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, T. Karle und S. Manmana
Uni–Stuttgart

11. Februar 1999

Inhaltsverzeichnis

1 Einführung **1**

1.1 Was heißt "Physik auf dem Computer" 1

1.1.1 Vom Menschen zum Computer und zurück 2

1.1.2 Das erste Beispiel 3

1.2 Programmier-Philosophie 6

1.3 Geschichte der Programmierung mit C++ 7

1.4 Anwendungsbeispiele 8

1.5 Literatur 8

2 Grundlagen von C++ **9**

2.1 Sprachkonstrukte 9

2.2 Kommentare 11

2.3 Datentypen, Variablen, Konstanten 12

2.3.1 Grundlegende Datentypen und Deklaration 12

2.3.2 Variablenattribute 15

2.3.3 Lebensdauer und Lage der Variablen im Systemspeicher 15

2.3.4 Felder 16

2.4 Kontrollstrukturen 17
2.4.1 if-Anweisung 17
2.4.2 Ternärer ?:-Operator 18
2.4.3 (do) while-Schleifen 19
2.4.4 for-Schleifen 20
2.4.5 break und continue 20
2.4.6 goto-Anweisung, Labels 21
2.4.7 switch-Anweisung 22
2.5 Funktionen und Operatoren 23
2.5.1 Funktionen ohne Argumente oder Rückgabewert, void 26
2.5.2 Bibliotheken 26
2.5.3 Operatoren 27
2.5.4 inline Funktionen 30
2.5.5 Defaultargumente 32
2.5.6 Überladen von Funktionen 33
2.6 Zeiger, Zeiger-Feld-Dualität, und Referenzen 34
2.6.1 Zeiger 34
2.6.2 Zeiger-Feld-Dualität 39
2.6.3 Feldvariablenübergabe an Funktionen, dynamische Speicherverwaltung 39
2.6.4 Referenzen 41
2.6.5 typedef 42
2.7 Bezugsrahmen von Bezeichnern 42

2.8 Ein- und Ausgabe 43

2.8.1 Elementfunktionen der iostreams 44

2.8.2 Formatierung 45

2.8.3 Dateien (Files) 47

2.9 Dynamische Speicherverwaltung 48

2.10 Organisation in Implementierungs- und Header-Dateien 49

2.11 Weiterführende Literatur 50

3 Lösung der Newtonschen Bewegungsgleichung 51

3.1 Der Harmonische Oszillator 51

3.1.1 Das Euler-Verfahren 53

3.1.2 Die Euler-Cromer-Methode 55

3.2 Das Pendel 55

3.2.1 Die Verlet-Methode 57

3.2.2 Das allgemeine Pendel 59

3.2.3 Das Runge-Kutta-Verfahren 59

3.2.4 Chaos 62

3.2.5 Der Poincaréschnitt 63

3.2.6 Übergang periodisch – chaotisch 64

3.3 Himmelsmechanik 65

3.3.1 Erde und Jupiter 70

3.3.2 Das Drei–Körper–Problem 73

3.3.3 Perihelbewegung des Merkurs 74

3.3.4 Starre Körper 74

3.4 Molekulardynamik 74

3.4.1 Das Heliumatom 75

3.4.2 Optimierung 76

3.4.3 Weitere Vielteilchensysteme 77

4 Populationsdynamik **79**

4.1 Betrachtung einer einzelnen Spezies 79

4.1.1 Iterative Modelle 79

4.1.2 Gewöhnliche Differentialgleichungen 80

4.2 Zwei und mehr Spezies 83

4.2.1 Volterra–Gleichung (1925) 83

4.2.2 Lotka–Volterra–Gleichung (1926) 84

4.2.3 Folgerungen 85

4.3 Ausblick: Diskrete Modelle 86

5 Gitter- oder Stochastische Modelle **87**

5.1 Zufallszahlen 87

5.1.1 Kongruentieller RNG “IBM” 87

5.1.2 Lagged Fibbonacci Sequenzen 90

5.1.3 Testen von Zufallszahlen 92

5.2 Zellularautomaten 93

5.3 Game of life 96

5.4 Random walks (RW) 100

5.5 Random Walk und Diffusion 105

5.6 Fehler, Varianz und lineare Regression 106

5.6.1 Fehler 106

5.6.2 Lineare Regression 107

5.7 Weitere Anwendungsbeispiele 108

5.7.1 Perkolation 109

5.7.2 Das Edenmodell 115

5.7.3 Diffusions-limitierte Anlagerung DLA 119

5.7.4 Fraktale durch Iterationen 123

5.7.5 Random Midpoint Displacement 128

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

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

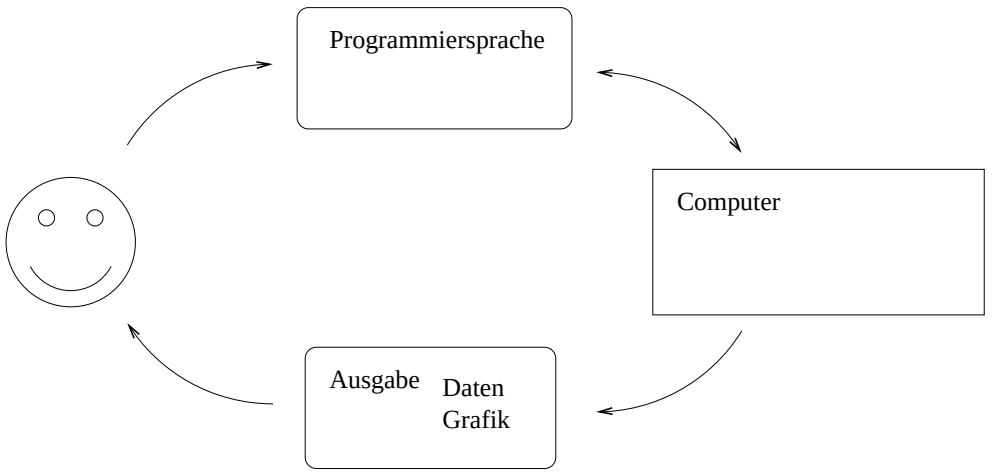


Abbildung 1.1: Schematische Darstellung der Interaktion zwischen Mensch und Computer

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.

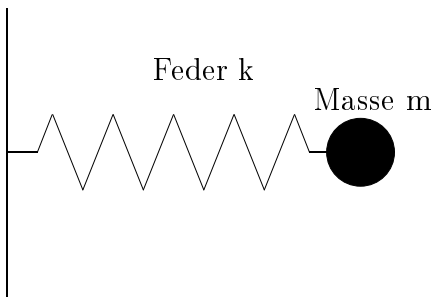


Abbildung 1.2: Harmonischer Oszillator

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()
{
```

```
// 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
```

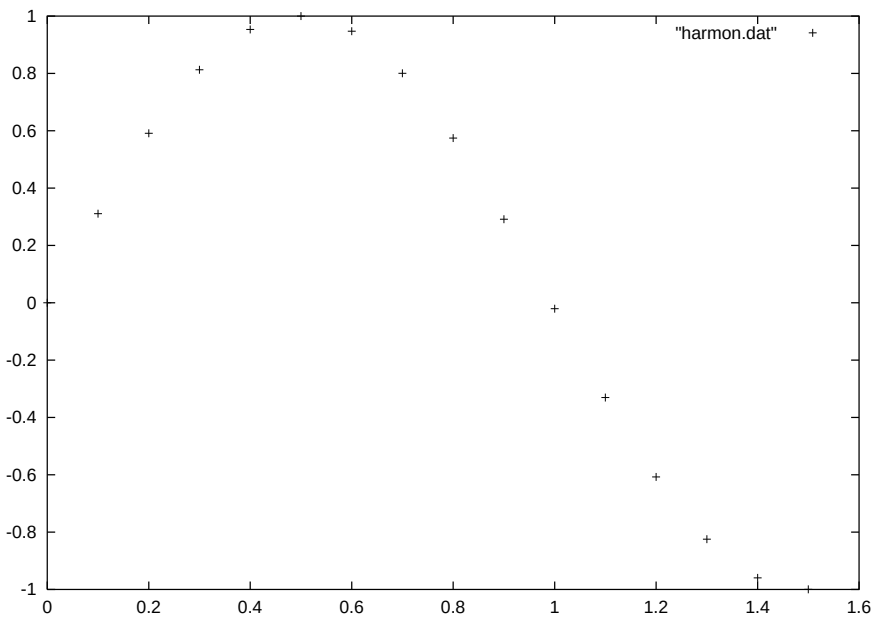


Abbildung 1.3: Grafische Darstellung der zweispaltigen Zahlenreihe auf dem Bildschirm. Die erste Spalte definiert die x-, die zweite die y-Koordinate der Punkte.

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.

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 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.

- 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 verwendeten. 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
```

```
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.

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 `automatic`. 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>
```

```
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;
```

```
}
```

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.

```
// 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
```

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:

```
#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:
```

```
#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
```

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:

