßpunkte x_n die Positionen eines RW, so kennt man die Varianz und damit den zu erwartenden Fehler in der Schätzung der mittleren Position des RW.

5.6.2 Lineare Regression

Gegeben seien N Datenpunkte (x_n, y_n) mit $n = 1, \cdots, N$, die in etwa auf einer Geraden liegen. Durch lineare Regression kann man die Steigung m und den Achsenabschnitt b der Geraden $y = mx + b$ bestimmen.

Parallel zur y-Achse ist der Abstand eines Punktes zu dieser Geraden

$$d_n = y_n - (m x_n + b) \ . \tag{5.44}$$

Gesucht ist nun jene Gerade, die die Summe der Abstandsquadrate

$$S = \frac{1}{N} \sum_{n=1}^{N} d_n^2 \tag{5.45}$$

minimiert. Sieht man die d_n als Meßwerte an, so muß die Gerade gefunden werden, die zur kleinsten Varianz der d_i führt. Das Minimum von S findet man indem man nach den beiden Unbekannten m und b ableitet und die Nullstellen der Ableitungen identifiziert. Die Ableitungen von S sind:

$$\frac{\partial S}{\partial m} = \frac{-2}{N}\sum_{n=1}^{N} x_n(y_n - mx_n - b) = 0 \tag{5.46}$$

und

$$\frac{\partial S}{\partial b} = \frac{-2}{N}\sum_{n=1}^{N}(y_n - mx_n - b) = 0\ . \tag{5.47}$$

Die Gleichungen 5.46 und 5.47 lassen sich unter Verwendung der Definitionen im vorigen Abschnitt vereinfachen:

$$\langle xy\rangle_N = m\langle x^2\rangle_N + b\langle x\rangle_N \tag{5.48}$$

und

$$b = \langle y\rangle_N - m\langle x\rangle_N\ . \tag{5.49}$$

Durch Einsetzen von Glg. 5.49 in Glg. 5.48 ergibt sich durch Auflösen nach m schließlich die Steigung (und damit b in Glg. 5.49) als Funktion der verschiedenen Mittelwerte

$$m = \frac{\langle xy\rangle_N - \langle x\rangle_N\langle y\rangle_N}{\langle x^2\rangle_N - \langle x\rangle_N^2}\ . \tag{5.50}$$

Vor der linearen Regression sollte amn allerdings eine Sichtprüfung der Daten durchführen, um festzustellen, ob die Daten wenigstens annähernd auf einer Geraden liegen. Ist dies nicht der Fall, so macht die “lineare” Regression keinen Sinn und man sollte ein anderes Anpassungsverfahren anwenden. Hier seien lediglich die “Spline-” oder die “Padé-Approximation” als alternative Verfahren erwähnt.

5.7 Weitere Anwendungsbeispiele

Nach Zellularautomaten und Random Walks sollen nun etwas interessantere Anwendungsmöglichkeiten von Zufallszahlen in der numerischen Physik vorgestellt werden. Perkolation, Epidemie-Modelle und Diffusionsbestimmte Aggregation sind die im Folgenden genauer diskutierten Beispiele.

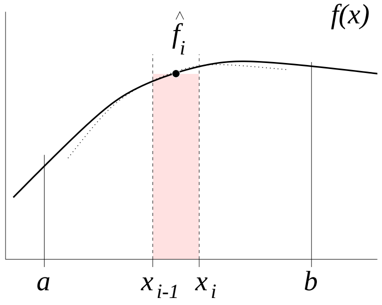


Abbildung 6.4: Schematische Darstellung eines Intervalls, dessen Fläche nach der Simpsonregel durch eine Parabel approximiert wird.

Unter Benutzung der zweiten Ableitung aus Glg. (6.9):

$$f''(\bar{x}_i) = \frac{f(x_i) - 2f(\bar{x}_i) + f(x_{i-1})}{(h/2)^2} = \frac{24}{h^2}\left(\hat{f}_i - f(\bar{x}_i)\right) \tag{6.21}$$

und der Taylorentwicklung von $f(x)$ aus Glg. (6.16), ergibt sich für den Fehler:

$$\begin{aligned}\Delta I_{ab} &= \sum_{i=1}^{N}\int_{x_{i-1}}^{x_i} f(x)\,dx - h\sum_{i=1}^{N}\hat{f}_i \\ &= h\sum_{i=1}^{N}\left[f(\bar{x}_i) + \frac{h^2}{24}f''(\bar{x}_i) + \underbrace{\frac{(h/2)^4}{120}f''''(\bar{x}_i)}_{\Delta I_{x_{i-1}x_i}}\right]\end{aligned}$$

Daraus folgt der Fehler $\Delta_{ab} = (b-a)(h/2)^4 f''''(\bar{x}_i)/120$, d.h. die Ordnung des Fehlers des Simpson-Verfahrens ist 4.

6.3 Gaußsche Quadratur

Bislang wurden die zu integrierenden Funktionen in den vorgegebenen Intervallen wie folgt approximiert:

- Trapezregel: Eine Gerade, d.h. linear
- Simpsonregel: Eine Parabel, d.h. quadratisch

6.2.2 Die Trapezregel

Das Trapez, das von den Funktionswerten an den Aufpunkten x_i und x_{i-1} aufgespannt wird hat die Fläche $(f_i + f_{i-1})h_i/2$, womit man ein Integral der Form

$$I_{ab} = \frac{1}{2}\sum_{i=1}^{N} h_i(f_i + f_{i-1}) \tag{6.18}$$

definieren kann. Diese Methode ergibt allerdings ebenfalls einen Fehler der Ordnung h^2. Die Trapezregel ist in Abb. 6.3 gezeichnet.

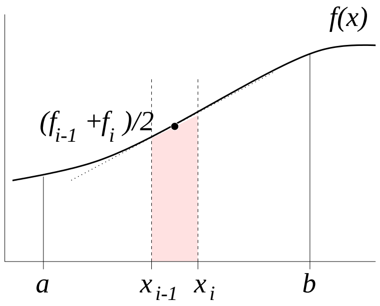


Abbildung 6.3: Schematische Darstellung eines Intervalls, dessen Fläche nach der Trapezregel durch eine Gerade approximiert wird.

6.2.3 Die Simpsonregel

Nun sei ein gewichteter Funktionsmittelwert gegeben als:

$$\hat{f}_i = \frac{1}{6}\left[f(x_i) + 4f\left(\frac{x_i + x_{i-1}}{2}\right) + f(x_{i-1})\right] , \tag{6.19}$$

bzw. die Kurve wird durch eine Parabel angenähert. Diese ist in Abb. 6.4 als gepunktete Linie zu sehen.

Weiterhin kann man zur Vereinfachung konstante Intervallbreiten $h = x_i - x_{i-1}$ annehmen. Das Integral I_{ab} berechnet sich somit zu:

$$I_{ab} = \frac{1}{6}\sum_{i=1}^{N}(x_i - x_{i-1})\left[f(x_i) + 4f(\bar{x}_i) + f(x_{i-1})\right] = h\sum_{i=1}^{N}\hat{f}_i \, . \tag{6.20}$$

5.7.1 Perkolation

Bei Perkolationsmodellen wird auf einem Gitter jeder Gitterplatz mit einer Wahrscheinlichkeit p besetzt oder bleibt dementsprechend mit der Wahrscheinlichkeit $1 - p$ frei. Die besetzten und nichtbesetzten Gitterplätze können nun zum Beispiel für Wasser und Fels oder für Leiter und Isolator in einem Stoffgemisch stehen. Verbindet man alle aktiven Gitterplätze, die auch nächste Nachbarn sind, miteinander, so erhält man große "Cluster". Je größer p gewählt wurde, umso größere Cluster kann man erwarten. Cluster können dabei Hohlräume (mit Wasser) oder Festkörper, bzw. Leiter oder Nichtleiter sein. Die mit solchen Modellen untersuchten Fragen könnten dann lauten: (i) Wieviel Prozent meines porösen Materials muß Hohlraum sein, damit Wasser fließen kann oder (ii) wieviel Leiteranteil muß sich in der Probe befinden, damit Strom fließt.

