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.

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

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.

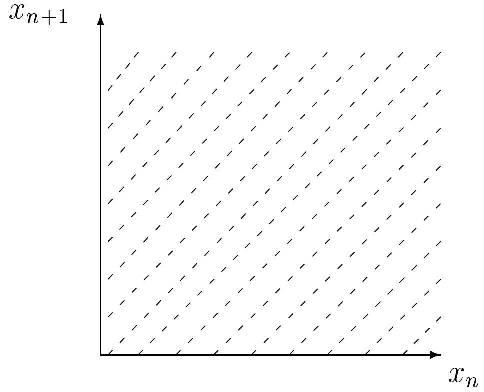


Abbildung 5.1: Squaretest (schematisch) für einen Zufallszahlengenerator.

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.

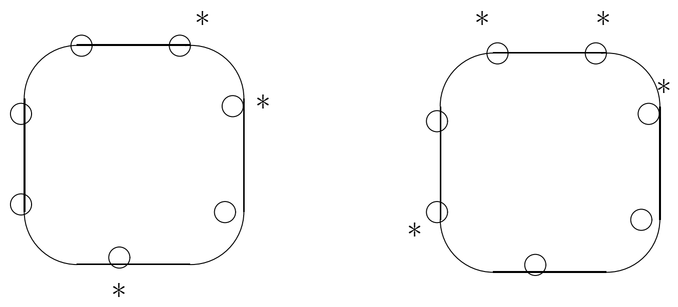


Abbildung 5.2: Beispiel der Anwendung der Regel 90.

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

$$f'(i) = f(i+1) \oplus f(i-1) \qquad \text{für } i = 2, \cdots, N-1, \tag{5.10}$$

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:

$$\begin{aligned} f_l(i) = i-1 \quad (i \geq 2), &\qquad \text{und} \quad f_l(1) = N \; , \\ f_r(i) = i+1 \quad (i \leq N-1), &\qquad \text{und} \quad f_r(N) = 1 \; . \end{aligned} \tag{5.12}$$

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.

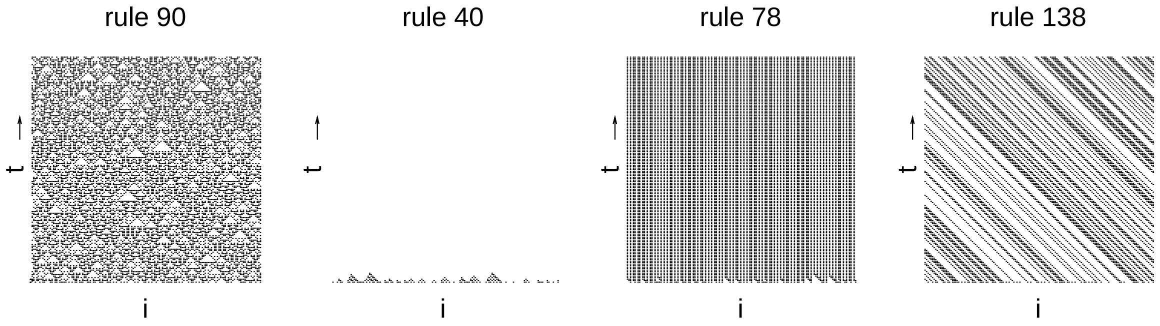


Abbildung 5.3: Jeweils 150 Zeit- und Ort-Schritte für die Regeln 90, 40, 78 und 138 bei zufälligen Anfangsbedinungen.

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-

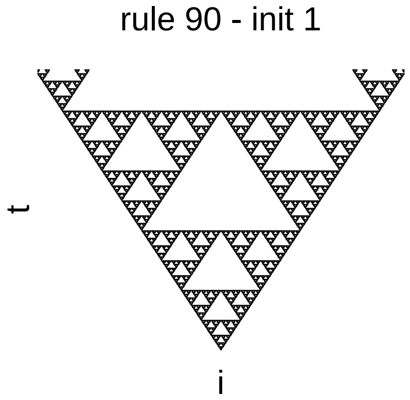


Abbildung 5.4: 300 Zeit-Schritte für die Regel 90 mit einer Eins als Anfangsbedinung.