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

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:

```
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

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.

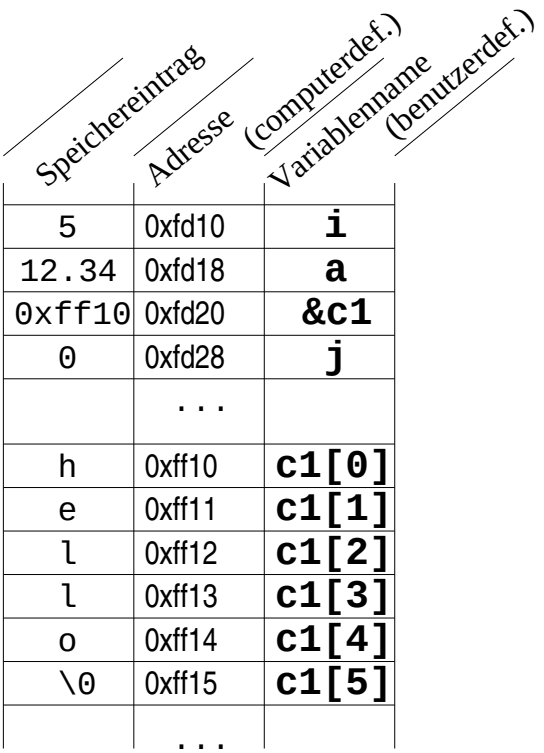


Abbildung 2.1: Speicherbelegung (schematisch) mit realem Speichereintrag (Links), einer willkürlich gewählten, Computer-internen Adresse (Mitte) und dem benutzerdefinierten Variablennamen (Rechts).

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.

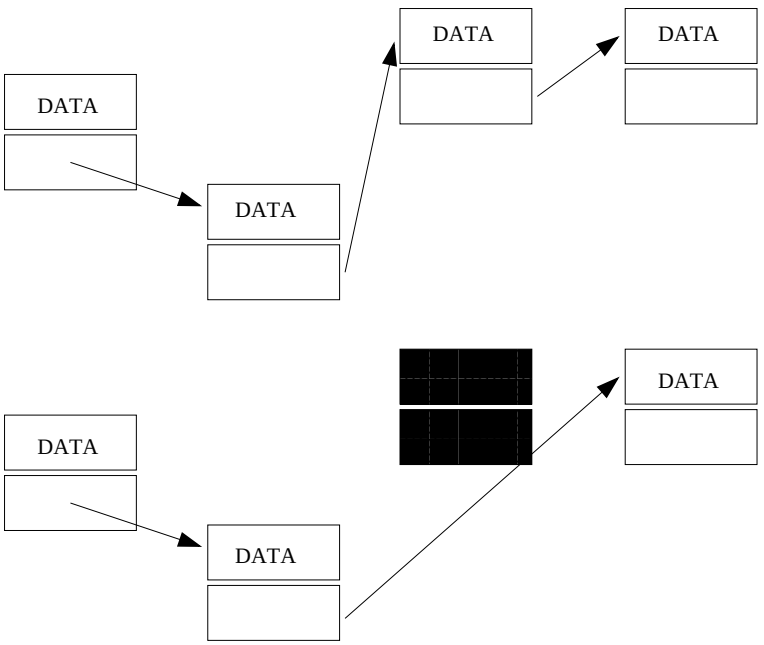


Abbildung 2.2: Benutzung von Zeigern am Beispiel einer verketteten Liste. Oben sieht man die Liste vor Entfernen eines Elementes. Unten ist es nicht in der Liste vorhanden.

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.

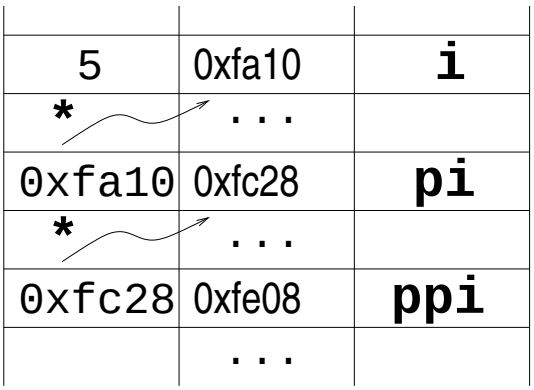


Abbildung 2.3: Speicherbelegung: Bedeutung der Spalten wie in Abb. 2.1.

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 Speicherallozііerungsroutine `malloc`). In C++ sollten in solchen Fällen `template`-Funktionen zum Einsatz kommen, die die Typsicherheit des Programmes erhalten.

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
```

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

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.

`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.

```
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;
```

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.

Kapitel 3

Lösung der Newtonschen Bewegungsgleichung

3.1 Der Harmonische Oszillator

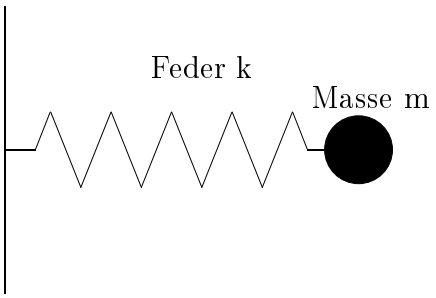


Abbildung 3.1: Schwingungsfeder

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
```

```
   }

// 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.

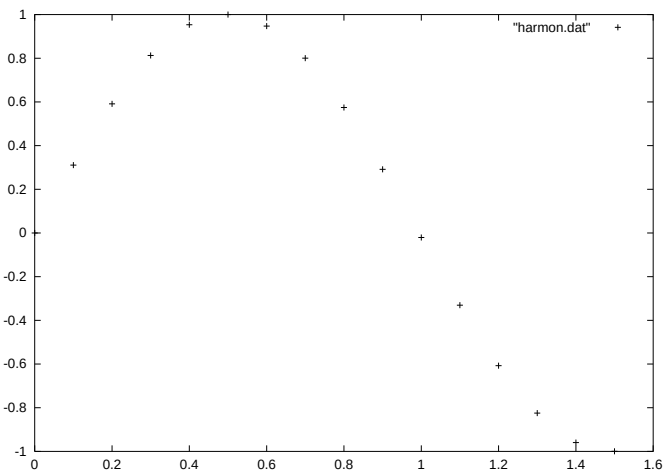


Abbildung 3.2: Ausdruck von gnuplot für $A = 1$, $\delta = 0$, $m = 1$, $k = 10$, $t_{\max} = 1.5$, und $\Delta t = 0.1$, direkt aus der analytischen Lösung berechnet.