Mit anderen Worten wird nach der kritischen Perkolationswahrscheinlichkeit p_c gefragt, bei der ein Cluster das gesamte System ausfüllt. Dieser sog. Perkolationsschwellwert p_c grenzt die Bereiche voneinander ab in denen kein Cluster vom einen Ende des Systems zum anderen reicht ($p < p_c$) und in denen ein Cluster durch das gesamte System läuft ($p > p_c$). Für Quadratgitter ist $p_c = 0.592745$ numerisch berechnet worden, für andere Gitter, wie z.B. ein Dreiecksgitter findet man mit $p_c = 0.5$ andere Schwellwerte.

In Abbildung 5.11 sind Perkolationsgitter mit den Wahrscheinlichkeiten $p = 0.4$, 0.5 und 0.6 dargestellt. Besetzte Gitterplätze sind als Kreise eingezeichnet. Nächste Nachbarn werden durch sog. "Bonds" verbunden, welche als Linien eingezeichnet sind. Alle direkt verbundenen, besetzten Gitterplätze bilden sog. Cluster, und das größte Cluster ist durch schwarze Punkte gekennzeichnet. Unterhalb der kritischen Besetzungswahrscheinlichkeit findet man kein Cluster, welches das ganze System von oben bis unten durchläuft. Bei $p = 0.6$ existiert ein solches Cluster.

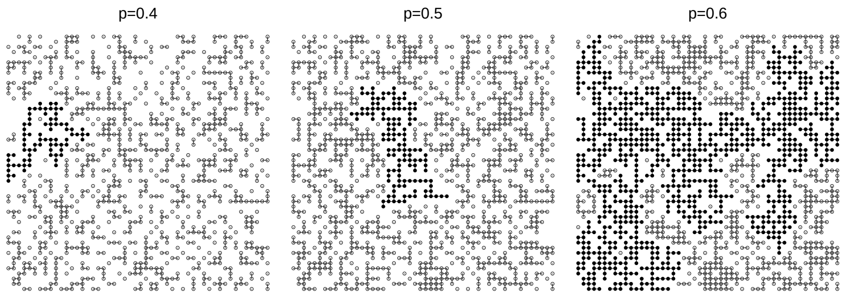


Abbildung 5.11: Perkolation mit verschiedenen Besetzungswahrscheinlichkeiten p. Besetzte Plätze sind als offene Kreise dargestellt, Verbindungen als Linien und die zum größten Cluster gehörenden besetzten Plätze als gefüllte Kreise.

Zur Simulation eines Perkolationsgitters werden zuerst die Gitterplätze zufällig mit der Wahrscheinlichkeit p besetzt. Im folgenden Programmsegment wird ein Gitter mit Seitenlänge 50 erzeugt und belegt. Dazu ist nur das Feld `world` nötig, die anderen Felder werden später zur Cluster-Identifizierung verwendet.

```
#include<iostream>
#include<fstream>
using std::cout;
using std::cin;
#include"rng2.h"

int main()
{
  const int NL=50;
  int world[NL+2][NL+2], sitemax=0;    // define the lattice
  int bonds[NL+1][NL+1][2], bondmax=0; // bonds field
  int icl[NL*NL+1], jcl[NL*NL+1];      // coordinates of site i
  int cnext[NL*NL+1], cfirst[NL*NL+1], slast, snext;  // pointers
  int clusterX, clusterA, clusterB, cluster[NL*NL+1]; // clusters

  float prob;
  cout << "# Perkolationswahrscheinlichkeit ? ";
  cin >> prob;   cout << prob << '\n';

  rng0(12345678);                      // initialize rng()

  ofstream outfile1("sites.dat");
  for(int i=1; i<=NL; i++ ){           // occupy sites
    for(int j=1; j<=NL; j++ ){
      if( double(rng())/dp > prob )
        world[i][j]=0;                 // empty
      else{
        sitemax += 1;                  // occupied
        world[i][j] = sitemax;         // every particle is first
        icl[sitemax]=i;                //  one cluster of size 1
        jcl[sitemax]=j;
        cluster[sitemax] = sitemax;
        cnext[sitemax] = 0;
        cfirst[sitemax] = sitemax;
        outfile1 << i << ' ' << j << ' ' << world[i][j] << '\n';
      }
    }
  }
  outfile1.close();
  cout << "# Created " << sitemax << " occupied sites \n";
```

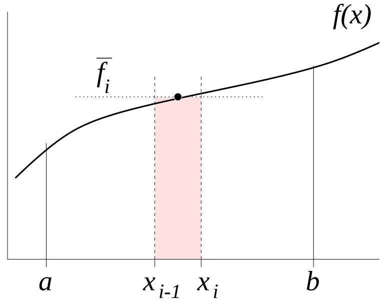



Abbildung 6.2: Schematische Darstellung eines Intervalls, dessen Fläche nach der Mittelpunktsregel durch den Funktionswert im Zentrum des Intervalls approximiert wird.

Damit ist I_{ab} im Fall gleichgroßer Intervalle:

$$I_{ab} = h \sum_{i=1}^{N} \bar{f}_i \ , \tag{6.13}$$

mit $h = h_i = (b-a)/N$, oder bei allgemeinen Intervall-Längen:

$$I_{ab} = \sum_{i=1}^{N} (x_i - x_{i-1}) \ f\left(\frac{x_i + x_{i-1}}{2}\right) = \sum_{i=1}^{N} h_i \bar{f}_i \ . \tag{6.14}$$

Der Fehler einer Integration ist somit die Differenz zwischen der exakten und der genäherten Lösung:

$$\Delta I_{ab} = \int_a^b f(x)\,dx - h \sum_{i=1}^{N} f(\bar{x}_i) \ , \tag{6.15}$$

wie sich durch Taylorentwicklung von $f(x)$ in kleinem Abstand δ um $\bar{x}_i$:

$$f(x) \quad = \quad f(\bar{x}_i) + \delta f'(\bar{x}_i) + \frac{1}{2}\delta^2 f''(\bar{x}_i) + ... \tag{6.16}$$

zeigen läßt. Setzt man Glg. (6.16) in Glg. (6.15) ein und integriert, jeweils für ein Intervall, von $-h/2$ bis $h/2$ über δ, so sieht man, daß der erste Summand der Taylorreihe mit der Summe in Gleichung (6.13) identisch ist. Der zweite Summand verschwindet, da bei symmetrischer Integration eine zu $\bar{x}_i$ symmetrische Funktion (hier: alle δ mit ungerader Potenz) verschwindet. Damit errechnet sich der Fehler aus Glg. (6.15) zu:

$$\Delta I_{ab} = \frac{1}{2} f''(\bar{x}_i) \int_{-h/2}^{h/2} \delta^2 \, d\delta = \frac{h^2}{24} f''(\bar{x}_i) \ , \tag{6.17}$$

d.h. der Fehler der Mittelpunktsregel ist von 2. Ordnung in h.

Funktion kann in geschlossener Form integriert werden, so z.B. die Fehlerfunktion $\mathrm{erf}(x)$, die Gamma-Funktion $\Gamma(x)$ oder die Besselfunktionen. Ebenso wie bei der Differentiation, ist auch die Integration einer Funktion, von der nur Datenpunkte bekannt sind, möglich.

Als ein Anwendungsbeispiel läßt sich die Teilchengrößenverteilung in einem Korngemisch (z.B. Erdreich) nennen, die man durch Sortieren des Ausgangsmaterials mit Sieben verschiedener Maschenweite erhält, Abb. 6.1. Diese Funktion $p(x)$ gibt an, wie groß der Anteil von Körnern des Durchmessers x am Gesamtgemisch ist. Genauer gesagt, ist $p(x)\Delta x$ die Wahrscheinlichkeit, daß ein zufällig ausgewähltes Korn einen Durchmesser im Bereich zwischen x und $x + \Delta x$ aufweist. Die Verteilungsfunktion $P_S = P(x < S)$ gibt an, wieviele Teilchen kleiner als eine bestimmte Größe S sind. Die Funktion P_S erhält man durch numerische Integration aus $p(x)$.

