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.

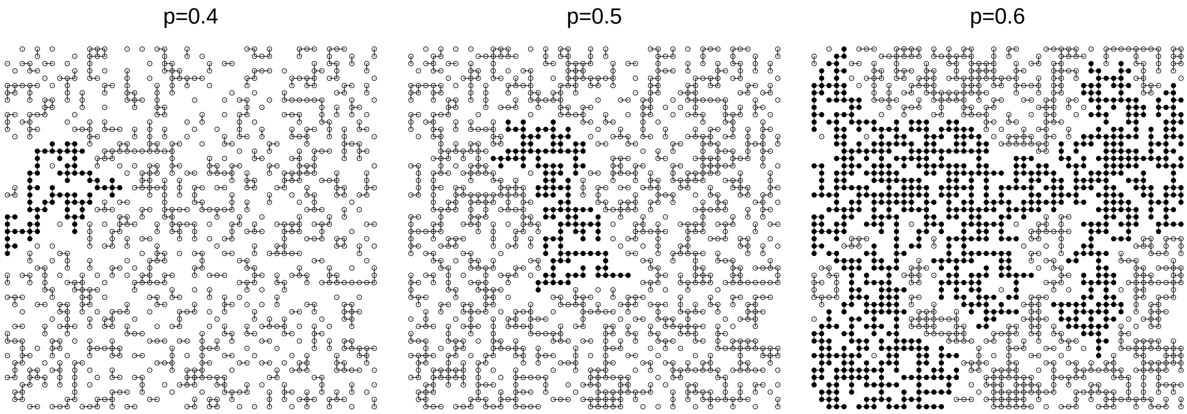

Abbildung 5.11: Perkolation mit verschiedenen Besetzungswahrscheinlichkeiten p. Besetzte Plätze sind als offene Kreise dargestellt, Verbindungen als Linien und die zum größten Cluster gehörenden besetzten Plätze als gefüllte Kreise.

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.

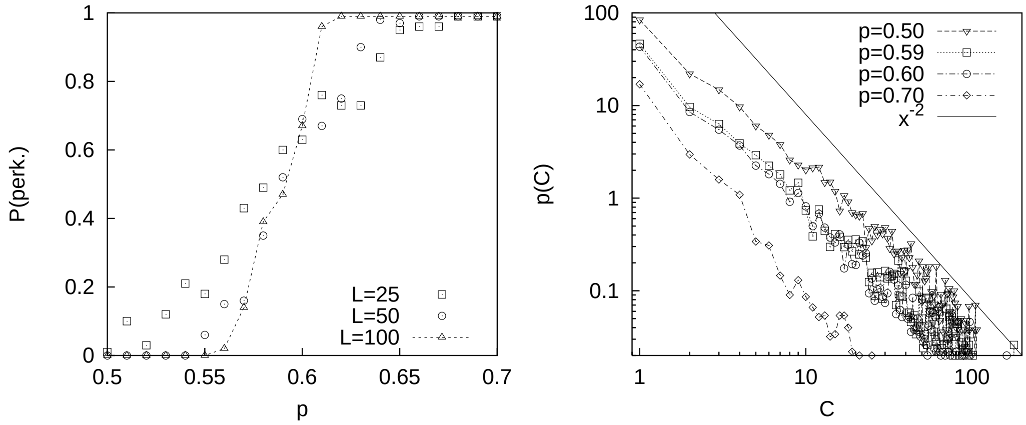

Abbildung 5.12: Perkolationswahrscheinlichkeit als Funktion der Besetzungswahrscheinlichkeit p für verschiedene Systemgrößen L, also für $N = L^2$ Gitterplätze (Links). Clustergrößenverteilung für verschiedene p (Rechts).