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.

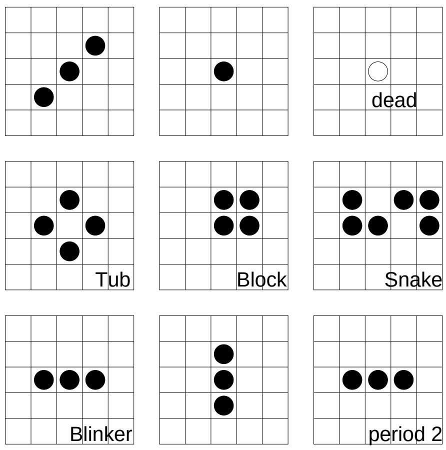


Abbildung 5.5: Aussterbende (oben), stabile (mitte) und periodische (unten) Figuren im Game of Life.

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).

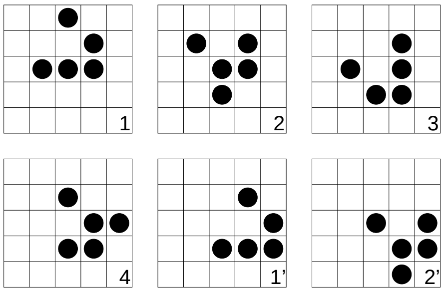


Abbildung 5.6: Die vier Phasen des Gleiters – verbunden mit einer Bewegung nach rechts unten.

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
```

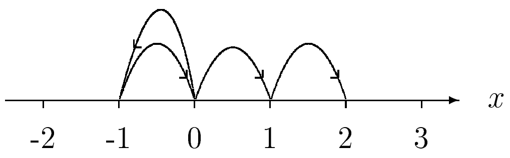


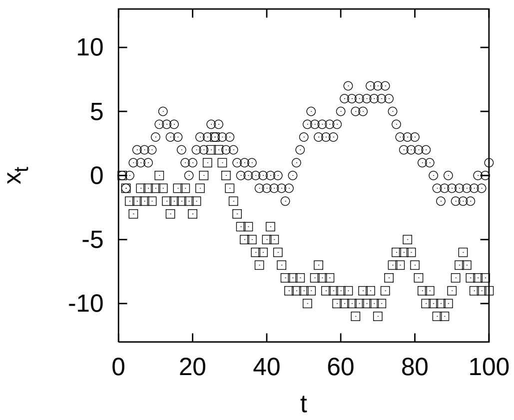
Abbildung 5.7: Ein-dimensionaler Zufallsweg: Start bei $x = 0$ in der Folge: l-r-r-r

```
    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.



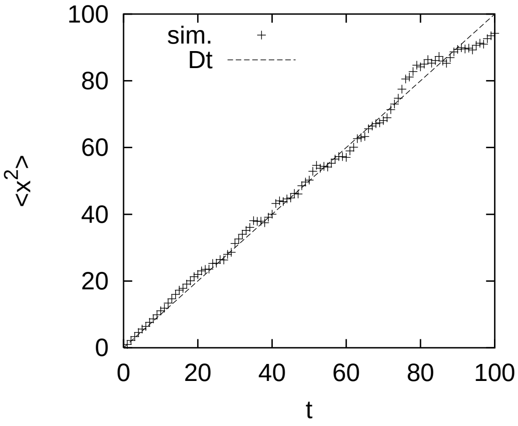


Abbildung 5.8: Position x_t von zwei verschiedenden random walks (links) und mittleres Verschiebungsquadrat $\langle x^2 \rangle$ im Mittel über 300 RW (rechts) als Funktion der Zeit.