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();
```

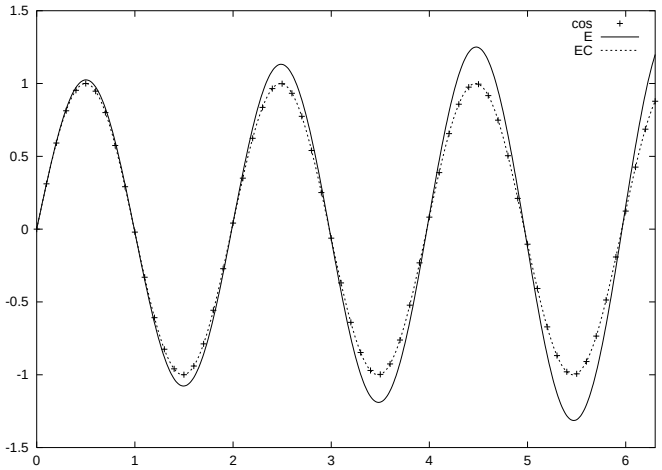


Abbildung 3.3: Ausdruck von gnuplot für $A = 1$, $\delta = 0$, $m = 1$, $k = 10$, $t_{\max} = 6.3$, und $\Delta t = 0.01$, berechnet mit dem Euler-Verfahren (E) im Vergleich mit der exakten Lösung aus Abb. 3.2, und dem Euler-Cromer Verfahren (EC).

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

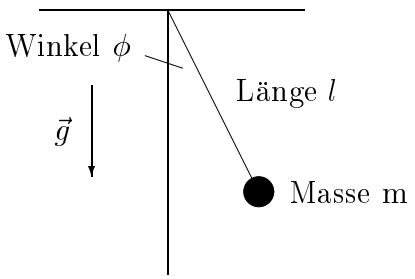


Abbildung 3.4: Das Pendel mit masseloser Aufhängung

Bei einem Pendel ist die (punktförmige) Masse m an einem starren, masselosen Stab der Länge l aufgehängt und schwingt reibungsfrei im Schwerefeld $\vec{g}$. $\varphi(t)$ 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

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

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:

$$F(k,\varphi) = \int_0^{\varphi} \frac{d\psi}{\sqrt{1 - k^2\sin^2\psi}} \ , \tag{3.12}$$

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

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.

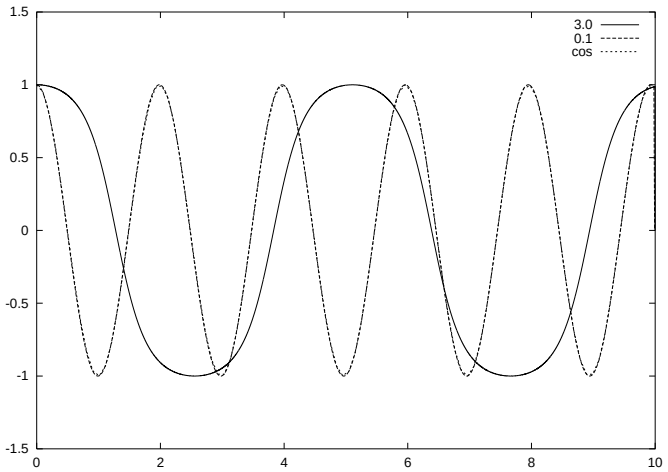


Abbildung 3.5: Ausdruck von $\varphi(t)/\varphi_0$ für $g = 10$, $l = 1$, $m = 1$, $\dot{\varphi}_0 = 0$, $t_{\max} = 10$ und $\Delta t = 0.01$, berechnet mit dem Verlet-Verfahren für $\varphi_0 = 0.1$ und $\varphi_0 = 3$, verglichen mit $\cos(\sqrt{10}t)$.

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

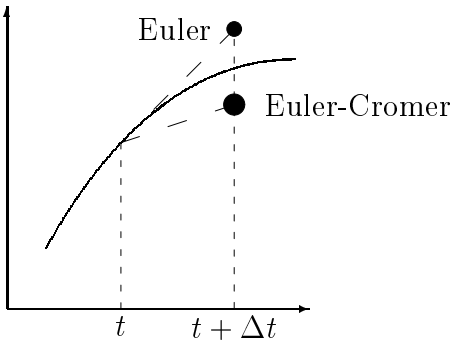


Abbildung 3.6: Fehler beim Euler- / Euler-Cromer-Verfahren

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.

$$\varphi(t + \Delta t) = \varphi(t) + \Delta t \; \bar{\omega} \tag{3.28}$$

$$\omega(t + \Delta t) = \omega(t) + \Delta t \; f\left[\bar{\varphi}, \bar{\omega}, t + \frac{\Delta t}{2}\right] \tag{3.29}$$

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.

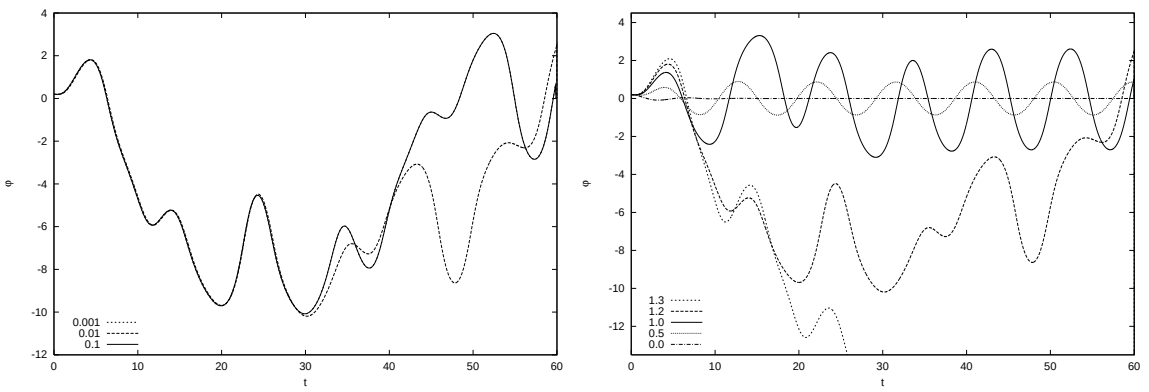


Abbildung 3.7: Links: Pendelausschlag φ als Funktion der Zeit t für verschiedene Δt. Rechts: φ als Funktion von t für verschiedene F_0.

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

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 $\lambda > 0$.

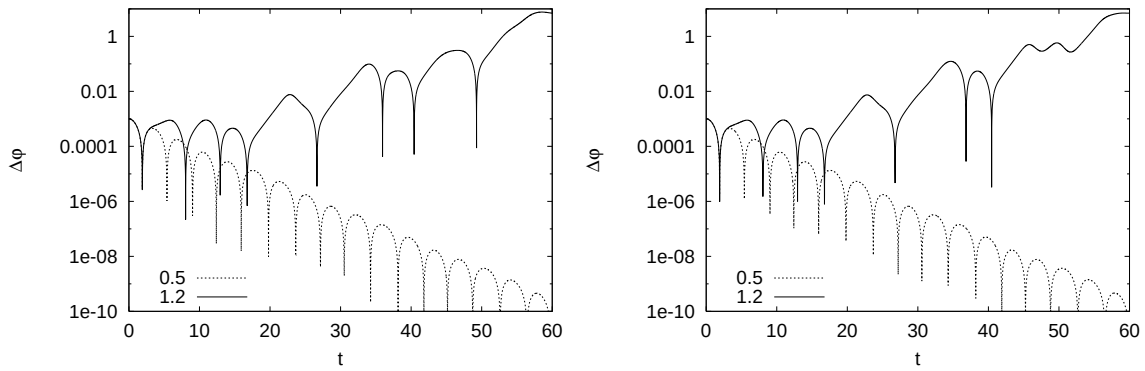


Abbildung 3.8: Damage Spreading: Der Abstand zwischen den Anfangsauslenkungen $\varphi_{01} = 0.2$ und $\varphi_{01} = 0.201$ wurde links mit Euler-Cromer, rechts mit Runge-Kutta berechnet.

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).

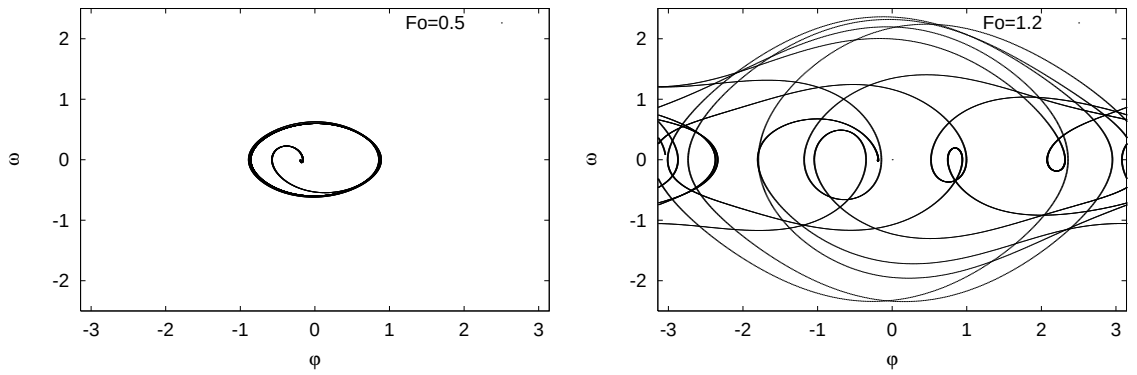


Abbildung 3.9: Phasendiagramme des allgemeinen Pendels, links für $F_0 = 0.5$, rechts für $F_0 = 1.2$.

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.

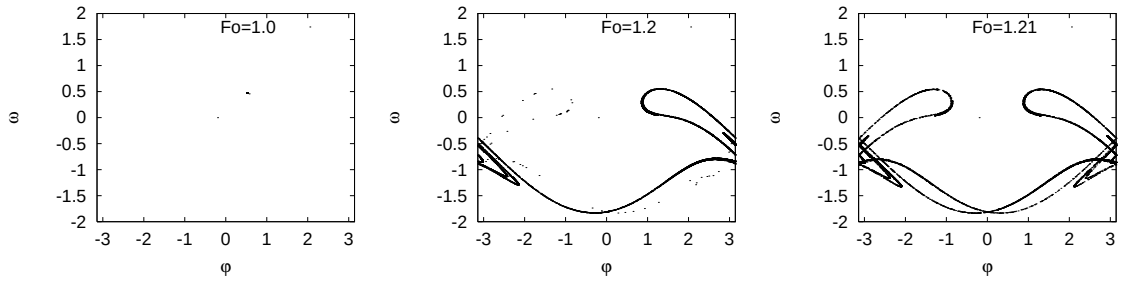


Abbildung 3.10: Poincaréschnitte für das allg. Pendel

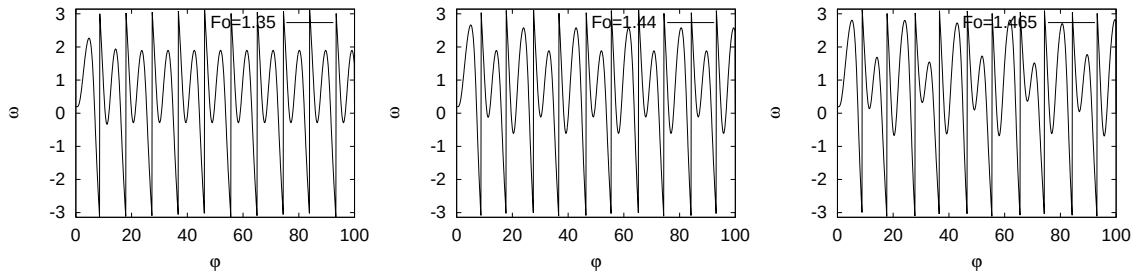


Abbildung 3.11: Unterschiedliche Perioden bei verschieden starken Antreibkräften