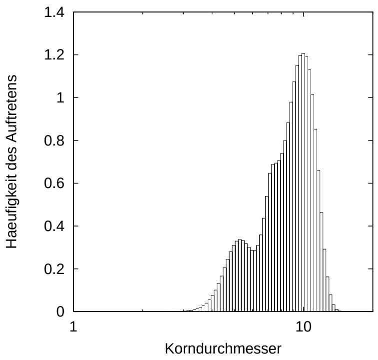


Abbildung 6.1: Beispiel für eine Teilchengrößenverteilung. Aufgetragen ist die Häufigkeit, mit der ein bestimmter Durchmesser vorkommt (in willkürlichen Einheiten), gegen den Durchmesser des Korns (mm).

Im allgemeinen besteht die Aufgabe also darin, $I_{ab} = \int_a^b f(x)\,dx$ numerisch zu berechnen. Drei der einfachsten Integrationsverfahren werden im Folgenden genauer diskutiert. Die Mittelpunktsregel und die Trapezregel

6.2.1 Die Mittelpunktsregel

Im Folgenden sei der Integrationsbereich von a bis b in N Intervalle mit Randpunkten $x_i = a + ih$ $(i = 0, ..., N)$, und Breite $h_i = x_i - x_{i-1}$ unterteilt. Weiterhin bedeutet die Schreibweise $\bar{f}_i = f(\bar{x}_i)$ eine Funktionsauswertung am Punkt $\bar{x}_i = (x_i + x_{i-1})/2$, in der Mitte des Intervalls zwischen x_{i-1} und x_i, sofern die analytische Form von f bekannt ist. Diese Situation ist in Abb. 6.2 zu sehen.


```
clusterX = sitemax;
```

Hat man alle Gitterplätze durchlaufen, kann man noch die nächsten Nachbarn miteinander verbinden. Dieses Programmsegment wurde für die Darstellung der Linien in Abb. 5.11 benötigt.

```
ofstream outfile2("bonds.dat");
for(int i=1; i<=NL; i++ ){            // occupy bonds
  for(int j=1; j<=NL; j++ ){
    bonds[i][j][0]=0;
    bonds[i][j][1]=0;
    if( world[i][j] >= 1 ){
      if( world[i-1][j] >= 1 ){      // left neighbor
        bondmax += 1;
        bonds[i][j][0]=bondmax;
        outfile2 << i << ' ' << j << '\n';
        outfile2 << i-1 << ' ' << j << '\n' << '\n';
      }
      if( world[i][j-1] >= 1 ){      // lower neighbor
        bondmax += 1;
        bonds[i][j][1]=bondmax;
        outfile2 << i << ' ' << j << '\n';
        outfile2 << i << ' ' << j-1 << '\n' << '\n';
      }
    }
  }
}
outfile2.close();
cout << "# Created " << bondmax << " interconnections \n";
```

Zur Erkennung von Perkolation kann man den sog. “burning”-Algorithmus (siehe Übung), oder ein ähnliches Verfahren verwenden. Hier soll jedoch ein Cluster-Erkennungs Algorithmus vorgestellt werden. Jeder besetzte Gitterplatz wird am Anfang als eigenständiges Cluster angesehen und durchnummeriert. Danach werden alle nächsten Nachbarn (NN) verglichen. Falls zwei NN bereits zum gleichen Cluster gehören wird nichts unternommen. Gehören sie dagegen zu verschiedenen Clustern, so wird Cluster B an Cluster A angehängt und alle darin enthaltenen Teilchen dementsprechend als zum Cluster A zugehörig markiert. Der letzte Cluster X wird nun nach dem freigewordenen Cluster B umbenannt und die Clusteranzahl wird um Eins erniedrigt. Danach fährt man mit dem Vergleich der nächsten Nachbarn fort.

```
// merge bonded sites to clusters N*N
for(int i=1; i<=NL; i++ ){
```

```
for(int j=1; j<=NL; j++ ){
                             // act only if site (i,j) is occupied ...
  if( world[i][j] >= 1 ){
    clusterA = cluster[ world[i][j] ];
    for( int k=1; k<=2; k++ ){
      if( k == 1 ) snext = world[i-1][j];      // left neighbor
      if( k == 2 ) snext = world[i][j-1];      // lower neighbor
      if( snext >= 1 ){
                                  // act only if bonded sites belong
                                  //   to different clusters
        clusterB = cluster[ snext ];
        if( clusterA != clusterB ){
                                  // start with the first site (i,j)
          if( clusterA == clusterX ){
            clusterA = clusterB;
            clusterB = clusterX;
            slast = snext;
          }
          else
            slast = world[i][j];
          snext = cnext[ slast ];
                                  // seek for the last bond that belongs
                                  //   to the same cluster as (i,j)
          while( snext != 0 ){
            slast = snext;
            snext = cnext[ slast ];
          }
          snext = cfirst[ clusterB ];
                                  // change cluster number of clusterB
          cnext[ slast ] = snext;
          while( snext != 0 ){
            cluster[ snext ] = clusterA;
            snext = cnext[ snext ];
          }
          if( clusterB != clusterX ){
            cfirst[ clusterB ] = cfirst[ clusterX ];
                                 // replace clusterB by the last clusterX
            snext = cfirst[ clusterB ];
            while( snext != 0 ){
              cluster[ snext ] = clusterB;
              snext = cnext[ snext ];
            }
          }
          clusterX -= 1;           // one cluster less
        }
```


6.1.2 Die zweite Ableitung

Für höhere Ableitungen gibt es ähnliche Entwicklungen wie oben gezeigt; diese sind manchmal zur Lösung von Differentialgleichungen nötig:

$$\begin{aligned} a(t) &= \dot{v}(t) = \ddot{x}(t) \\ a_i &= \frac{\frac{x_{i+1}-x_i}{h} - \frac{x_i - x_{i-1}}{h}}{h} = \frac{x_{i+1} - 2x_i + x_{i-1}}{h^2} \; . \end{aligned} \tag{6.9}$$

Der Quotient in Gleichung (6.9) kann durch eine Vorwärts- mit darauffolgender Rückwärts-Ableitung berechnet werden. Die auftretenden Restterme in Gleichung (6.9) sind von Ordnung h^2.

Eine Formel für die zweite Ableitung mit Fehler 4. Ordnung in h ist z.B.

$$a_i = \frac{1}{12h^2}(-x_{i+2} + 16x_{i+1} - 30x_i + 16x_{i-1} - x_{i-2}) \; . \tag{6.10}$$

6.1.3 Numerische Probleme

Neben der Tatsache, daß eine Berechnung mit höherer Genauigkeit auch mehr Aufwand (Funktionsberechnungen) erfordert, hat die numerische Berechnung von Ableitungen das prinzipielle Problem der sog. Stellenauslöschung: Bei endlicher Darstellungsgenauigkeit (`doubles` haben etwa 16, `floats` nur 7 Stellen Genauigkeit) erhält man bei Differenzenbildung einen Verlust an Genauigkeit, wenn Zahlen in der gleichen Größenordnung voneinander subtrahiert werden. Zum Beispiel verliert man in der folgenden Rechnung

$$\frac{0.264345 - 0.259123}{0.129001 - 0.121112} \approx 0.66193 \tag{6.11}$$

2 Stellen Genauigkeit, wenn man nur mit 3 Stellen in der Mantisse arbeitet:

$$\frac{0.264 - 0.259}{0.129 - 0.121} = \frac{0.005}{0.008} = 0.625. \tag{6.12}$$

6.2 Quadraturen

Problem des Anaxagoras: Quadratur des Kreises

Dies ist analog zu dem Problem der Flächenberechnung des Kreises als Summe von Quadraten (=Produkt), das 1882 als unmöglich bewiesen wurde.

Allgemein wird der Wert eines Integrals als eine Summe von Produkten numerisch berechnet. Die numerische Integration ist aus verschiedenen Gründen wichtig: Nicht jede

(es wird der *"rückwärts"* nächstliegende Punkt $x(t_{i-1})$ zu Hilfe genommen)

3. "zentrales Ableiten":

$$v_i^z = \frac{x_{i+1} - x_{i-1}}{t_{i+1} - t_{i-1}} \tag{6.3}$$

(die beiden Punkte *vor und nach* $x(t_i)$ werden benutzt)

Frage:
Welche dieser drei Möglichkeiten gibt den besten bzw. genauesten Wert der Ableitung?

