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 `int`egers mit Überlauf zu verwenden kann man auf 64-bit Maschinen `long int`egers 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.

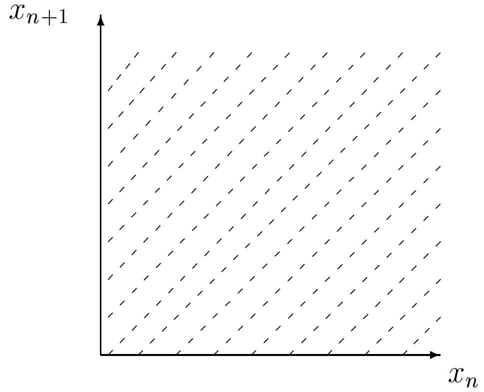

Abbildung 5.1: Squaretest (schematisch) für einen Zufallszahlengenerator.

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).

$$x_{n+1} = (x_{n-a} + x_{n-b})\text{mod}2 \quad \hat{=}\ x_{n-a} \oplus x_{n-b}\ . \tag{5.5}$$

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.

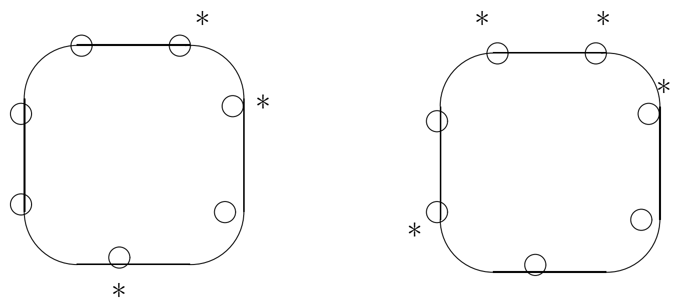

Abbildung 5.2: Beispiel der Anwendung der Regel 90.

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.

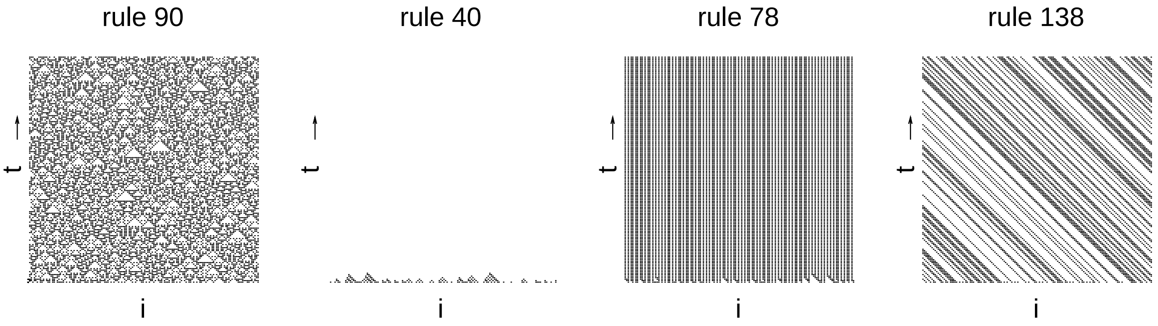

Abbildung 5.3: Jeweils 150 Zeit- und Ort-Schritte für die Regeln 90, 40, 78 und 138 bei zufälligen Anfangsbedinungen.

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:

$$
\begin{array}{ccccccccccccccc}
 & \ldots & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & \ldots & \\
 & & & & & & \mathbf{1} & 0 & \mathbf{1} & & & & & & \\
 & & & & & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & & & & & \\
 & & & & \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} & & & & \\
 & & & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & & & \\
 & & \mathbf{1} & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & \mathbf{1} & & \\
 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & \\
\mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1}
\end{array}
$$

und entsprechend der Darstellung in Abb. 5.3 ergibt sich Abbildung 5.4.

5.3 Game of life

Ein sehr bekanntes Beispiel für Zellularautomaten ist das 1970 von John Conway entwickelte Populationssystem: Game of life. Es besteht aus einem zweidimensionalen Sy-

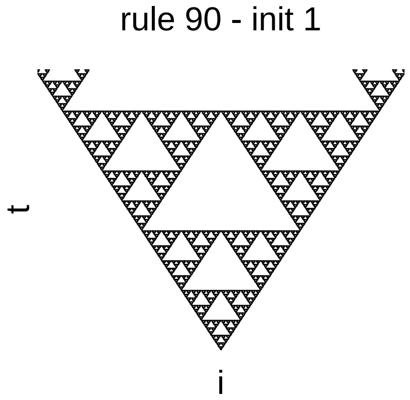

Abbildung 5.4: 300 Zeit-Schritte für die Regel 90 mit einer Eins als Anfangsbedinung.

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.

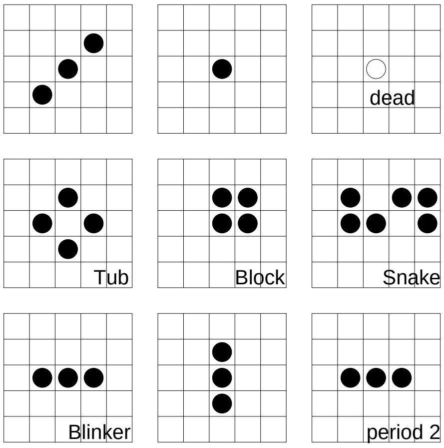

Abbildung 5.5: Aussterbende (oben), stabile (mitte) und periodische (unten) Figuren im Game of Life.

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).

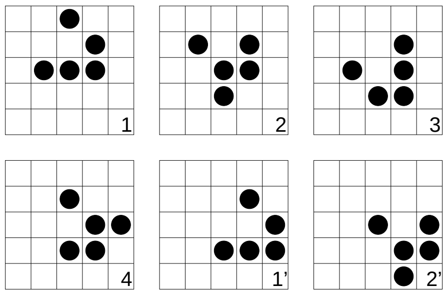

Abbildung 5.6: Die vier Phasen des Gleiters – verbunden mit einer Bewegung nach rechts unten.

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
```