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

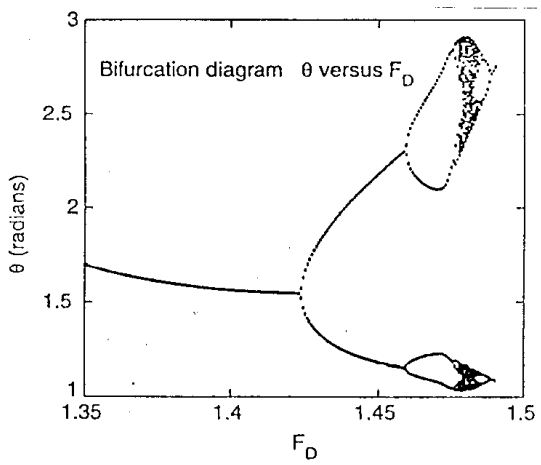


Abbildung 3.12: Bifurkationsdiagramm des Pendels

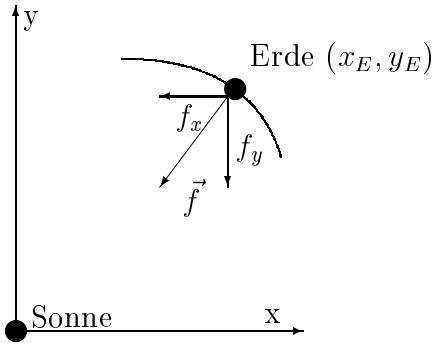


Abbildung 3.13: System Erde–Sonne

der Sonne (m_S) und der Masse der Erde (m_E). Die Formel für diese Kraft lautet :

$$\vec{f} = -\frac{Gm_S m_E}{r^2} * \frac{\vec{r}}{r} \tag{3.31}$$

Der Faktor G ist die Gravitationskonstante und der Vektor $\vec{r} = (x, y)$ 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

$$f_x = -\frac{Gm_S m_E x}{r^3} \ , \ f_y = -\frac{Gm_S m_E y}{r^3} \text{ und } r = \sqrt{x^2 + y^2} \tag{3.32}$$

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$\approx$ 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

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).

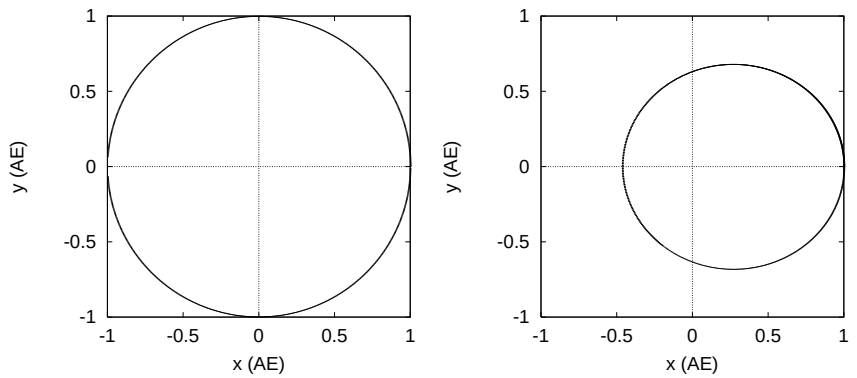


Abbildung 3.14: Bahnkurven der Erde um die Sonne für verschiedene Anfangsgeschwindigkeiten $vy_0 = 2\pi$ (links) und $vy_0 = 5$ (rechts).

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

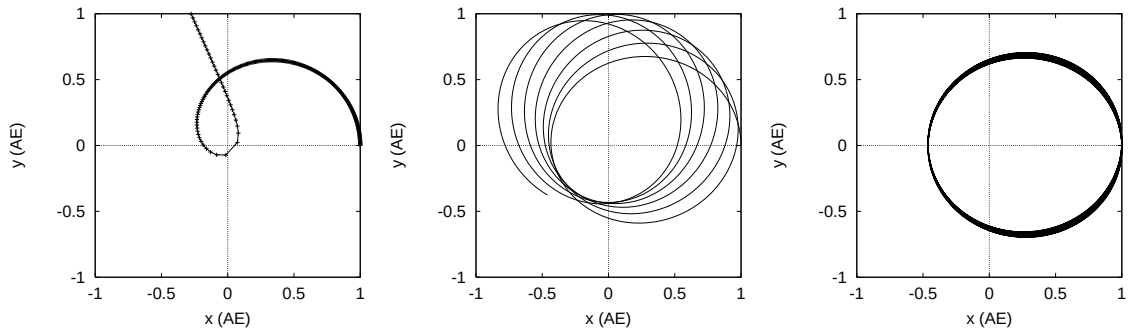


Abbildung 3.15: Bahnkurven bei versch. Gravitationsgesetzen $\beta = 3$ (links), $\beta = 2.1$ (mitte) und $\beta = 2.01$ (rechts).

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 $\vec{v} = \Delta\vec{r}/\Delta t$ 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 $\Delta E > 10^{-5} \rightarrow \Delta t' = \Delta t/2$ und bei $\Delta E < 10^{-7} \rightarrow \Delta t' = \Delta t * 2$ gewählt werden.

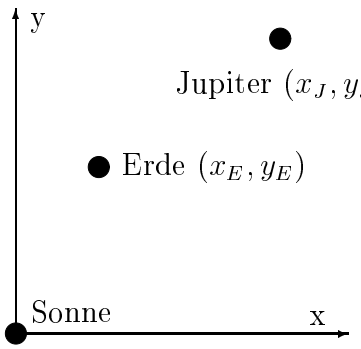


Abbildung 3.16: System Erde–Jupiter

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

$$Gm_E = 12 * \pi^2 * 10^{-6} \tag{3.46}$$

$$Gm_J = 4 * \pi^2 * 10^{-3} * M_J \tag{3.47}$$

berechnen. Als numerisches Verfahren verwenden wir wieder die Verlet-Methode. Die Gleichungen

$$x_E(t+\Delta t) = -x_E(t-\Delta t) + 2x_E(t) - \\ -\Delta t^2 4\pi^2 \left(\frac{x_E(t)}{r_E^3(t)} + \frac{10^{-3}(x_E(t) - x_J(t)) * M_J}{r_{EJ}^3(t)} \right) \tag{3.48}$$

$$x_J(t+\Delta t) = -x_J(t-\Delta t) + 2x_J(t) - \\ -\Delta t^2 4\pi^2 \left(\frac{x_J(t)}{r_J^3(t)} + \frac{3 * 10^{-6}(x_J(t) - x_E(t))}{r_{EJ}^3(t)} \right) \tag{3.49}$$

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!

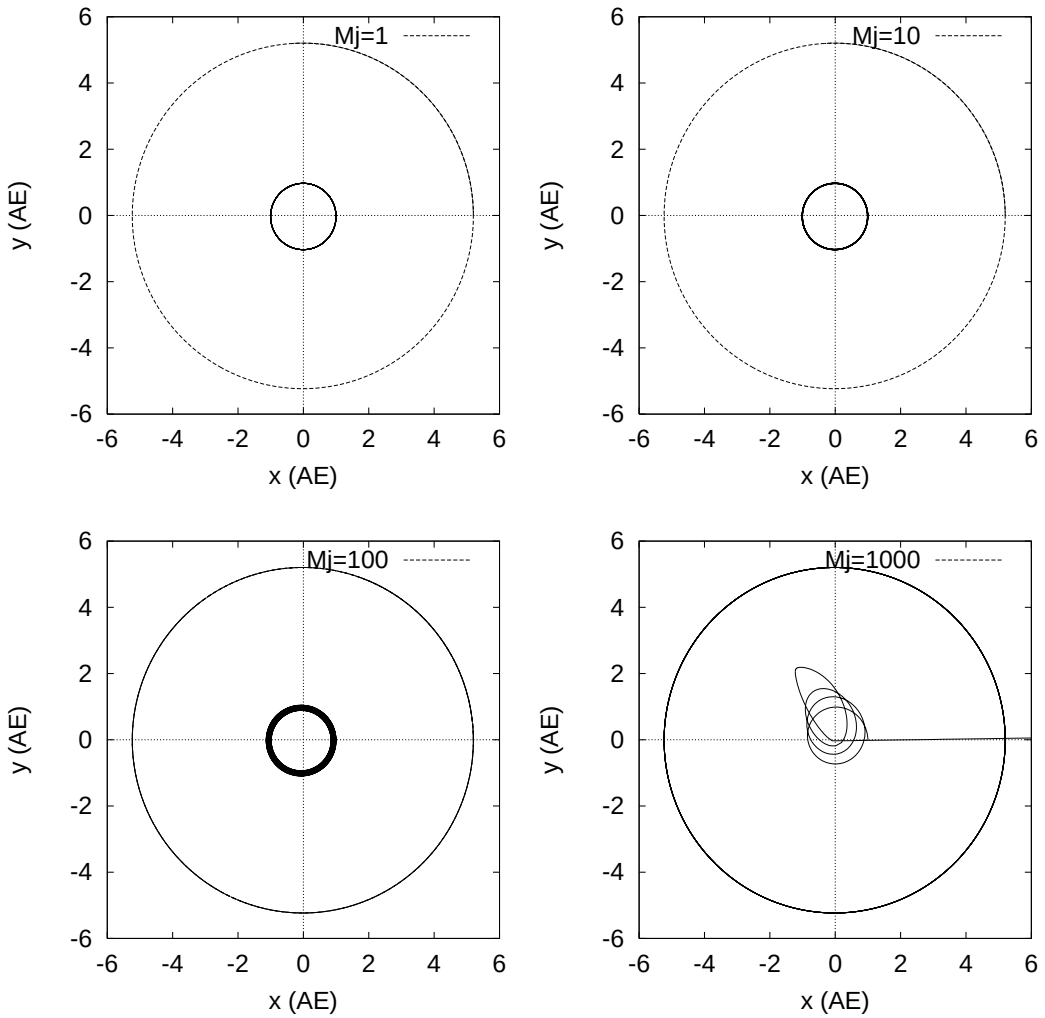


Abbildung 3.17: 2–Körper–Simulation mit verschiedenen Jupiter-Massen mit fixierter Sonne.

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.

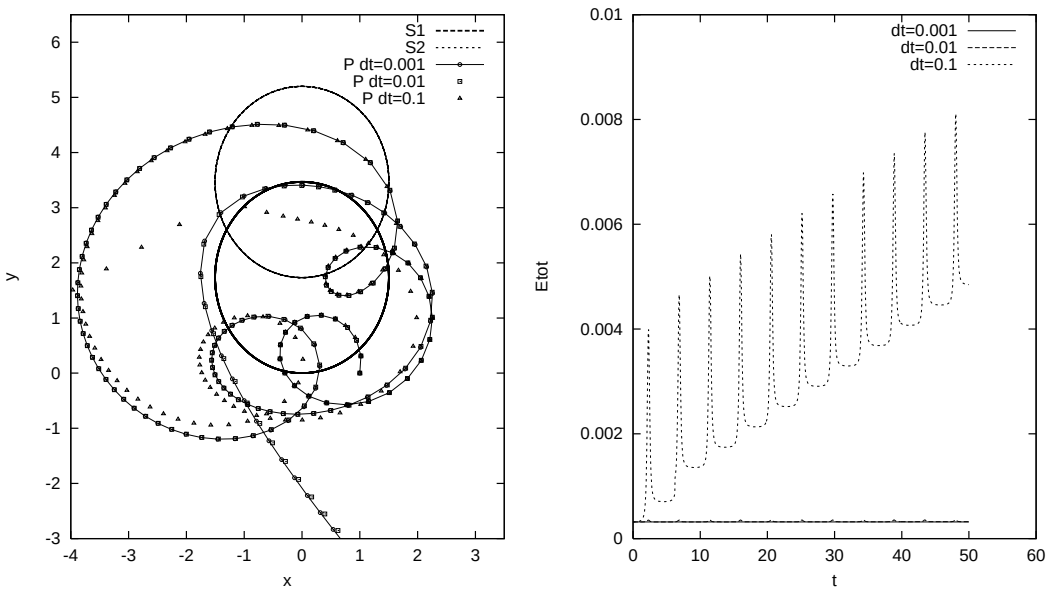


Abbildung 3.18: 3–Körper–Simulation mit Jupiter-Masse $M_j = 1000$ und beweglicher Sonne. Links: Trajektorien und rechts: Energie als Funktion der Zeit.

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 = $(1/3600)°$). 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:

$$f \approx \frac{G m_S m_M}{r_{SM}^2}(1 + \frac{\alpha}{r_{SM}^2}), \text{ mit } \alpha \approx 1.1 * 10^{-8}\text{AE}^2 \tag{3.52}$$