Mit $h := t_{i+1} - t_i \; \forall \, i$ ist die *Taylorentwicklung* für $x(t \pm h)$:

$$x(t \pm h) = \underbrace{x(t)}_{0.} \pm \underbrace{hv(t)}_{1.} + \underbrace{\frac{h^2}{2}\dot{v}(t)}_{2.} \pm \underbrace{\frac{h^3}{6}\ddot{v}(t)}_{3.\text{Ordnung}} + ... \tag{6.4}$$

Damit erhält man für die obigen Ableitungsformeln (6.1)-(6.3)

$$v^v(t) = \frac{x(t+h) - x(t)}{h} = v(t) + \frac{h}{2}\dot{v}(t) + ... = v(t) + O(h^1) \; , \tag{6.5}$$

$$v^r(t) = \frac{x(t) - x(t-h)}{h} = v(t) + \frac{h}{2}\dot{v}(t) + ... = v(t) + O(h^1) \tag{6.6}$$

und

$$v^z(t) = \frac{x(t+h) - x(t-h)}{2h} = v(t) + \frac{2h^2}{3}\ddot{v} + ... = v(t) + O(h^2) \tag{6.7}$$

Der Fehler von $v^v(t)$ und $v^r(t)$ ist also von 1. Ordnung, während der Fehler von $v^z(t)$ nur von 2. Ordnung in h ist, d.h. $v^z(t)$ stellt die "bessere" Ableitung dar.

Die numerische Ableitung kann systematisch weiter verbessert werden: Eine Formel mit Genauigkeit höherer Ordnung wäre z.B.:

$$v_i^{sz} = \frac{1}{12h}(-x_{i+2} + 8x_{i+1} - 8_{i-1} + x_{i-2}) \tag{6.8}$$

Aufgabe: Zeigen Sie durch Taylorentwicklung von $x(t \pm 2h)$ und $x(t \pm h)$, daß v_i^{sz} aus (6.8) die Ableitung bis auf Terme der Ordnung h^4 richtig wiedergibt!


```
        }   // end of world[i-1][j] or world[i][j-1] loop ...
      }
    }       // end of world[i][j] loop ...
  }
}
```

Nach dem Durchlauf des oben beschriebenen Verfahrens bleiben nun `clusterX` Cluster übrig. Sinnvollerweise sucht man den größten der Cluster und prüft ob er durch das gesamte System läuft, also z.B. von oben nach unten reicht.

```
int csize, stotal=0, maxcsize=0, maxnum=0;

ofstream outfile3("cluster.dat");
for( int i=1; i<=clusterX; i++ ){
  outfile3 << i << "   ";
  snext = cfirst[ i ];            // select first site of cluster i
  csize=0;
  while( snext != 0 ){
     outfile3 << snext << ' ';
     snext = cnext[ snext ];      // hop to next site
     csize += 1;
     stotal += 1;                 // test: sum up all occupied sites
  }
  if( csize > maxcsize ){
     maxcsize=csize;              // set maximum size
     maxnum=i;
  }
  outfile3 << '\n';
}
outfile3.close();

int bottom = 0;                   // check if max_cluster hits bottom
int top = 0;                      //              ...          top

ofstream outfile4("mcl.dat");
snext = cfirst[ maxcnum ];
while( snext != 0 ){
                                  // check if max_cluster spans system
  if( icl[snext] == 1  ) bottom = 1;
  if( icl[snext] == NL ) top = 1;
  outfile4 << snext << ' ' << icl[snext] << ' ' << jcl[snext] << '\n';
  snext = cnext[ snext ];
}
outfile4.close();
```

```
  cout << "# Max Cluster Size " << maxcsize << "\n";
  if(( bottom == 1 ) && ( top == 1 ))
    cout << "# Cluster percolates - " << stotal
         << " total sites occupied \n";
  else
    cout << "# Cluster does not percolate - " << stotal
         << " total sites occupied \n";

  return 0;
}
```

Um die Perkolationswahrscheinlichkeit zu bestimmen führt man bei gegebenem p viele Simulationen durch. Die Perkolationswahrscheinlichkeit gewonnen aus 100 Simulationen ist in Abbildung 5.12 als Funktion der Besetzungswahrscheinlichkeit p aufgetragen. Man erkennt daß der Übergang mit zunehmender Systemgröße immer schärfer wird. Im rechten Teil der Abbildung ist die Clustergrößenverteilung für verschiedene p dargestellt. Kleine Cluster werden mit zunehmendem p immer unwahrscheinlicher, da sich alle belegten Gitterplätze im größten Cluster treffen.

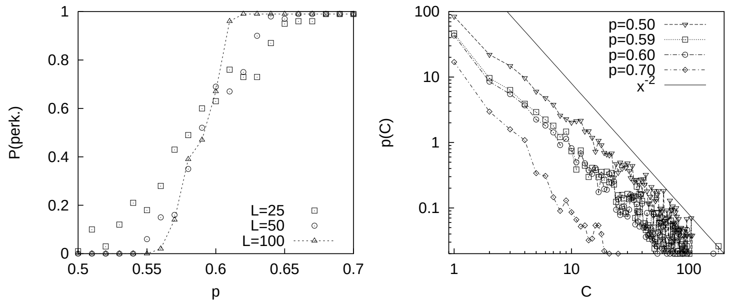


Abbildung 5.12: Perkolationswahrscheinlichkeit als Funktion der Besetzungswahrscheinlichkeit p für verschiedene Systemgrößen L, also für $N = L^2$ Gitterplätze (Links). Clustergrößenverteilung für verschiedene p (Rechts).

Ein Beispiel für die erfolgreiche Anwendung der Perkolation als Modell ist die Herstellung von Gelatine, dem Übergang von einem Sol zum Gel. Die Moleküle in einem Sol wachsen zusammen und verzweigen sich. Durch Polymerisation bilden sich lange Molekülketten, bis irgendwann das Gel entsteht. Dieser Zeitpunkt ist identisch mit der systemüberspannenden Perkolation bei $p = p_c$. Im Gel ist immer noch sehr viel Flüssigkeit enthalten, der große Cluster gibt dem ganzen Objekt jedoch eine Struktur und Scherfestigkeit – im Gegensatz zu einer Flüssigkeit.

Kapitel 6

Numerisches Differenzieren und Integrieren

6.1 Differenzieren

Ein Anwendungsbeispiel für numerische Differentiation ist die Bestimmung der Momentangeschwindigkeit $v = \dot{x}(t) = \lim_{\Delta t \to 0} \frac{x(t+\Delta t) - x(t)}{\Delta t}$. Außer im Fall einer im gesamten Wertebereich bekannten Funktion $x(t)$, ist der Grenzfall $\lim_{\Delta t \to 0}$ allerdings nicht zu erreichen - man hat häufig nur diskrete Datenpunkte zur Verfügung.