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:

$$r \hat{=} \sqrt{\langle x^2 \rangle} \ \sim \sqrt{t} \tag{5.19}$$

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:

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

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

Die ganze Bewegung kann auch dreidimensional realisiert werden. Dafür werden alle Berechnungen für x nochmals unabhängig für die beiden anderen Richtungen y und z durchgeführt. Die Summe der Quadrate wird zur quadratischen Entfernung vom Ursprung

$$r^2 = x^2 + y^2 + z^2 \tag{5.24}$$

Diese Entfernung ist immer noch proportional zur Wurzel der Zeit.

Eine andere Variationsmöglichkeit ist die Implementierung einer zufälligen Weglänge mit maximaler Sprungweite a. Anstatt einer dualen Zufallszahl für den random walk benutzen wir eine reelle Zufallszahl, beispielsweise zwischen $-a$ und a. Abbildung 5.9 (links) zeigt diesen random walk mit zufälliger Weglänge und $a = 2$. In der rechten Grafik ist das mittlere Quadrat der Entfernung aufgetragen und es ist zu erkennen, daß sie wieder proportional zur Zeit ist, aber diesmal mit kleinerem Faktor.

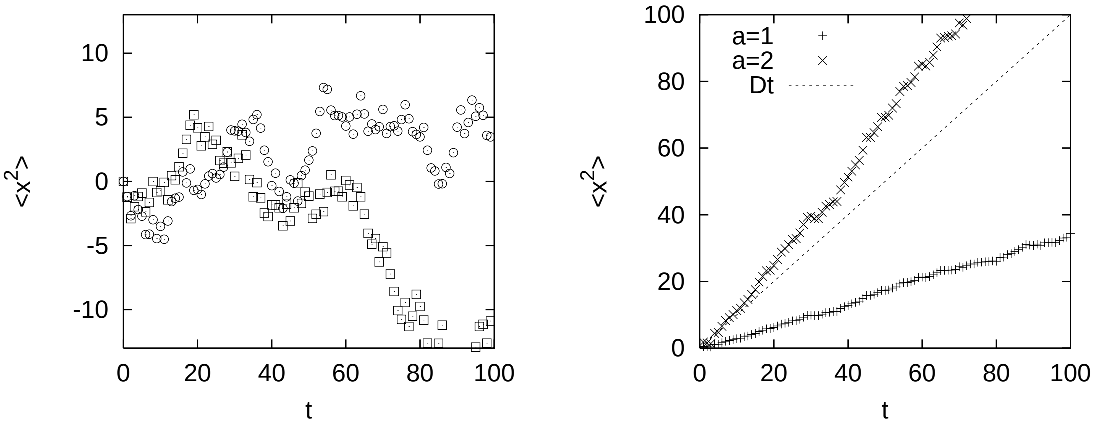


Abbildung 5.9: RW mit zufälliger Sprungweite und -richtung. Zwei Realisierungen (links) und $\langle x^2 \rangle$ im Mittel über 300 Läufe (rechts).

Trägt man zur Zeit t die Anzahl $N_t(x)$ der Teilchen (von verschiedenen RW) an einem Ort x auf, so entsteht eine Gaußsche Glockenkurve, siehe Abbildung 5.10. Das bedeutet daß $N_t(x)$ normalverteilt ist:

$$n_t(x) = \frac{N_t(x)}{\sum_x N_t(x)} = \frac{1}{\sqrt{\pi}\sigma} \mathrm{e}^{-\frac{(x-\mu)^2}{\sigma^2}} \ . \tag{5.25}$$

Hierbei ist der Mittelwert $\mu = \langle x_t \rangle = 0$. Für die Standardabweichung σ, also die Breite der Normalverteilung, gilt:

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

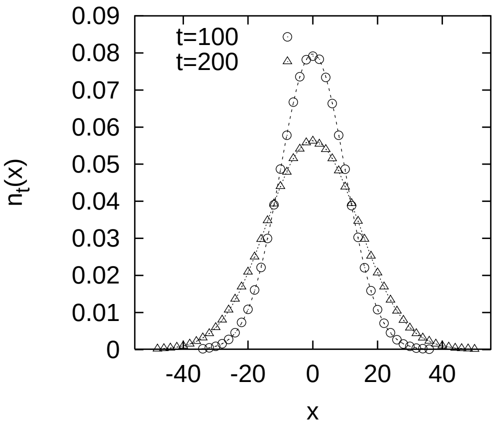


Abbildung 5.10: Gaußsche Glockenkurve der Verteilungen $2n_t(x)$. Symbole sind Daten der Simulation in Abb. 5.8, und die Linien stellen die entsprechenden Funktionen in Gleichung 5.25 dar.