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 $\vec{r}$ und Geschwindigkeit $\vec{v}$ wird nun auch der Drehwinkel φ und die Winkelgeschwindigkeit ω verwendet. Das Bewegungsgesetz lautet:

$$\ddot{\varphi} = \dot{\omega} = \frac{M}{J} \text{ , mit} \tag{3.53}$$

$$J = \sum_i m_i(\vec{r}_i - \vec{r}_M)^2 \text{ (Trägheitsmoment)} \tag{3.54}$$

$$M = \sum_i (\vec{r}_i - \vec{r}_M) \times \vec{f_i} \text{ (Drehmoment)} \tag{3.55}$$

m_i sind die Massenpunkte, aus denen sich der Körper zusammensetzt, $\vec{r}_i$ ihre jeweiligen Koordinaten, f_i ihre anliegenden Kräfte und $\vec{r}_M$ 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.

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

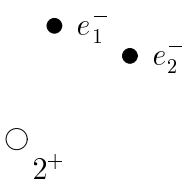


Abbildung 3.19: Elektronenbewegung im Heliumatom

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 :

$$f_1 = -2\frac{\vec{r}_1}{r_1^3} + \frac{\vec{r}_1 - \vec{r}_2}{r_{12}^3} \tag{3.56}$$

$$f_2 = -2\frac{\vec{r}_2}{r_2^3} + \frac{\vec{r}_2 - \vec{r}_1}{r_{12}^3} \tag{3.57}$$

$$\tag{3.58}$$

Durch die möglichen starken Kräfte ist hierbei ein adaptiver Zeitschritt notwendig. Die entstehende Bewegung ist wieder chaotisch.

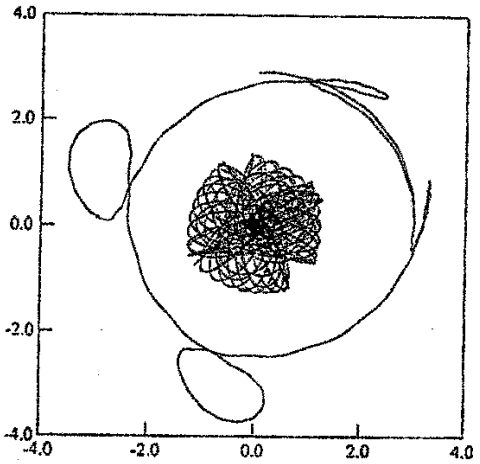


Abbildung 3.20: Bewegung der zwei Elektronen im Heliumatom.

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.

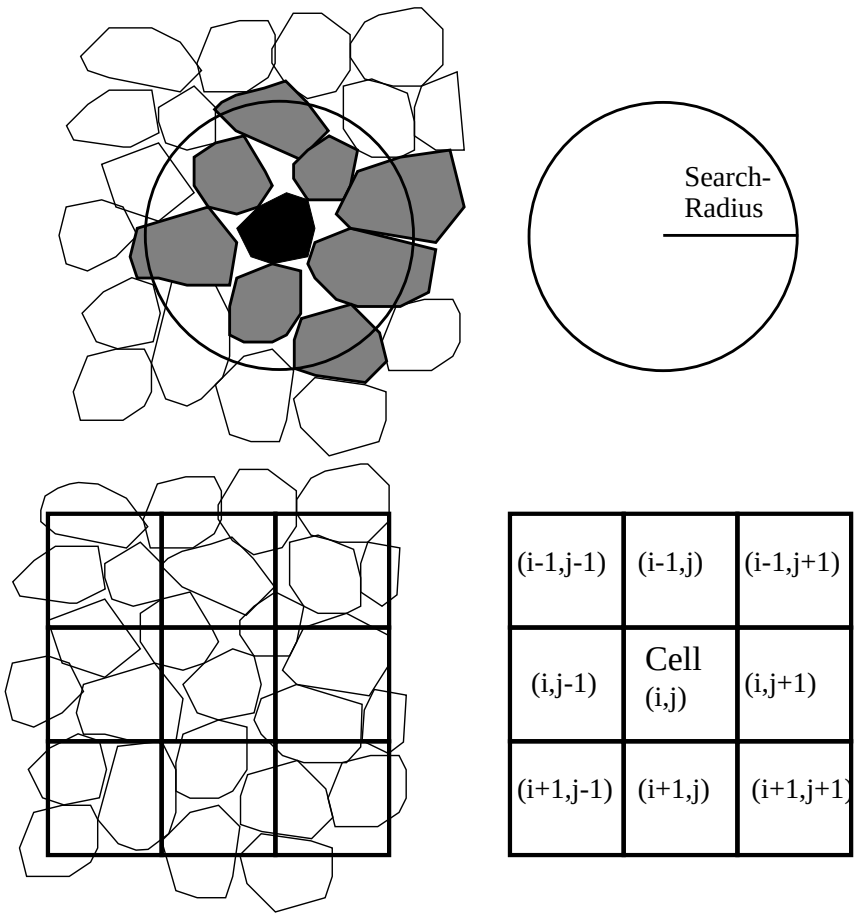


Abbildung 3.21: Verlet–Nachbarschaftstafeln (oben) und linked–cell (unten)

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.

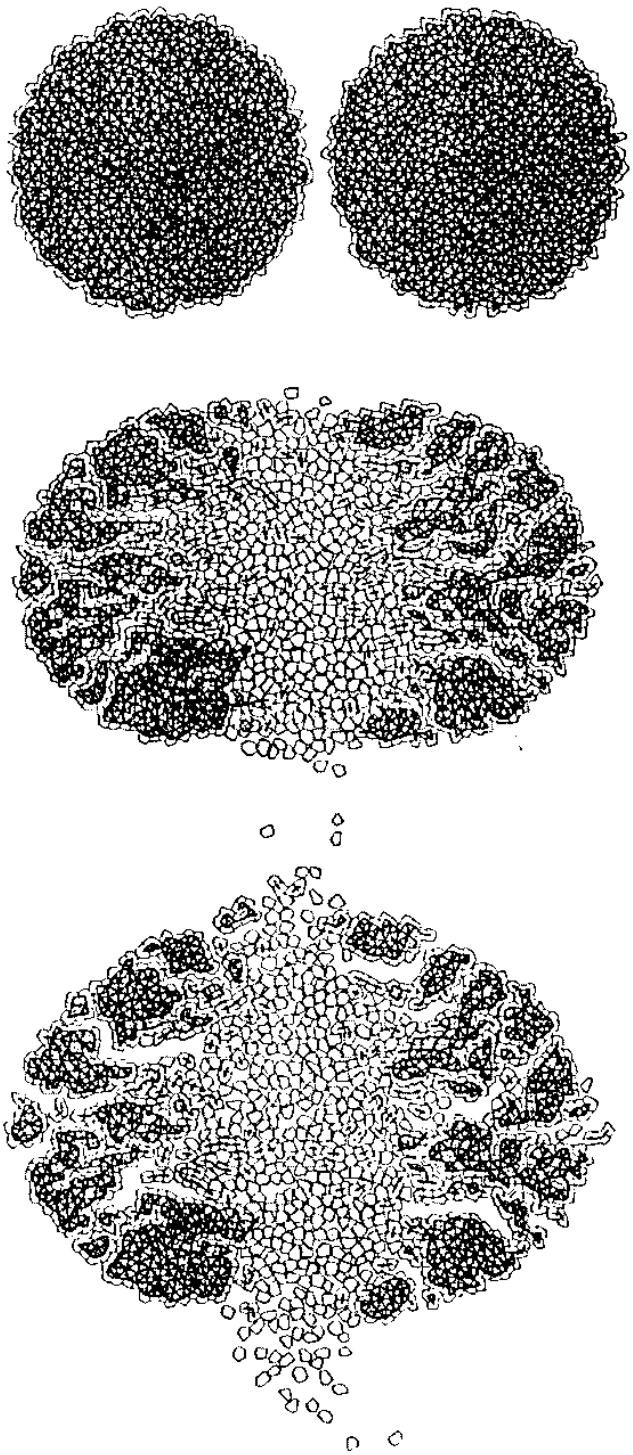


Abbildung 3.22: Zusammenstoß zweier Atomkerne

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.

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:

$$\frac{dy}{dt} = \mu y(1 - \frac{y}{K}) - v \ , \tag{4.8}$$

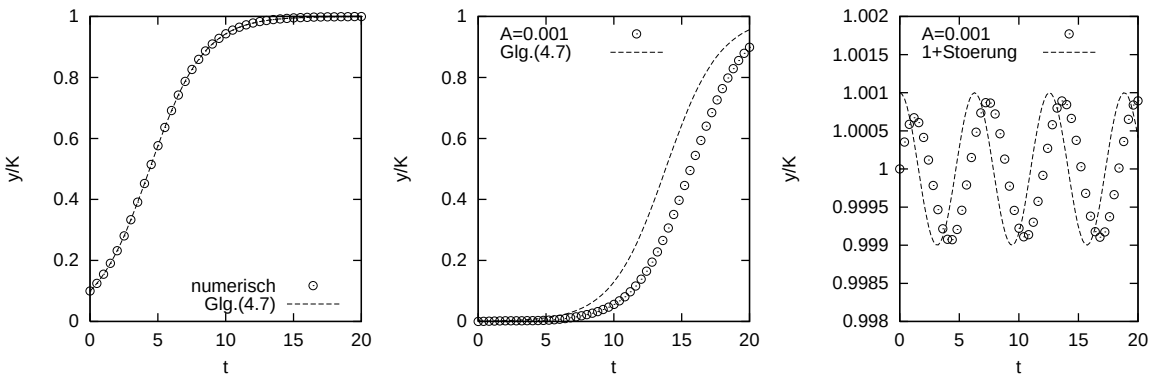


Abbildung 4.1: Numerische Lösung der Populationsgleichung 4.6 mit $\mu = 0.5$ und $K = 1$. Links: $y_0 = 0.1$, $A = 0$, Mitte: $y_0 = 0$, $A = 0.001$, Rechts: $y_0 = K$, $A = 0.001$.

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

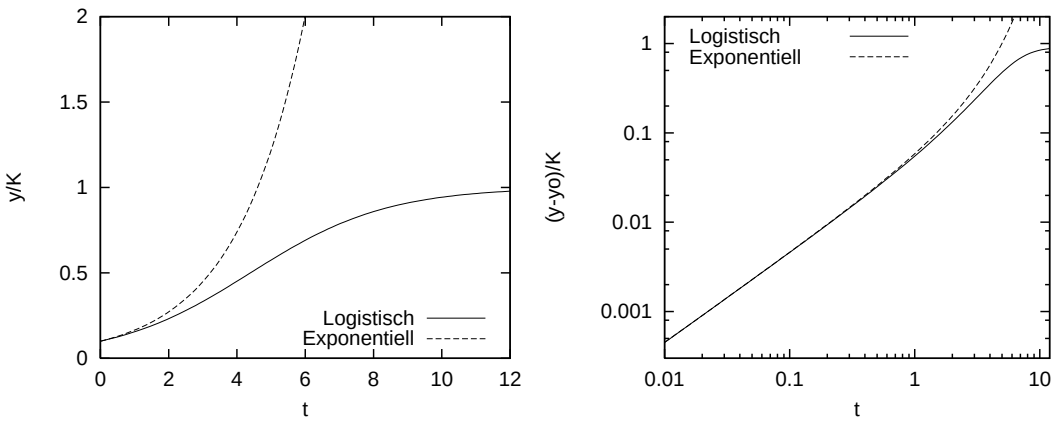


Abbildung 4.2: Vergleich von exponentiellem und logistischem Wachstum, Links: linear, Rechts: Halblogarithmisch. Gezeigt ist $(y(t) - y_0)/K$ aus Gleichung 4.6 (durchgezogene Linie) und exponentielles Wachstum $(1 - y_0/K)y_0(\exp(\mu t) - 1)$ (gestrichelte Linie).

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).

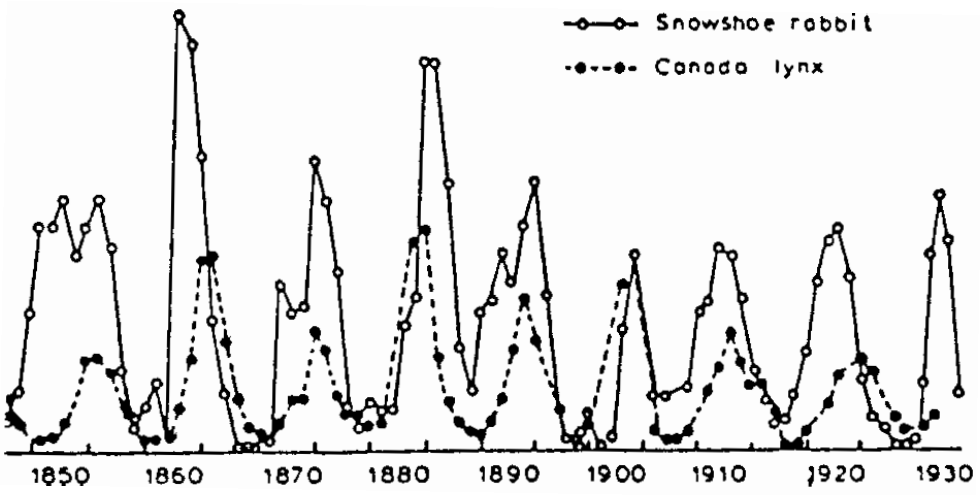


Abbildung 4.3: Räuber–Beute–System

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.

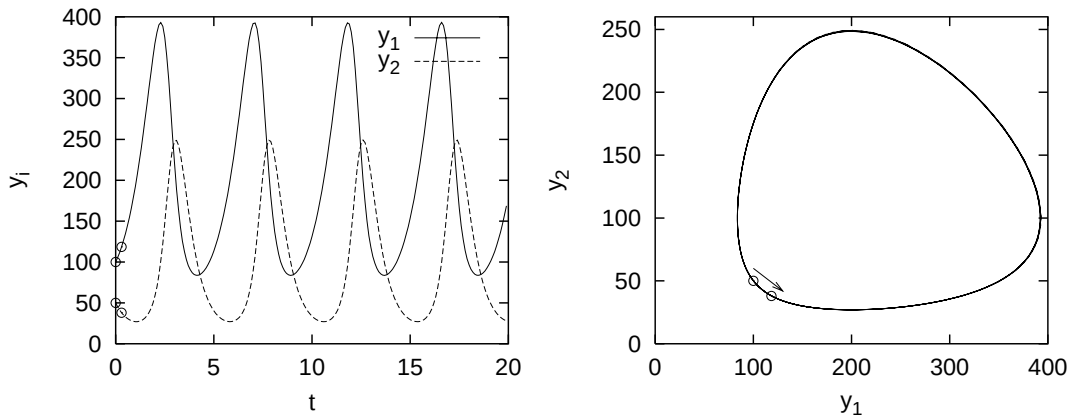