6.1.1 Die erste Ableitung

Die diskrete, mittlere Geschwindigkeit kann man durch Messung des zurückgelegten Weges x_i zu bestimmten Zeitpunkten t_i bestimmen (wobei $i = 1, ..., N$). Man erhält eine Tabelle mit N Datenpunkten, aus denen die Steigung $\dot{x}(t)$ der Kurve $x(t)$ bestimmt werden kann. Mit $x_i = x(t_i)$ und $v_i = v(t_i)$ kann man die folgenden drei einfachsten Möglichkeiten definieren:

1. “vorwärts Ableiten”:

$$v_i^v = \frac{x_{i+1} - x_i}{t_{i+1} - t_i} \tag{6.1}$$

(es wird der *“vorwärts”* nächstliegende Punkt $x(t_{i+1})$ zu Hilfe genommen)

2. “rückwärts Ableiten”:

$$v_i^r = \frac{x_i - x_{i-1}}{t_i - t_{i-1}} \tag{6.2}$$

```
                 if(i.ne.1) a(i,j)=a(i,j)+a(i-ns,j)
                 if(i.ne.ni2+1) a(i,j)=a(i,j)+a(i+ns,j)
                 a(i,j)=a(i,j)/2.+(rand()-pn)*ns
                 if(a(i,j).gt.amax) amax=a(i,j)
              else
                 if(j.ne.1) a(i,j)=a(i,j)+a(i,j-ns)
                 if(j.ne.ni2+1) a(i,j)=a(i,j)+a(i,j+ns)
                 a(i,j)=a(i,j)/2.+(rand()-pn)*ns
                 if(a(i,j).gt.amax) amax=a(i,j)
              endif
           endif
           if(l2) then
              l2=.false.
           else
              l2=.true.
           endif
11      continue
        if(l1) then
           l1=.false.
        else
           l1=.true.
        endif
        l2=.false.
10    continue
      ns=ns/2
      if(ns.ge.1) goto 1
c
c
      open(99,file='f')
c     write(99,*) ni2
      do 20 j=1,ni2+1
         do 21 i=1,ni2+1
            if(a(i,j)/amax.le.w) a(i,j)=w
            write(99,*) i,j,int(a(i,j)*1000*dd)
21       continue
         write(99,'(a,$)') char(10)
20    continue
      close(99)
      end
```


Eine Möglichkeit zur experimentellen Bestimmung von p_c ist eine binäre Mischung aus leitenden und nicht-leitenden Kugeln (z.B. Glas und Stahl). Sind wenig Stahlkugeln im System als der Bruchteil p_c, so leitet die Versuchsanordnung nicht. Bei einem größeren Anteil leitender Kugeln stellt man zunehmende Leitfähigkeit mit zunehmendem Stahlanteil fest.

Anstatt auf einem Gitter kann man Perkolation auch im Kontinuum untersuchen. Anstelle einzelner Gitterplätze belegt man das System mit Scheiben (oder Löchern) an zufälligen $x-y-$Koordinaten und mit zufälliger Größe. Nach dem entstehenden Bild wird dieses Modell auch "Swiss-Cheese-Model" genannt.

5.7.2 Das Edenmodell

Eine weitere Anwendung von Zufallszahlen ist das im Deutschen Krebsforschungszentrum für Tumorwachstum entwickelte Edenmodell. Auf einem Gitter wird ein beliebiger Platz besetzt - dies definiert den Ur-Cluster. An der Oberfläche erlaubt man nun das Wachsen des "Tumors". Für die Simulation benötigt man deshalb eine Liste aller Oberflächenplätze an denen der Cluster wachsen kann. Davon wird einer ausgesucht und aus der Liste gestrichen, nachdem dort das Wachstum stattgefunden hat. Mit dieser Simulationsmethode stellt man nach einiger Zeit einen zerfransten Rand fest.

Eine Alternative ist nicht von einer Keimzelle, sondern von einer Linie auszugehen. Die durch das Wachstum entstehende Oberfläche ist wieder rauh. Als Maß für die Rauhigkeit kann man die Schwankung W der gewachsenen Oberfläche h_i um Ihren Mittelwert $\langle h \rangle$ verwenden:

$$W = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(h_i - \langle h \rangle\right)^2}\ , \ \text{mit} \ \ \langle h \rangle = \frac{1}{N}\sum_{i=1}^{N} h_i\ . \tag{5.51}$$

Dieses Verfahren kann zur Simulation des Wachstums von Pflanzen und Kristallen verwendet werden.

Bei einer Epidemie können Zellen gegen Krankheitserreger immun werden. An solchen Stellen kann sich die Krankheit nicht weiter ausbreiten. Ist die Wahrscheinlichkeit p, daß eine Zelle immun wird zu groß, so hört das Wachstum ganz auf, da alle Zellen an der Oberfläche immun geworden sind. Je nach Modell kann man wieder die kritische Wahrscheinlichkeit p_c aus dem letzten Abschnitt entdecken. Bei Werten von p in der Nähe von p_c entstehen durch die immunen Zellen immer größere Löcher im Cluster. Große Löcher bedeutet eine fraktale Struktur, da die quadratische Ausdehnung des Clusters stärker anwächst als seine Masse, siehe Abb. 5.13.

Das zur Erzeugung verwendete Programm sieht folgendermassen aus:

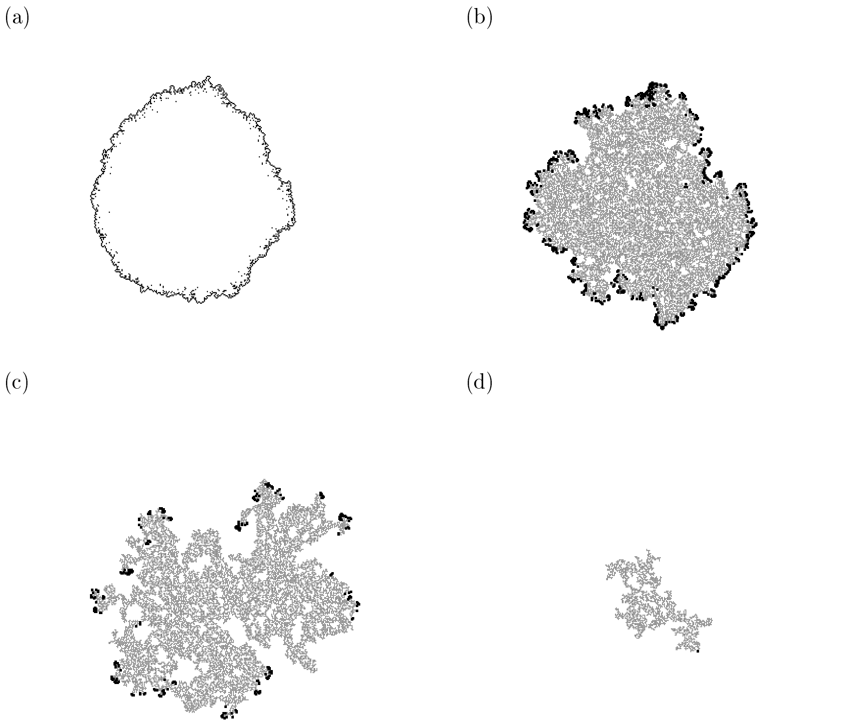


Abbildung 5.13: Eden-Wachstumsmodell mit Immunitätsraten (a) $p = 0$, (b) $p = 0.35$, (c) $p = 0.40$ und (d) $p = 0.415$.

```
#include<iostream>
#include<fstream>
using std::cout;
using std::cin;
#include"rng2.h"

int newsurf( int i, int j );
static const int NL=500;
static int world[NL+2][NL+2];    // define the lattice
static int surface[NL*NL][2], nums=0;
static int *psurf[NL*NL];

int main()
```



```
c
      program RMD
c
      implicit integer(h-k,m-n)
      implicit real(a-g,o-z)
      implicit logical(l)
c
      parameter(nmax=9,n2=2**nmax)
      dimension a(n2+1,n2+1)
c
      write(*,'($,a)') 'Iterationen?       '
      read(*,*) it
      write(*,'($,a)') 'Negativparameter?  '
      read(*,*) pn
      write(*,'($,a)') 'Streckung?         '
      read(*,*) dd
      write(*,'($,a)') 'SEED1?              '
      read(*,*) NSEED1
c
      call srand(NSEED1)
c
      w=-.005
c
      ni2=2**it
      ns=ni2/2
c
      a(1,1)=w
      a(1,ni2+1)=w
      a(ni2+1,1)=w
      a(ni2+1,ni2+1)=w
      amax=w
c
1     continue
      l1=.false.
      l2=.false.
      do 10 i=1,ni2+1,ns
         do 11 j=1,ni2+1,ns
            if(l1.or.l2)then
               a(i,j)=0.
               if(l1.and.l2)then
                  a(i,j)=(a(i+ns,j+ns)+a(i-ns,j+ns)+
     1                    a(i+ns,j-ns)+a(i-ns,j-ns))/4.+
     2                    (rand()-pn)*ns
                  if(a(i,j).gt.amax) amax=a(i,j)
               else if(l1.and.(.not.l2)) then
```

5.7.5 Random Midpoint Displacement

Ein schöner Algorithmus, den man verwenden kann um fraktale Landschaften zu erzeugen, ist das sog. “random midpoint displacement” (RMD). Die Idee ist daß man ausgehend von einer Einheitszelle (Dreieck oder Quadrat) alle Verbindungslinien halbiert (in beiden Fällen) und diese Punkte dann zufällig vertikal versetzt. Im Fall des Quadrats muß man außerdem noch den Punkt im Zentrum versetzen, damit man vier Quadrate, diesmal halber Gr”ße, erhält. Diese Operation führt man nun für alle “Tochter”-Quadrate iterativ beliebig oft durch. Mit jeder Iteration steigt der Aufwand um den Faktor 4, und man hat nach n Iterationen bereits ein Gitter der Auflösung 2^n. Bei $n = 10$ sind sowohl dem Speicherplatz des Computers als auch der Auflösung am Bildschirm natürliche Grenzen gesetzt.