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:

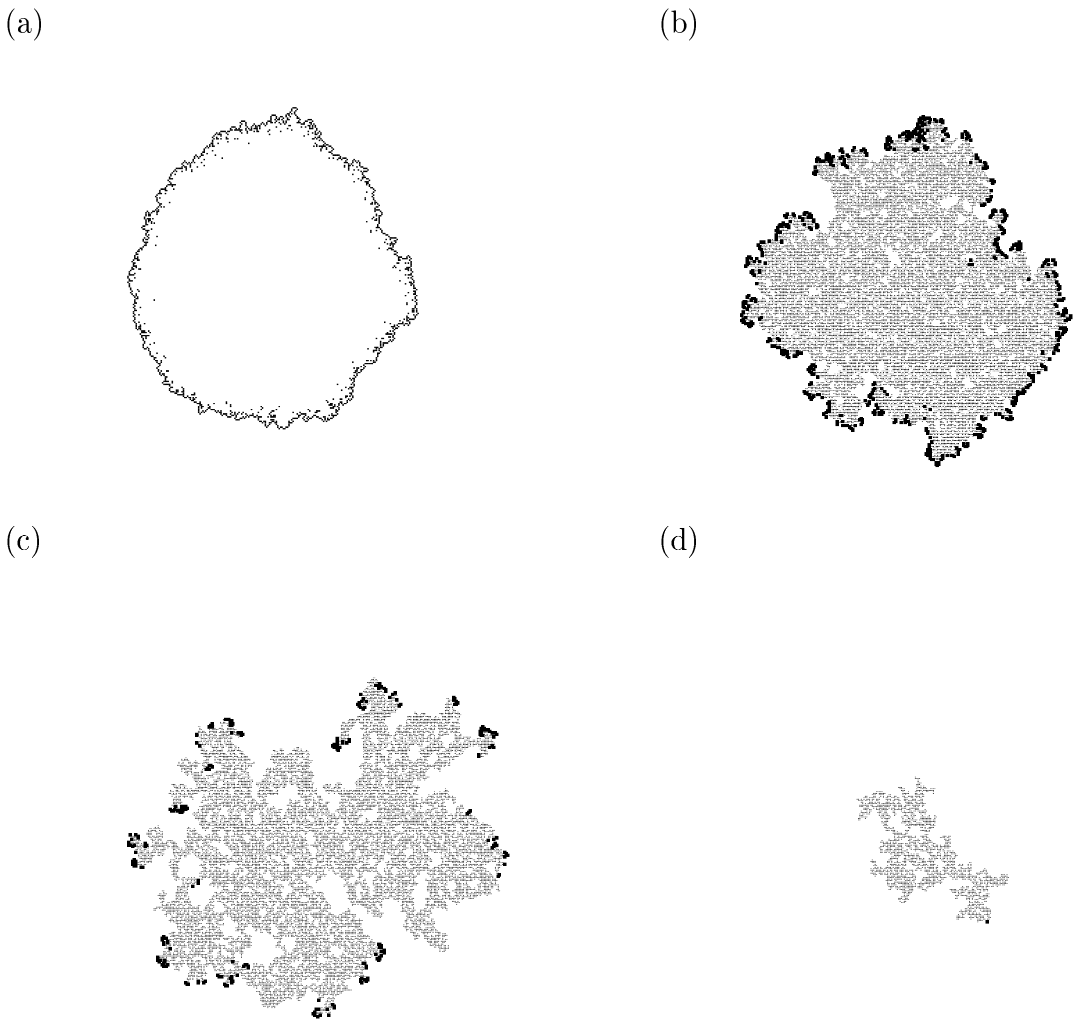

Abbildung 5.13: Eden-Wachstumsmodell mit Immunitätsraten (a) $p = 0$, (b) $p = 0.35$, (c) $p = 0.40$ und (d) $p = 0.415$.

```
#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).

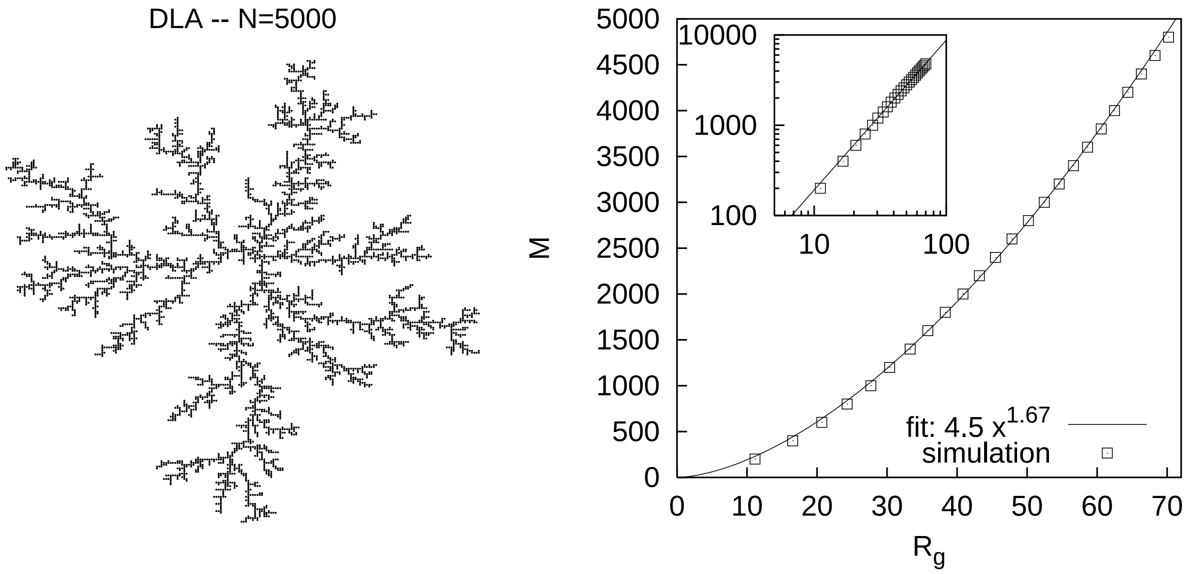

Abbildung 5.14: DLA Cluster mit $N = 5000$ Molekülen (Links). Masse als Funktion des Gyrationsradius R_g (Rechts) – auch in doppelt-logarithmischer Darstellung (Fenster).

Man kann die Ausdehnung des Objekts durch den sog. Gyrationsradius

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

quantifizieren und dann die Masse M (=Anzahl der Moleküle N) als Funktion von R_g darstellen, siehe Abb. 5.14 (Rechts). Die Steigung der Geraden, die man in doppelt-logarithmischer Darstellung findet, ist die sog. fraktale Dimension d_f, womit die Masse dem Gesetz

$$M \propto R_g^{d_f} \tag{5.53}$$

folgt. Ist d_f von der Dimension des einbettenden Raums (hier $D = 2$) verschieden, dann nennt man das Objekt ein Fraktal. Für kompakte Objekte ist d_f eine ganze Zahl $d_f = D$, für Fraktale eine reelle Zahl – im Fall der DLA ist $d_f \approx 1.66$. Ein DLA-Cluster ist also ein Objekt zwischen einer Linie und einer Fläche, mit mehr Masse als die Linie, aber weniger Masse als die solide Fläche bei gleicher Ausdehnung.

Eine Konsequenz von Gleichung 5.53 ist die Selbstähnlichkeit der Fraktale: Betrachtet man ein Fraktal (im Mikroskop) bei verschiedenen Vergrößerungen, so kann man subjektiv keinen Unterschied feststellen.

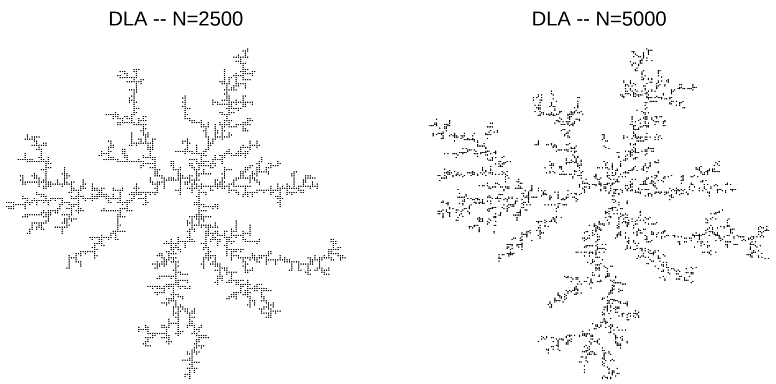

Abbildung 5.15: Verschieden große DLA Cluster mit $N = 2500$, 5000 und 10000 Molekülen. Die Vergrößerung ist so eingestellt daß die DLA Cluster in etwa die selben Flächen bedecken.