Da die diskrete Simulation zur Gewinnung der Daten verwendet wurde, können bei geraden Zeiten t auch nur gerade Orte x besetzt sein. Zur Zeit $t = 0$ hat die Funktion $n_t(x)$ nur bei $x = 0$ den Wert $n_t(0) = 1$, überall sonst ist verschwindet die Funktion. Im Laufe der Zeit läuft die Funktion auseinander und wird flacher, die Fläche unter $n_t(x)$ beleibt durch die Definition erhalten.

Das mittlere Abstandquadrat eines RW hat also die Bedeutung der Breite der Verteilungsfunktion, die ebenso proportional zur Zeit ist:

$$\sigma \sim t \ . \tag{5.27}$$

Ein RW entfernt sich also mit der Zeit immer weiter von seinem Ausgangspunkt, kehrt jedoch immer wieder zurück, so daß die mittlere Position immer der Ausgangspunkt ist. Der RW unternimmt also immer längere Reisen zur Erkundung des Raumes.

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.

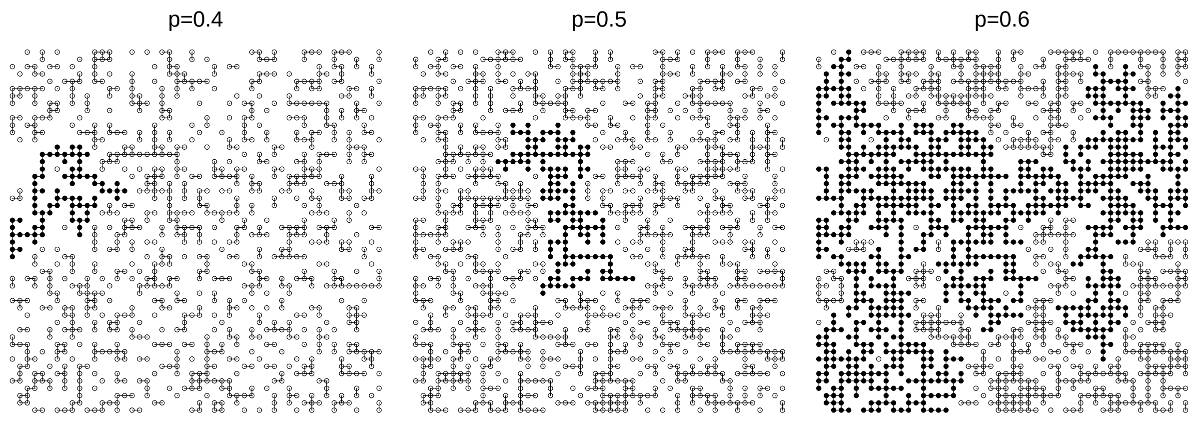


Abbildung 5.11: Perkolation mit verschiedenen Besetzungswahrscheinlichkeiten p. Besetzte Plätze sind als offene Kreise dargestellt, Verbindungen als Linien und die zum größten Cluster gehörenden besetzten Plätze als gefüllte Kreise.

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.