In Abb. 5.18 sind die ersten sechs Iterationen einer bestimmten Zufallszahlenserie abgebildet. Diese Bilder wurden mit dem `gnuplot`-script

```
set term post enhanced color
set hidden3d
set contour
set out 'rmd6.ps
splot 'f.6' u 1:2:($3/150) t 'Iteration 6' w l lt 1
```

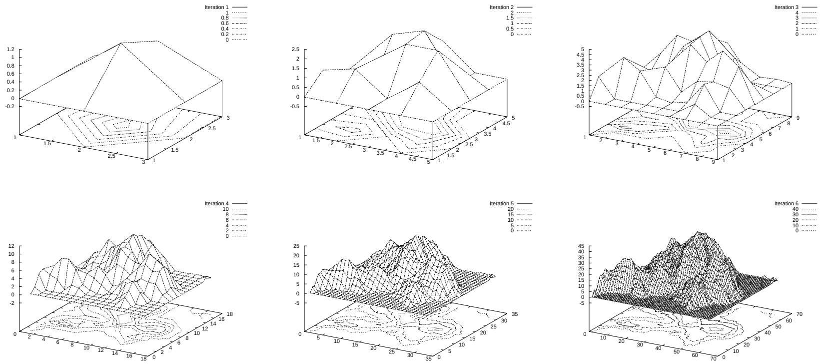


Abbildung 5.18: Random Midpoint displacement verschiedener Iterationstiefe.

Zum Abschluß dieses Kapitels (und zur Abschreckung) wird in diesem Abschnitt das Beispielprogramm in der Programmiersprache FORTRAN vorgestellt.


```
{
  int NGROW=30000;
  float prob;
  int igrow;

  cout << "# Perkolationswahrscheinlichkeit ? ";
  cin >> prob;   cout << prob << '\n';

  for(int i=1; i<=NL; i++ )
    for(int j=1; j<=NL; j++ )
      world[i][j]=0;                     // empty sites
  world[NL/2][NL/2]=2;                   // only one is occupied

  newsurf( NL/2, NL/2 );

  rng0(12345678);                        // initialize rng()

  for( int it=1; it<=NGROW; it++ )
  {
    igrow = int( nums*double(rng())/dp );
    if( double(rng())/dp > prob ){
      world[ *(psurf[igrow]) ][ *(psurf[igrow]+1) ]=2;
      if( newsurf( *(psurf[igrow]), *(psurf[igrow]+1) ) == -1 ){
        cout << "# hit boundary of field ... \n";
        break;
      }
    }
    else
      world[ *(psurf[igrow]) ][ *(psurf[igrow]+1) ]=-1;
    surface[igrow][0]=surface[nums-1][0];
    surface[igrow][1]=surface[nums-1][1];
    nums -= 1;
    if( nums == 0 ){
      cout << "# last surface site ... \n";
      break;
    }
  }

  ofstream outfile1("edens.dat");
  for(int i=0; i<=nums; i++ ){
    outfile1 << *psurf[i] << ' ' << *(psurf[i]+1) << '\n';
  }
  outfile1.close();

  ofstream outfile2("edenc.dat");
```

```
  for(int i=1; i<=NL; i++ ){
    for(int j=1; j<=NL; j++ ){
      if(world[i][j] == 2)
        outfile2 << i << ' ' << j << ' ' << world[i][j] << '\n';
    }
  }
  outfile2.close();

  ofstream outfile3("edenx.dat");
  for(int i=1; i<=NL; i++ ){
    for(int j=1; j<=NL; j++ ){
      if(world[i][j] == -1)
        outfile3 << i << ' ' << j << ' ' << world[i][j] << '\n';
    }
  }
  outfile3.close();
  return 0;
}

int newsurf( int i, int j )
{
  if(( i<1 ) || ( i>NL ) || ( j<1 ) || (j>NL ))
     return -1;

  if( world[i-1][j] == 0 ){
    surface[nums][0]=i-1;
    surface[nums][1]=j;
    world[i-1][j] = 1;
    psurf[nums]=&surface[nums][0];
    nums++;
  }
  if( world[i+1][j] == 0 ){
    surface[nums][0]=i+1;
    surface[nums][1]=j;
    world[i+1][j] = 1;
    psurf[nums]=&surface[nums][0];
    nums++;
  }
  if( world[i][j-1] == 0 ){
    surface[nums][0]=i;
    surface[nums][1]=j-1;
    world[i][j-1] = 1;
    psurf[nums]=&surface[nums][0];
    nums++;
  }
```



```
double determinants( int NA )
{
  double sumdets = 0.0;
  for( int i=0; i<NA; i++ ){
    P[i] =  D[i][0][0] * D[i][1][1]          // calculate determinants
          - D[i][0][1] * D[i][1][0];
    if( P[i] <= 0.0 ) P[i]=-P[i];
    sumdets += P[i];
  }
  for( int i=0; i<NA; i++ ){
    P[i] /= sumdets;                          // normalize
  }
  return sumdets;
}

int iteration( int NA, double &x, double &y )
{
  int ierr = 0, iabb = 0;
  double sum = 0.0, z;

  if( rng()/dp > 0.1 ){                        // do this 90%
    z = rng()/dp;
    for( int i = 0; i < NA; i++ ){
      sum += P[i];
      if( sum >= z ){
        iabb = i;                              // select D_i according
        break;                                 // to the weight of its
      }                                        // determinant
    }
  }
  else{                                        // do this 10%
    iabb = int( rng()/dp*NA );                 // random D_i
  }
  z = D[iabb][0][0] * x + D[iabb][0][1] * y + C[iabb][0];
  y = D[iabb][1][0] * x + D[iabb][1][1] * y + C[iabb][1];
  x = z;

  return ierr;
}
```

```
  int NA, n1=1, n2=100;
  double sumdets=0.0;
  double x, y;

  cout << "# Inputfile ? ";
  cin >> cinfile;
  cout << " ... read from " << cinfile << '\n';
  cout << "# Iterations ? ";
  cin >> n2; cout << n2 << '\n';

  ifstream detfile(cinfile);
  detfile >> NA;
  cout << "# " << NA << " determinants ... \n";
  for( int i=0; i<NA; i++ ){
    cout << "# det " << i << ' ';
    for( int j=0; j<2; j++ ){
      for( int k=0; k<2; k++ ){
        detfile >> D[i][j][k];
        cout << D[i][j][k] << ' ';
      }
    }
    for( int j=0; j<2; j++ ){
      detfile >> C[i][j];
      cout << C[i][j] << ' ';
    }
    cout << '\n';
  }
  detfile.close();                          // end of reading data

  rng0(12345678);
  rng(); rng(); rng(); rng();               // warm up random numbers

  sumdets = determinants( NA );             // initialize determinant weights

  x = rng()/dp * 1000;
  y = rng()/dp * 1000;
  for( int i=0; i<500; i++ )                // start with 500 warm-up steps
     iteration( NA, x, y );

  for( int i=n1; i<=n2; i++ ){              // perform selected iterations
     if( iteration( NA, x, y ) < 0 )
        break;
     cout << x << ' ' << y << '\n';         // output of r'
  }
}
```



```
  if( world[i][j+1] == 0 ){
    surface[nums][0]=i;
    surface[nums][1]=j+1;
    world[i][j+1] = 1;
    psurf[nums]=&surface[nums][0];
    nums++;
  }
  return 0;
}
```

5.7.3 Diffusions-limitierte Anlagerung DLA

Das Modell der diffusions-limitierten Anlagerung (DLA = diffusion limited aggregation) wurde 1981 von Whitten und Sander eingeführt. Die zugrundeliegende Idee ist, Teilchen (Random Walker) aus dem Unendlichen losfliegen zu lassen und sie an einen Keim anzulagern, welcher durch diese Anlagerung immer weiter wächst. Die entstehenden Formen, siehe Abb. 5.14 (Links), sind sehr häufig in der Natur zu finden (Wasser das in Öl gepresst wird, Rußaggregate, Blitze).