Abbildung 4.4: Räuber–Beute als Funktion der Zeit (links) und im Phasenraum (rechts). Gleichung 4.10 wurde mit den Parametern $b_1 = 1$, $m_2 = 2$, $r_{21} = r_{12} = 0.01$ und den Anfangswerten $y_1(0) = 100$ und $y_2(0) = 50$ numerisch gelöst. Die Kreise zeigen die Populationen zur Zeit $t = 0$ und $t = 0.3$ 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.

$$\begin{aligned} \frac{dy_1}{dt} &= (b_1 - r_{21}y_2 - \frac{y_1}{K})y_1 \\ \frac{dy_2}{dt} &= (-m_2 + r_{12}y_1)y_2 \end{aligned} \tag{4.11}$$

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.

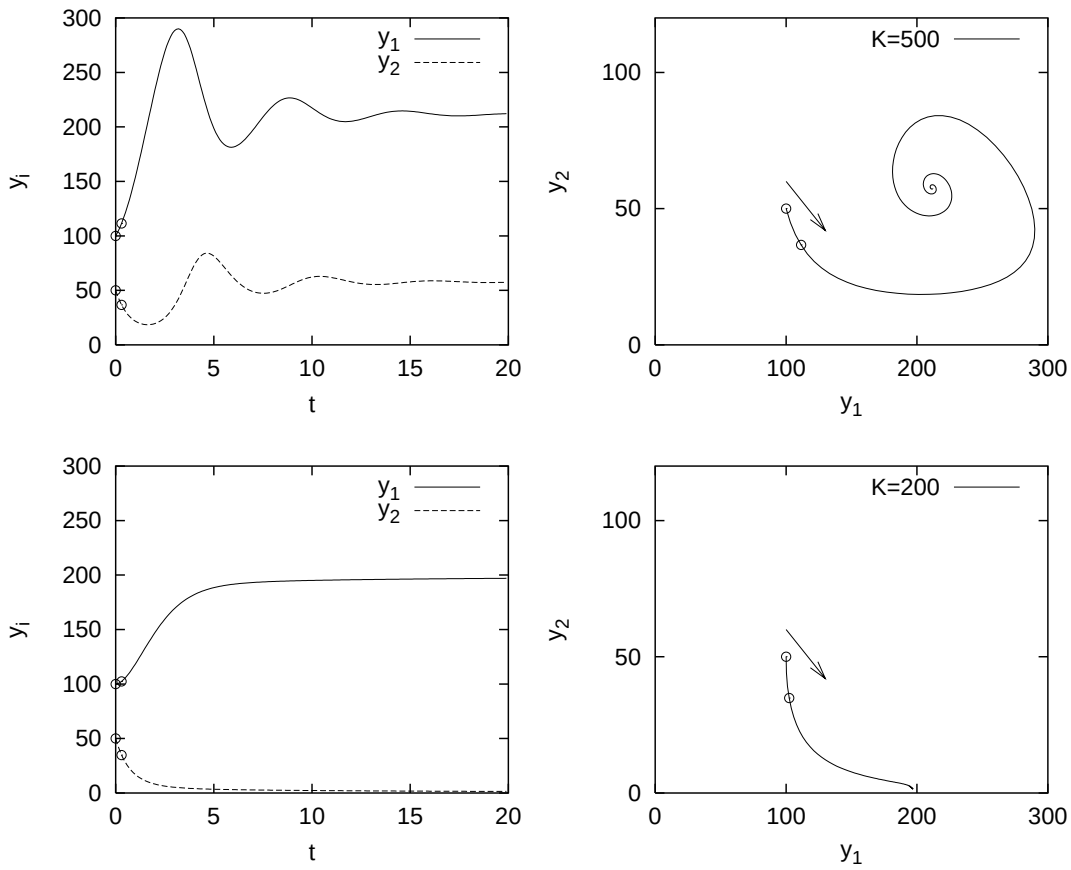


Abbildung 4.5: Räuber–Beute als Funktion der Zeit (links) und im Phasenraum (rechts). Gleichung 4.11 wurde mit den Parametern $b_1 = 1$, $m_2 = 2$, $r_{21} = r_{12} = 0.01$ und den Anfangswerten $y_1(0) = 100$ und $y_2(0) = 50$ numerisch gelöst. Oben hatte die Beutepopulation eine hohe ($K = 500$), unten 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 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()
{
```

```
   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.

$$y = \text{double}(x1 * a)/p \quad \Rightarrow \quad y \in]0, a] \tag{5.4}$$

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.

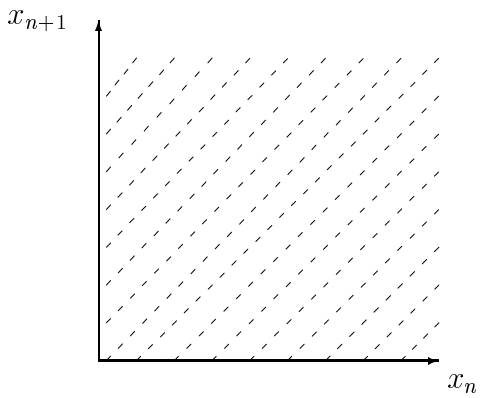


Abbildung 5.1: Squaretest (schematisch) für einen Zufallszahlengenerator.

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>
```

```
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).

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.

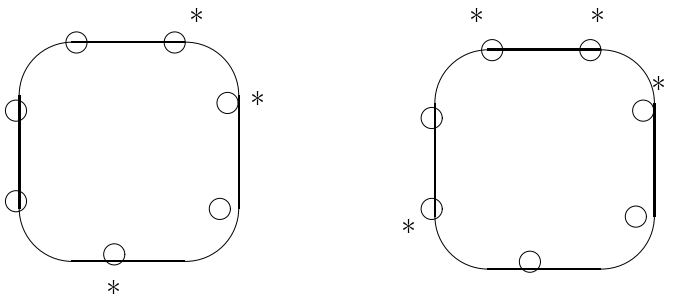


Abbildung 5.2: Beispiel der Anwendung der Regel 90.

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

$$f'(i) = f(f_r(i)) \oplus f(f_l(i)) \ . \tag{5.13}$$

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.

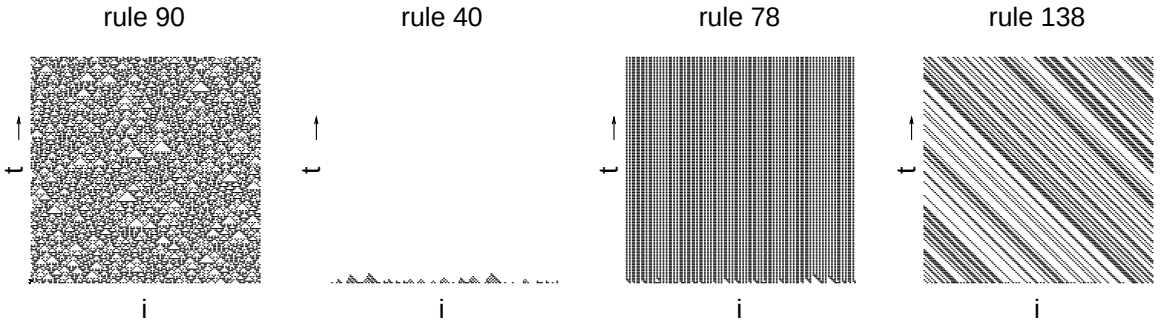


Abbildung 5.3: Jeweils 150 Zeit- und Ort-Schritte für die Regeln 90, 40, 78 und 138 bei zufälligen Anfangsbedinungen.

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:

...	0	0	0	0	0	**1**	0	0	0	0	0	...	
						1	0	**1**					
					1	0	0	0	**1**				
				1	0	**1**	0	**1**	0	**1**			
			1	0	0	0	0	0	0	0	**1**		
		1	0	**1**	0	0	0	0	0	**1**	0	**1**	
	1	0	0	0	**1**	0	0	0	**1**	0	0	0	**1**
1	0	**1**	0	**1**	0	**1**	0	**1**	0	**1**	0	**1**	0	**1**

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-

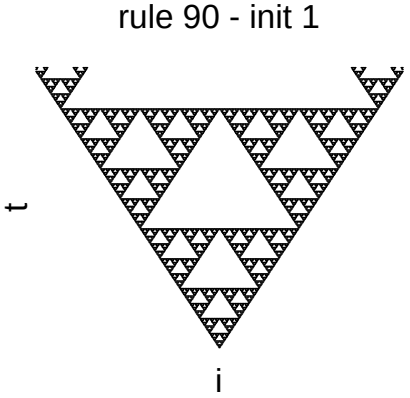


Abbildung 5.4: 300 Zeit-Schritte für die Regel 90 mit einer Eins als Anfangsbedinung.

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
```

```
         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.

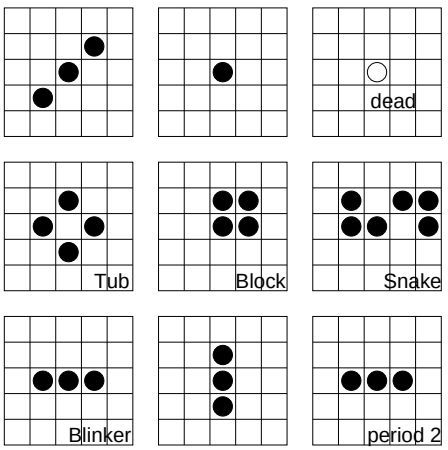


Abbildung 5.5: Aussterbende (oben), stabile (mitte) und periodische (unten) Figuren im Game of Life.

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).

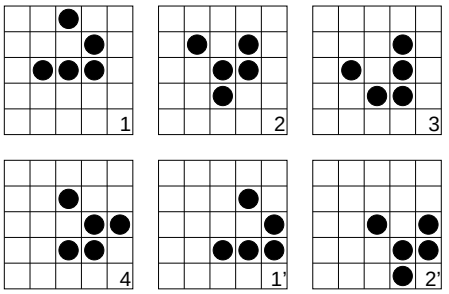


Abbildung 5.6: Die vier Phasen des Gleiters – verbunden mit einer Bewegung nach rechts unten.

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
```