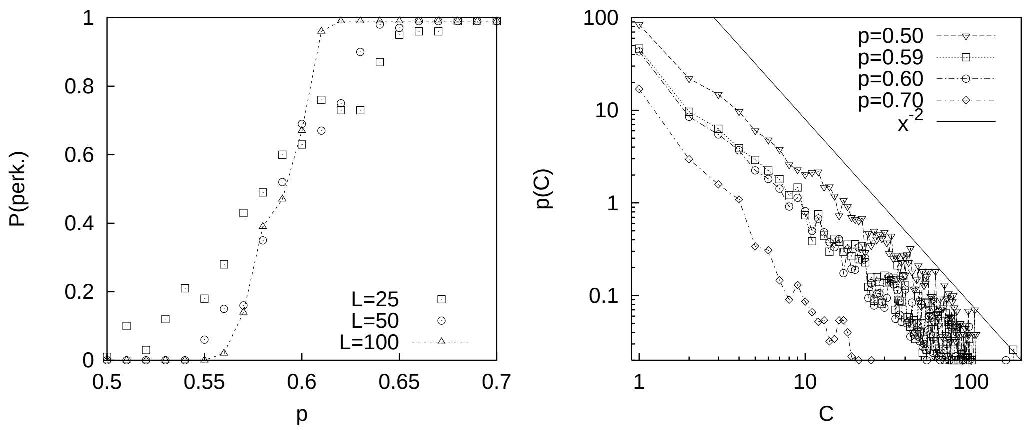


Abbildung 5.12: Perkolationswahrscheinlichkeit als Funktion der Besetzungswahrscheinlichkeit p für verschiedene Systemgrößen L, also für $N = L^2$ Gitterplätze (Links). Clustergrößenverteilung für verschiedene p (Rechts).

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:

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

Dieses Verfahren kann zur Simulation des Wachstums von Pflanzen und Kristallen verwendet werden.

Bei einer Epidemie können Zellen gegen Krankheitserreger immun werden. An solchen Stellen kann sich die Krankheit nicht weiter ausbreiten. Ist die Wahrscheinlichkeit p, daß eine Zelle immun wird zu groß, so hört das Wachstum ganz auf, da alle Zellen an der Oberfläche immun geworden sind. Je nach Modell kann man wieder die kritische Wahrscheinlichkeit p_c aus dem letzten Abschnitt entdecken. Bei Werten von p in der Nähe von p_c entstehen durch die immunen Zellen immer größere Löcher im Cluster. Große Löcher bedeutet eine fraktale Struktur, da die quadratische Ausdehnung des Clusters stärker anwächst als seine Masse, siehe Abb. 5.13.

Das zur Erzeugung verwendete Programm sieht folgendermassen aus:

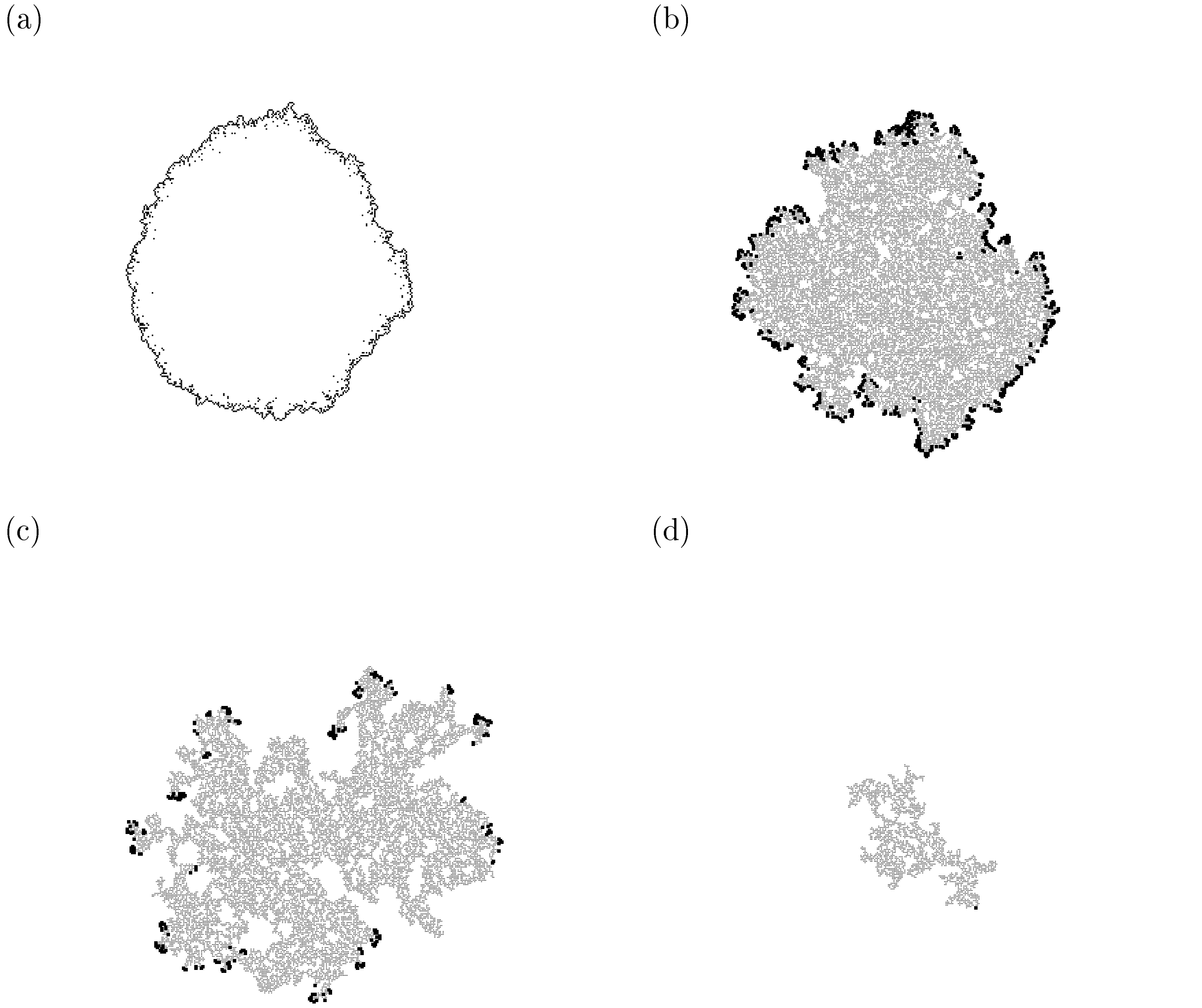


Abbildung 5.13: Eden-Wachstumsmodell mit Immunitätsraten (a) $p = 0$, (b) $p = 0.35$, (c) $p = 0.40$ und (d) $p = 0.415$.

```
#include<iostream>
#include<fstream>
using std::cout;
using std::cin;
#include"rng2.h"

int newsurf( int i, int j );
static const int NL=500;
static int world[NL+2][NL+2];    // define the lattice
static int surface[NL*NL][2], nums=0;
static int *psurf[NL*NL];

int main()
```