Zur Simulation von DLA bedient man sich im wesentlichen eines Random Walks aus Kapitel 5.4. Um Zeit zu sparen beginnt man mit dem Random Walk in der Nähe des Keims (bzw. des Clusters). Um keine Richtung zu bevorzugen setzt man alle RW nacheinander zufällig auf einen Kreis um den Cluster. Von dort läßt man ihn loslaufen – verwirft ihn aber auch wieder falls er aus dem Gitter (dem System) hinauslaufen will. Wieder wird als Kriterium der Abstand vom Zentrum genommen und nicht etwa der Rand des Simulationsgitters, da damit manche Richtungen bevorzugt würden. Sobald ein RW den DLA Cluster erreicht lagert man ihn an und läßt den nächsten RW loslaufen. Eine Variante, nämlich eine Anlagerungswahrscheinlichkeit mit $p_a < 1$ ist hier nicht realisiert.

Ein Simulationsprogramm, das den oben beschriebenen Algorithmus realisiert könnte folgendermaßen aussehen:

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

static const int NL=1000;        // grid size
static int world[NL+2][NL+2];    // define the grid for the DLA

int main()
{
  int NGROW=5000;                // stop after 5000 particles DLA
  int DNOUT=200;                 // control output after 200 new pps
  int OUTNXT=DNOUT;              // next output particle number
  int i0x, i0y, imerge;          // coordinates of the random walker
  int sum_particles = 0;         // pp-sum
  double dxy, rxm, rym, r2m;     // distance variables
  double rnum, rmax, rij;

  for(int i=1; i<=NL; i++ )
    for(int j=1; j<=NL; j++ )
      world[i][j]=0;             // all empty sites
  world[NL/2][NL/2]=1;           // only one is occupied

  rng0(12345670);                // initialize rng()
  ofstream outfile1("dla1.dat"); // open datafile for DLA output

  rmax = 5.;                     // radius of the initial setup circle
                                 // BEGIN loop for new RW
  for( int it=1; it<=NGROW*10; it++ )
  {
     rnum = 2.*M_PI*double(rng())/dp;             // choose angle on circle
     i0x = NL/2 + int( rmax * sin( rnum ) );      // determine the next RW