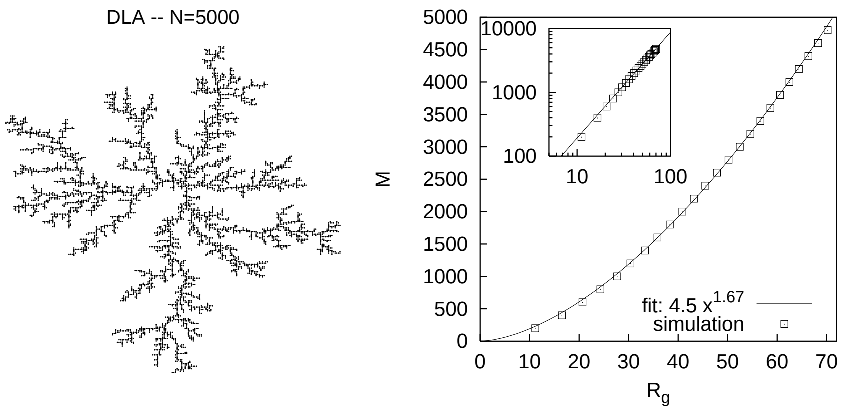


Abbildung 5.14: DLA Cluster mit $N = 5000$ Molekülen (Links). Masse als Funktion des Gyrationsradius R_g (Rechts) – auch in doppelt-logarithmischer Darstellung (Fenster).

Man kann die Ausdehnung des Objekts durch den sog. Gyrationsradius

$$R_g = \sqrt{(1/N) \sum_{i=1}^{N} (\mathbf{r}_i - < \mathbf{r} >_N)^2} \tag{5.52}$$

quantifizieren und dann die Masse M (=Anzahl der Moleküle N) als Funktion von R_g darstellen, siehe Abb. 5.14 (Rechts). Die Steigung der Geraden, die man in doppelt-logarithmischer Darstellung findet, ist die sog. fraktale Dimension d_f, womit die Masse dem Gesetz

$$M \propto R_g^{d_f} \tag{5.53}$$

folgt. Ist d_f von der Dimension des einbettenden Raums (hier $D = 2$) verschieden, dann nennt man das Objekt ein Fraktal. Für kompakte Objekte ist d_f eine ganze Zahl $d_f = D$, für Fraktale eine reelle Zahl – im Fall der DLA ist $d_f \approx 1.66$. Ein DLA-Cluster ist also ein Objekt zwischen einer Linie und einer Fläche, mit mehr Masse als die Linie, aber weniger Masse als die solide Fläche bei gleicher Ausdehnung.

Eine Konsequenz von Gleichung 5.53 ist die Selbstähnlichkeit der Fraktale: Betrachtet man ein Fraktal (im Mikroskop) bei verschiedenen Vergrößerungen, so kann man subjektiv keinen Unterschied feststellen.

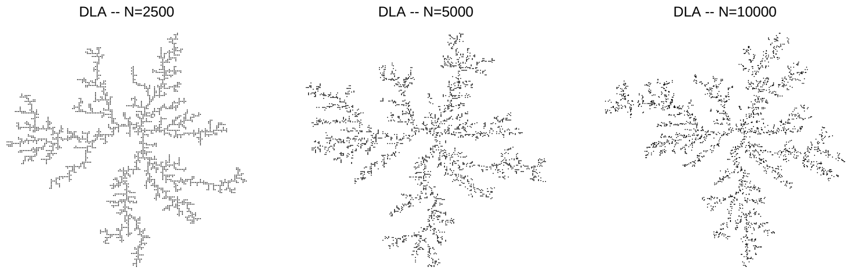


Abbildung 5.15: Verschieden große DLA Cluster mit $N = 2500$, 5000 und 10000 Molekülen. Die Vergrößerung ist so eingestellt daß die DLA Cluster in etwa die selben Flächen bedecken.

Zur Simulation von DLA bedient man sich im wesentlichen eines Random Walks aus Kapitel 5.4. Um Zeit zu sparen beginnt man mit dem Random Walk in der Nähe des Keims (bzw. des Clusters). Um keine Richtung zu bevorzugen setzt man alle RW nacheinander zufällig auf einen Kreis um den Cluster. Von dort läßt man ihn loslaufen – verwirft ihn aber auch wieder falls er aus dem Gitter (dem System) hinauslaufen will. Wieder wird als Kriterium der Abstand vom Zentrum genommen und nicht etwa der Rand des Simulationsgitters, da damit manche Richtungen bevorzugt würden. Sobald ein RW den DLA Cluster erreicht lagert man ihn an und läßt den nächsten RW loslaufen. Eine Variante, nämlich eine Anlagerungswahrscheinlichkeit mit $p_a < 1$ ist hier nicht realisiert.

Ein Simulationsprogramm, das den oben beschriebenen Algorithmus realisiert könnte folgendermaßen aussehen:

vor, während die Transformationsmatrizen lediglich um den Faktor zwei verkürzen. Hängt man die vierte Abbildung:

$$\mathcal{D}_4 = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix} \text{ und } \mathbf{C}_4 = \begin{pmatrix} 100 \\ 230 \end{pmatrix} \tag{5.58}$$

an die drei anderen, so ergibt sich die Projektion des dreidimensionalen Sierpinski Gitters, wie in Abb. 5.17 (rechts) dargestellt.

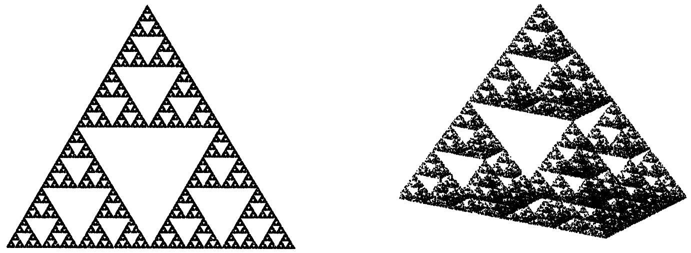


Abbildung 5.17: Das Sierpinski Gitter, erzeugt durch iterative Abbildung (links), und sein großer Bruder der durch vier hintereinandergeschaltete lineare Abbildungen erzeugt wurde.

```
#include<iostream>
#include<fstream>
using std::cout;
using std::cin;
using std::ofstream;
using std::ifstream;
#include"rng2.h"

static const int NDMAX=100;                    // max. number of ...
static double D[NDMAX][2][2], C[NDMAX][2];     // linear transformation
static double P[NDMAX];                        // det. weights

double determinants( int NA );
int iteration( int NA, double &x, double &y );

int main()
{
   char cinfile[20];
```

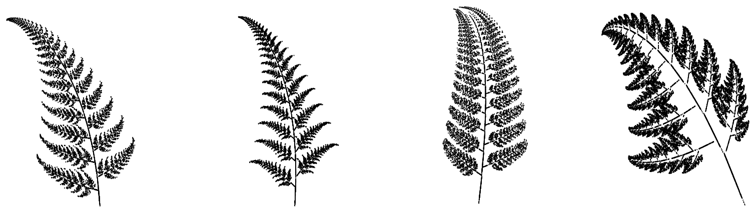


Abbildung 5.16: Vier verschiedene Farne – erzeugt durch verschiedene Iterationen.