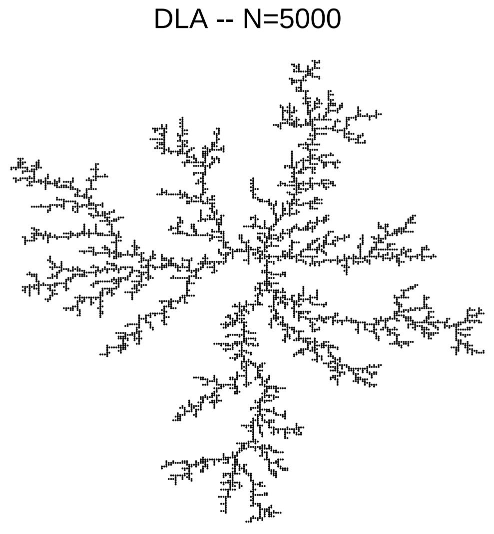
```
{
  int NGROW=30000;
  float prob;
  int igrow;

  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).



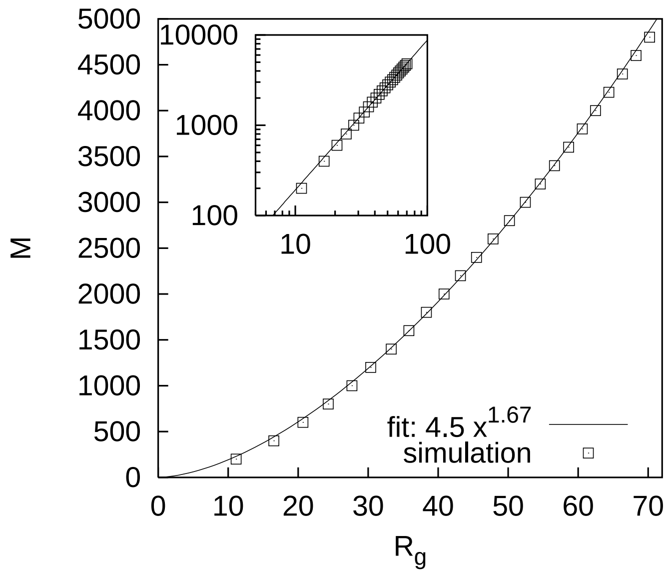


Abbildung 5.14: DLA Cluster mit $N = 5000$ Molekülen (Links). Masse als Funktion des Gyrationsradius R_g (Rechts) – auch in doppelt-logarithmischer Darstellung (Fenster).

Man kann die Ausdehnung des Objekts durch den sog. Gyrationsradius

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

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.

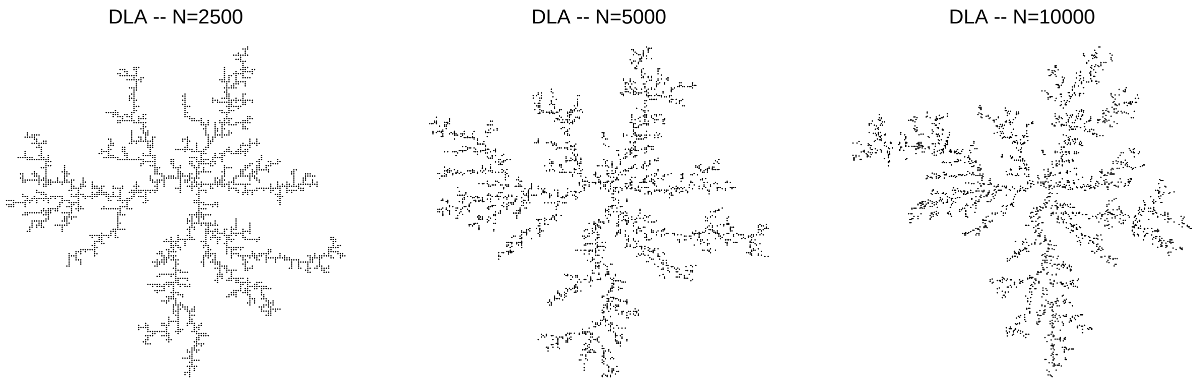


Abbildung 5.15: Verschieden große DLA Cluster mit $N = 2500$, 5000 und 10000 Molekülen. Die Vergrößerung ist so eingestellt daß die DLA Cluster in etwa die selben Flächen bedecken.

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.

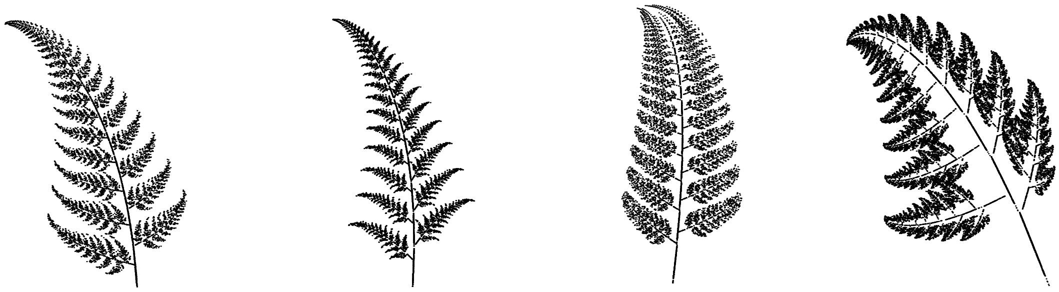


Abbildung 5.16: Vier verschiedene Farne – erzeugt durch verschiedene Iterationen.

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 **r** der Ausgangspunkt (x, y) und **r**' der seine Abbildung ist. Die Abbildung ist zusammengesetzt aus einem Verschiebungsvektor **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.

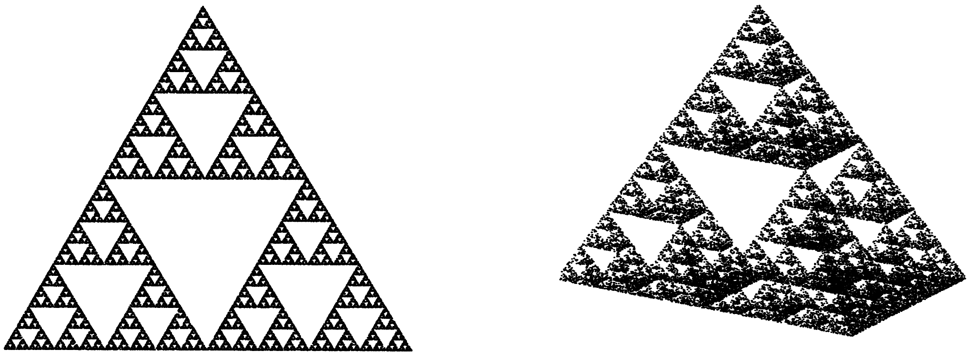


Abbildung 5.17: Das Sierpinski Gitter, erzeugt durch iterative Abbildung (links), und sein großer Bruder der durch vier hintereinandergeschaltete lineare Abbildungen erzeugt wurde.