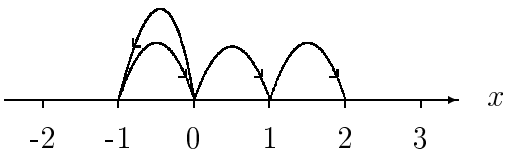


Abbildung 5.7: Ein-dimensionaler Zufallsweg: Start bei $x = 0$ in der Folge: l-r-r-r

```
    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.

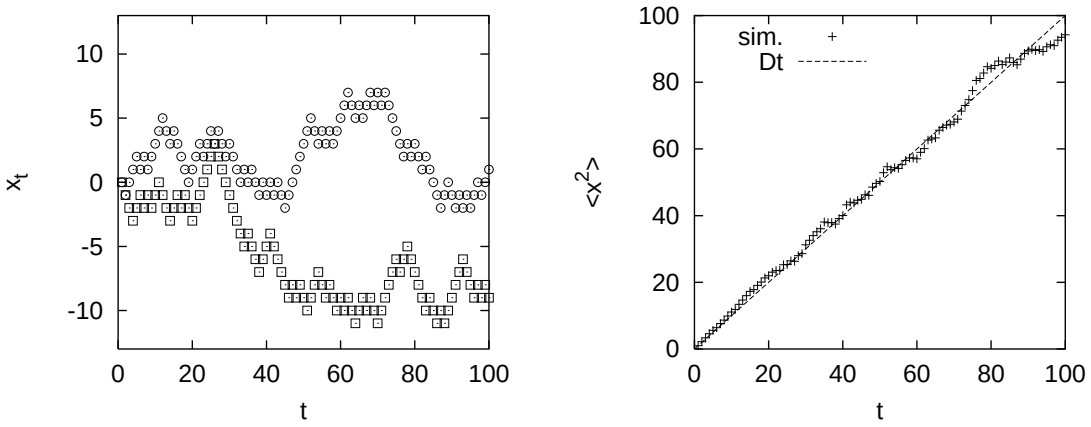


Abbildung 5.8: Position x_t von zwei verschiedenden random walks (links) und mittleres Verschiebungsquadrat $\langle x^2 \rangle$ im Mittel über 300 RW (rechts) als Funktion der Zeit.

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 $N = 1$:

$$x_t = \sum_{i=1}^{t} \gamma_i \tag{5.20}$$

$$\Rightarrow x_t^2 = \left(\sum_{i=1}^{n} \gamma_i \right)^2 = \sum_{i,j=1}^{n} \gamma_i \gamma_j \tag{5.21}$$

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:

$$x_t^2 = \sum_{i=1}^{t} \gamma_i^2 = t \ , \ \text{da} \ \gamma_i^2 \ = \ 1 \tag{5.22}$$

$$\Rightarrow D = \frac{x_t^2}{t} \ = \ 1 \ . \tag{5.23}$$

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.

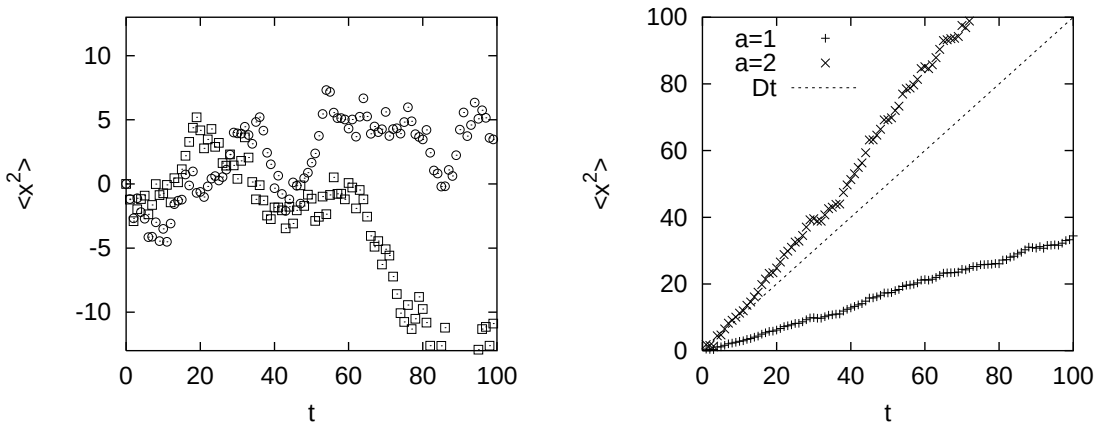


Abbildung 5.9: RW mit zufälliger Sprungweite und -richtung. Zwei Realisierungen (links) und $\langle x^2 \rangle$ im Mittel über 300 Läufe (rechts).

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:

$$\sigma^2 = 2Dt = 2\langle x_t^2 \rangle \tag{5.26}$$

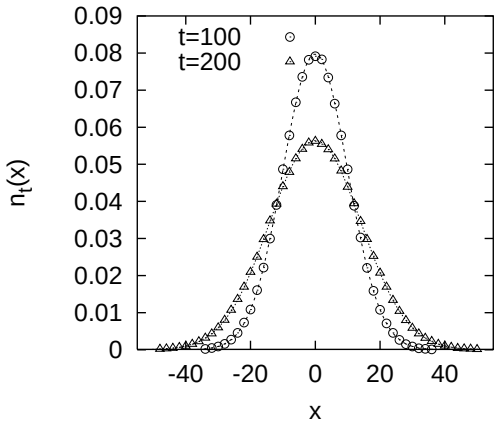


Abbildung 5.10: Gaußsche Glockenkurve der Verteilungen $2n_t(x)$. Symbole sind Daten der Simulation in Abb. 5.8, und die Linien stellen die entsprechenden Funktionen in Gleichung 5.25 dar.

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.

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:

$$\langle x\rangle_N = \langle M_i\rangle_k = \frac{1}{k}\sum_{i=1}^{k}\frac{1}{m}\sum_{j=1}^{m} x_{ij} \tag{5.39}$$

Würde man die Varianz der M_i als Fehler nehmen, so bekäme man:

$$\begin{aligned}\sigma_k^2 &= \langle (M_i - \langle M_i \rangle_k)^2 \rangle_k \\ &= \tfrac{1}{k} \sum_{i=1}^{k} (M_i - \langle x \rangle_N)^2 \\ &= \tfrac{1}{k} \sum_{i=1}^{k} \tfrac{1}{m^2} \sum_{j,n=1}^{m} (x_{ij} - \langle x \rangle_N)(x_{in} - \langle x \rangle_N) \\ &= \tfrac{1}{k} \sum_{i=1}^{k} \tfrac{1}{m^2} \sum_{j=1}^{m} (x_{ij} - \langle x \rangle_N)^2 \\ &= \tfrac{1}{m} \tfrac{1}{N} \sum_{i=1}^{k} \sum_{i=1}^{m} (x_{ij} - \langle x \rangle_N)^2 \\ &= \tfrac{1}{m} \sigma_N^2\end{aligned} \tag{5.40}$$

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

$$d_n = y_n - (mx_n + b) \ . \tag{5.44}$$

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:

$$\frac{\partial S}{\partial m} = \frac{-2}{N} \sum_{n=1}^{N} x_n (y_n - m x_n - b) = 0 \tag{5.46}$$

und

$$\frac{\partial S}{\partial b} = \frac{-2}{N} \sum_{n=1}^{N} (y_n - m x_n - b) = 0 \ . \tag{5.47}$$

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.

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.

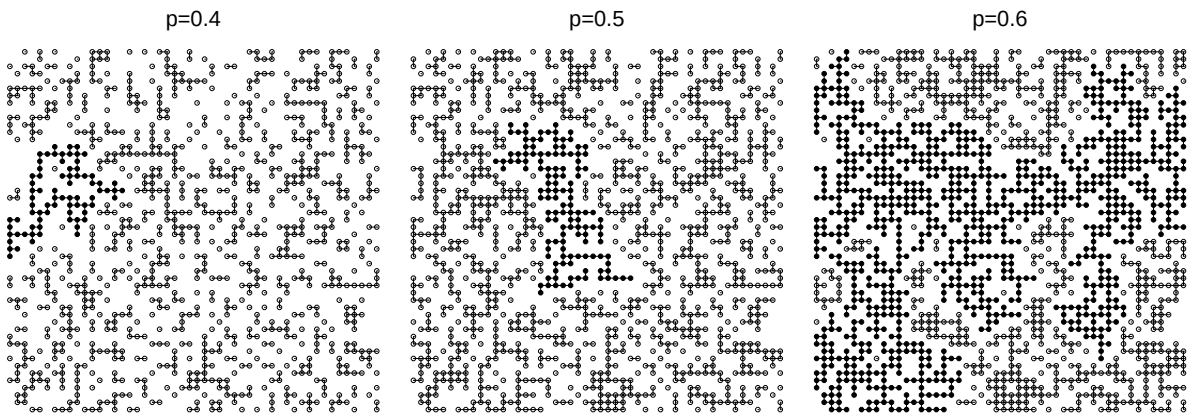


Abbildung 5.11: Perkolation mit verschiedenen Besetzungswahrscheinlichkeiten p. Besetzte Plätze sind als offene Kreise dargestellt, Verbindungen als Linien und die zum größten Cluster gehörenden besetzten Plätze als gefüllte Kreise.

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";
```

```
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
            }
```

```
        }   // 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, maxcnum=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
     maxcnum=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.

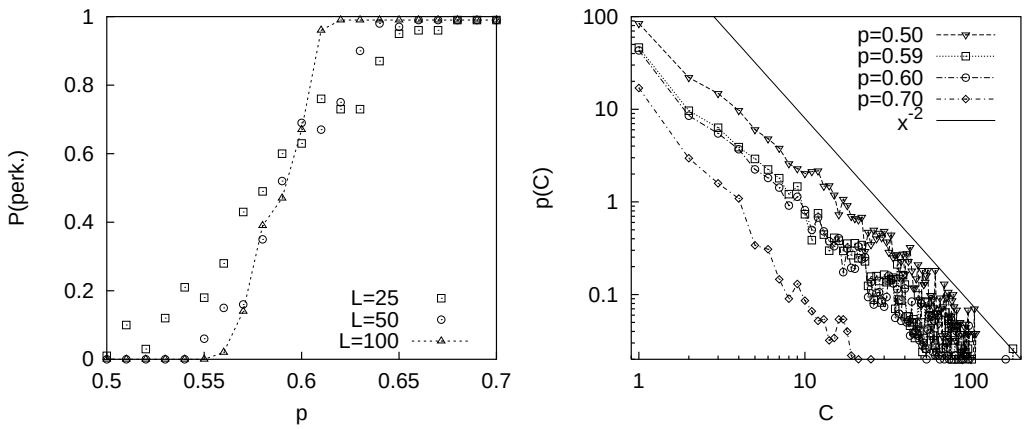


Abbildung 5.12: Perkolationswahrscheinlichkeit als Funktion der Besetzungswahrscheinlichkeit p für verschiedene Systemgrößen L, also für $N = L^2$ Gitterplätze (Links). Clustergrößenverteilung für verschiedene p (Rechts).

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.

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:

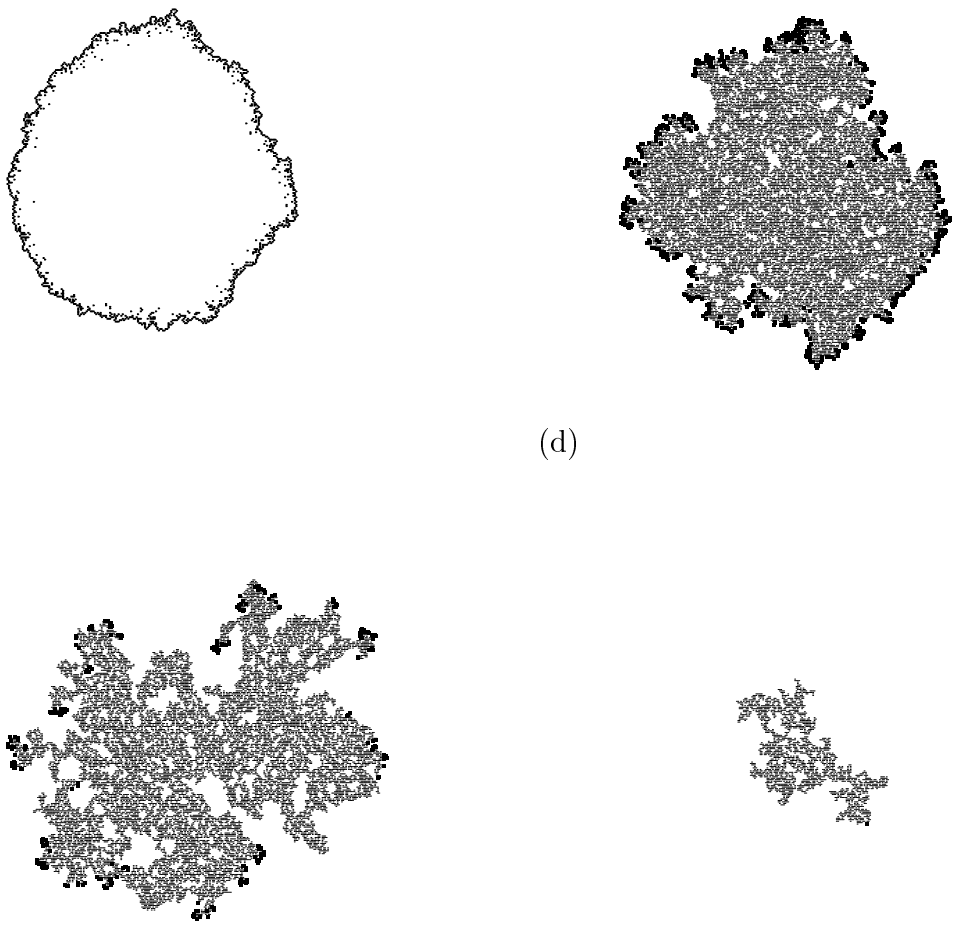
$$W = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(h_i - \langle h \rangle\right)^2} \ , \ \text{mit} \ \ \langle h \rangle = \frac{1}{N}\sum_{i=1}^{N} h_i \ . \tag{5.51}$$

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:

(a) (b)



(c) (d)

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()
```

```
{
  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++;
  }
```

```
  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).

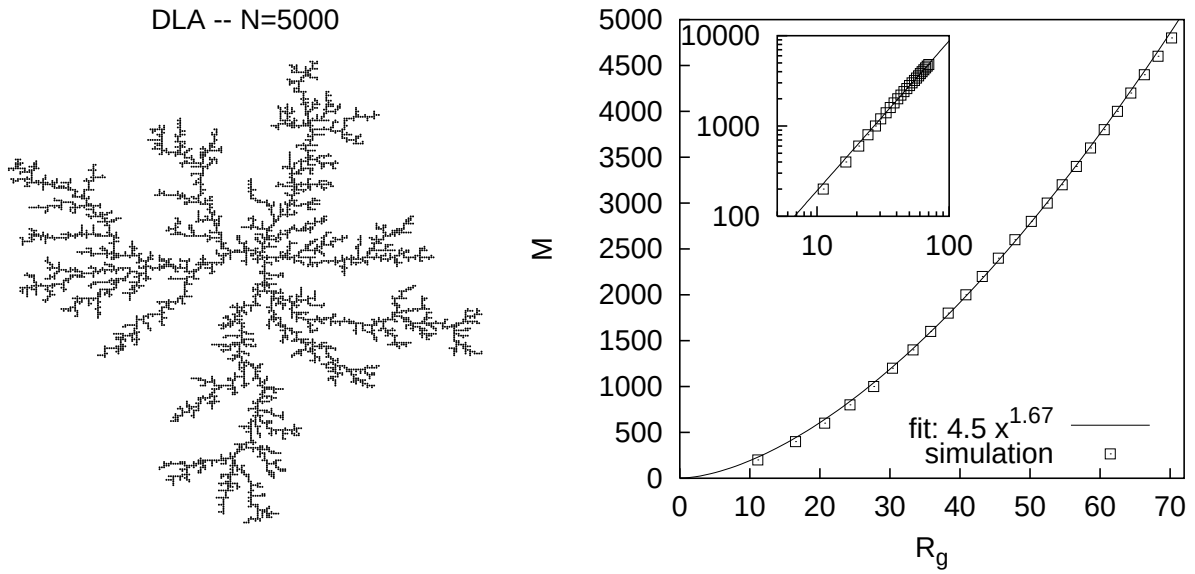


Abbildung 5.14: DLA Cluster mit $N = 5000$ Molekülen (Links). Masse als Funktion des Gyrationsradius R_g (Rechts) – auch in doppelt-logarithmischer Darstellung (Fenster).

Man kann die Ausdehnung des Objekts durch den sog. Gyrationsradius

$$R_g = \sqrt{(1/N)\sum_{i=1}^{N} (\vec{r}_i - <\vec{r}>_N)^2} \tag{5.52}$$

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.

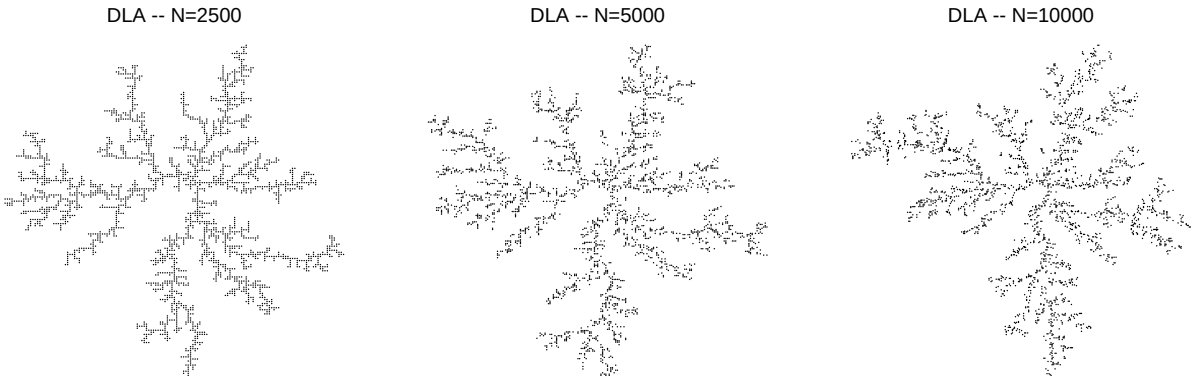


Abbildung 5.15: Verschieden große DLA Cluster mit $N = 2500$, 5000 und 10000 Molekülen. Die Vergrößerung ist so eingestellt daß die DLA Cluster in etwa die selben Flächen bedecken.

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:

```
#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.

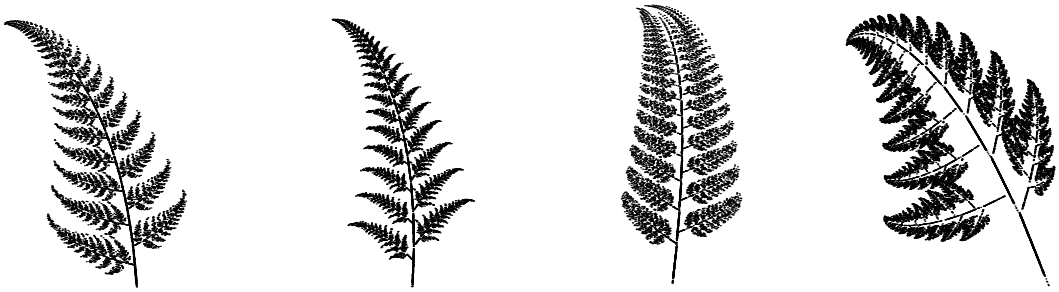


Abbildung 5.16: Vier verschiedene Farne – erzeugt durch verschiedene Iterationen.

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

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.

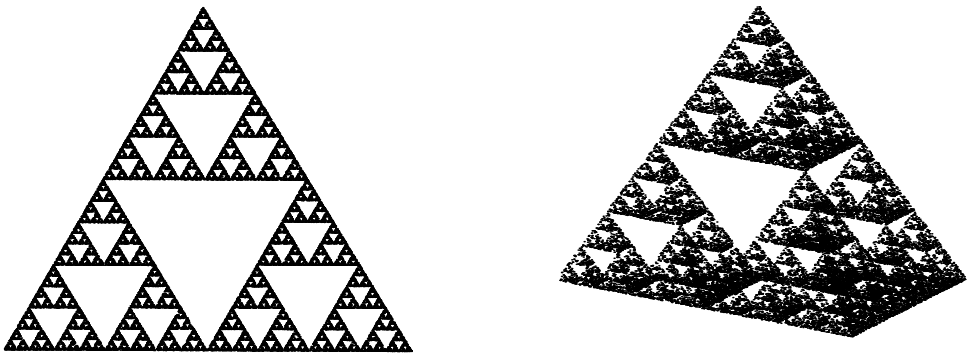


Abbildung 5.17: Das Sierpinski Gitter, erzeugt durch iterative Abbildung (links), und sein großer Bruder der durch vier hintereinandergeschaltete lineare Abbildungen erzeugt wurde.

```
#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];
```

```
   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'
   }
}
```

```
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;
}
```

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
```

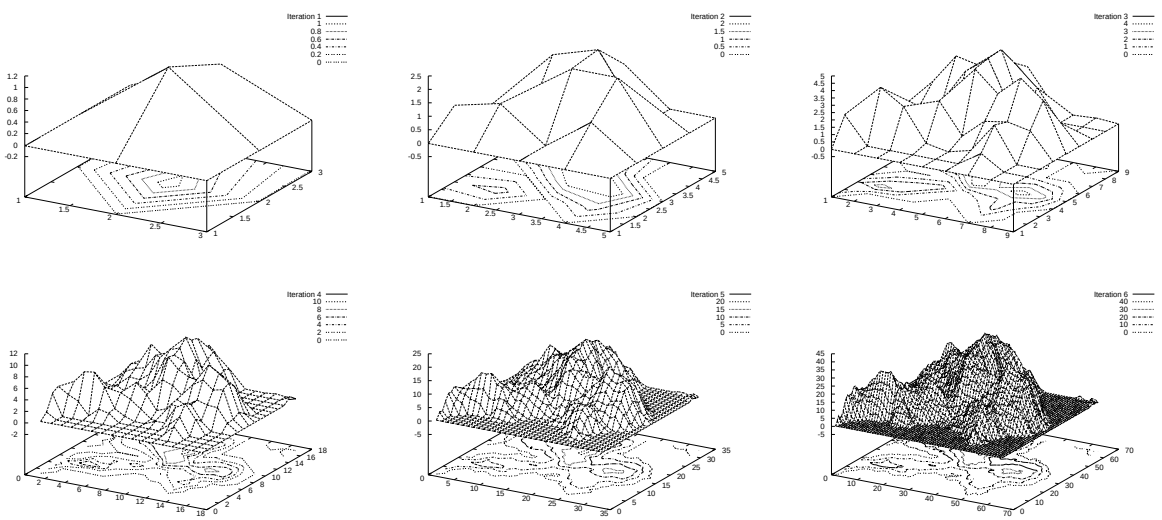


Abbildung 5.18: Random Midpoint displacement verschiedener Iterationstiefe.

Zum Abschluß dieses Kapitels (und zur Abschreckung) wird in diesem Abschnitt das Beispielprogramm in der Programmiersprache FORTRAN vorgestellt.

```
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
```

```
                  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
```