Die Abbildungen für Farne sind einigermassen kompliziert. Deshalb wird hier am Beispiel des bekannten Sierpinski Gitters die Erzeugung durch Iteration genauer diskutiert. Der Kern der Iteration besteht aus der linearen Abbildung

$$\mathbf{r}' = \mathcal{D}\mathbf{r} + \mathbf{C} \;, \tag{5.54}$$

wobei $\mathbf{r}$ der Ausgangspunkt (x, y) und $\mathbf{r}$' der seine Abbildung ist. Die Abbildung ist zusammengesetzt aus einem Verschiebungsvektor $\mathbf{C}$ und einer Transformationsmatrix D. Eine Iteration besteht dabei aus einer von `NA` linearen Abbildungen.

Für das Sierpinki Gitter braucht man die drei Abbildungen:

$$\mathcal{D}_1 = \begin{pmatrix} 0.50194129421 & 0 \\ 0 & 0.50194129421 \end{pmatrix} \text{ und } \mathbf{C}_1 = \begin{pmatrix} 152 \\ 7 \end{pmatrix} \;, \tag{5.55}$$

$$\mathcal{D}_2 = \begin{pmatrix} 0.50184091097 & 0 \\ 0 & 0.50184091097 \end{pmatrix} \text{ und } \mathbf{C}_2 = \begin{pmatrix} 36 \\ 191 \end{pmatrix} \tag{5.56}$$

und

$$\mathcal{D}_3 = \begin{pmatrix} 0.49697157843 & 0 \\ 0 & 0.49697157843 \end{pmatrix} \text{ und } \mathbf{C}_3 = \begin{pmatrix} 265 \\ 192 \end{pmatrix} \;. \tag{5.57}$$

Im wesentlichen sind das also drei Streckungen gekoppelt mit drei Verschiebungen. Das aus dieser Abbildung nach 40,000 Iterationen (bei 500 Aufwärm-Iterationen) entstehende Fraktal ist in Abbildung 5.17 (links) dargestellt. Die Abbildungen werden nach zwei Prinzipien ausgewählt. Einmal, mit 10-prozentiger Wahrscheinlichkeit, wählt man einfach zufällig eine der `NA` Abbildungen, oder man benutzt, mit 90-prozentiger Wahrscheinlichkeit, die linearen Abbildungen gewichtet nach der Determinante von $\mathcal{D}$.

Verwendet man anstelle der drei Abbildungen nur die ersten beiden, so erhält man lediglich eine Gerade. Die drei Verschiebungsvektoren geben im wesentlichen die Dreiecksstruktur


```
#include<iostream>
#include<fstream>
#include<cmath>
using std::cout;
#include"rng2.h"

static const int NL=1000;         // grid size
static int world[NL+2][NL+2];     // define the grid for the DLA

int main()
{
  int NGROW=5000;                 // stop after 5000 particles DLA
  int DNOUT=200;                  // control output after 200 new pps
  int OUTNXT=DNOUT;               // next output particle number
  int i0x, i0y, imerge;           // coordinates of the random walker
  int sum_particles = 0;          // pp-sum
  double dxy, rxm, rym, r2m;      // distance variables
  double rnum, rmax, rij;

  for(int i=1; i<=NL; i++ )
    for(int j=1; j<=NL; j++ )
      world[i][j]=0;              // all empty sites
  world[NL/2][NL/2]=1;            // only one is occupied

  rng0(12345670);                 // initialize rng()
  ofstream outfile1("dla1.dat"); // open datafile for DLA output

  rmax = 5.;                      // radius of the initial setup circle
                                  // BEGIN loop for new RW
  for( int it=1; it<=NGROW*10; it++ )
  {
     rnum = 2.*M_PI*double(rng())/dp;              // choose angle on circle
     i0x = NL/2 + int( rmax * sin( rnum ) );       // determine the next RW
     i0y = NL/2 + int( rmax * cos( rnum ) );       //   starting point

     imerge = 1;
     while(( world[ i0x+1 ][ i0y ] == 0 ) &&       // RW loop is performed
          ( world[ i0x ][ i0y+1 ] == 0 ) &&       // until the DLA is hit
          ( world[ i0x ][ i0y-1 ] == 0 ) &&
          ( world[ i0x-1 ][ i0y ] == 0 )){
          dxy = double(rng())/dp;                 // select RW direction

          if( dxy < 0.25 ){
             if( i0x > 1 )
                if( world[ i0x-1 ][ i0y ] == 0 ) i0x -= 1;       // LEFT
```

```
      }
      else{
         if( dxy < 0.5 ){
            if( i0x < NL )
               if( world[ i0x+1 ][ i0y ] == 0 ) i0x += 1;    // RIGHT
         }
         else{
            if( dxy < 0.75 ){
               if( i0y > 1 )
                  if( world[ i0x ][ i0y-1 ] == 0 ) i0y -= 1; // DOWN
            }
            else{
               if( i0y < NL )
                  if( world[ i0x ][ i0y+1 ] == 0 ) i0y += 1; // UP
            }
         }
      }
      rij=(i0x-NL/2)*(i0x-NL/2)+(i0y-NL/2)*(i0y-NL/2);
      if( sqrt( rij ) > 1.5*rmax ){
         imerge=0;                   // if the distance from the center
         break;                      //   is too large then forget the RW
      }
   }

   if( imerge == 1 ){                // add new particle to the DLA
      world[ i0x ][ i0y ] = sum_particles;
      outfile1 << i0x << ' ' << i0y << ' ' << world[i0x][i0y] << '\n';

      sum_particles += 1;            // count total number
      // cout << i0x << ' ' << i0y << ' '
      //      << sum_particles << ' ' << rmax << '\n';

      rij=(i0x-NL/2)*(i0x-NL/2)+(i0y-NL/2)*(i0y-NL/2);
      if( sqrt(rij)+5 > rmax ){
         rmax=sqrt(rij)+5;               // let inscribing circle grow
      }                                  //    if necessary
      if(( int( 1.5*rmax ) > NL/2-2 ) ||
         ( sum_particles >= NGROW )){    // STOP if outer circle
                                         //    hits world's end
         cout << "# last rmax = " << rmax << '\n';
         outfile1.close();
         return sum_particles;
      }                               // end of merge-IF
   }                                  // end of RW-WHILE
```



```
      rxm=0.; rym=0.; r2m=0.;              // calculate CM position and Rg
      if( OUTNXT == sum_particles ){       //    if next output time
         OUTNXT += DNOUT;                  // increase for the next output
         for(int i=1; i<=NL; i++ ){
           for(int j=1; j<=NL; j++ ){
             if(world[i][j] > 0){
               rxm+=i;                     // x-mean
               rym+=j;                     // y-mean
             }
           }
         }
         rxm /= sum_particles;             // center of mass position x
         rym /= sum_particles;             // center of mass position y

         for(int i=1; i<=NL; i++ ){
           for(int j=1; j<=NL; j++ ){
             if(world[i][j] > 0){
               rij=(i-rxm)*(i-rxm)
                  +(j-rym)*(j-rym);
               r2m+=rij;                  // average of the variance
             }
           }
         }
         r2m = sqrt(r2m/sum_particles);    // compute the radius of gyration

         cout << sum_particles << ' ' << rxm << ' ' << rym << ' '
              << r2m << ' ' << rmax << ' ' << '\n';
      }
   }
   outfile1.close();
   return 0;
}
```

5.7.4 Fraktale durch Iterationen

Eine elegante Möglichkeit Fraktale zu erzeugen ist die von Barnsley vorgeschlagene Methode iterativer Funktionensysteme. Angefangen mit einer Zufallskoordinate werden durch wiederholte, verschachtelte lineare Abbildungen immer weitere Punkte erzeugt, die in ihrer Gesamtheit schließlich ein fraktales Bild ergeben. Wie bei vielen ähnlichen Mehtoden läßt man den Algorithmus zuerst "warmlaufen", bevor man die Punkte zeichnet. Vier mit dieser Methode erzeugte Farne sind in Abb. 5.16 dargestellt. Der Unterschied zwischen den einzelnen Bildern sind nur die eingespeisten Abbildungen – es wurde jeweils derselbe Algorithmus verwendet.