```
#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
```

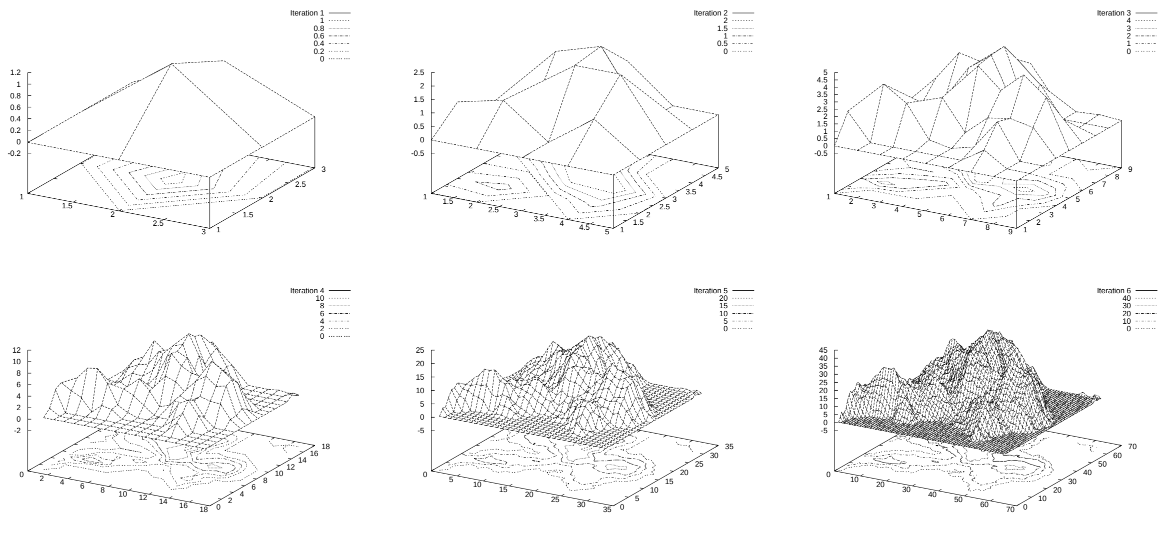


Abbildung 5.18: Random Midpoint displacement verschiedener Iterationstiefe.

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
```