     i0y = NL/2 + int( rmax * cos( rnum ) );      //   starting point

     imerge = 1;
     while(( world[ i0x+1 ][ i0y ] == 0 ) &&      // RW loop is performed
          ( world[ i0x ][ i0y+1 ] == 0 ) &&      // until the DLA is hit
          ( world[ i0x ][ i0y-1 ] == 0 ) &&
          ( world[ i0x-1 ][ i0y ] == 0 )){
          dxy = double(rng())/dp;                // select RW direction

          if( dxy < 0.25 ){
             if( i0x > 1 )
                if( world[ i0x-1 ][ i0y ] == 0 ) i0x -= 1;       // LEFT
```

```
      }
      else{
         if( dxy < 0.5 ){
            if( i0x < NL )
               if( world[ i0x+1 ][ i0y ] == 0 ) i0x += 1;    // RIGHT
         }
         else{
            if( dxy < 0.75 ){
               if( i0y > 1 )
                  if( world[ i0x ][ i0y-1 ] == 0 ) i0y -= 1; // DOWN
            }
            else{
               if( i0y < NL )
                  if( world[ i0x ][ i0y+1 ] == 0 ) i0y += 1; // UP
            }
         }
      }
      rij=(i0x-NL/2)*(i0x-NL/2)+(i0y-NL/2)*(i0y-NL/2);
      if( sqrt( rij ) > 1.5*rmax ){
         imerge=0;                   // if the distance from the center
         break;                      //   is too large then forget the RW
      }
   }

   if( imerge == 1 ){                 // add new particle to the DLA
      world[ i0x ][ i0y ] = sum_particles;
      outfile1 << i0x << ' ' << i0y << ' ' << world[i0x][i0y] << '\n';

      sum_particles += 1;            // count total number
      // cout << i0x << ' ' << i0y << ' '
      //      << sum_particles << ' ' << rmax << '\n';

      rij=(i0x-NL/2)*(i0x-NL/2)+(i0y-NL/2)*(i0y-NL/2);
      if( sqrt(rij)+5 > rmax ){
         rmax=sqrt(rij)+5;               // let inscribing circle grow
      }                                  //    if necessary
      if(( int( 1.5*rmax ) > NL/2-2 ) ||
         ( sum_particles >= NGROW )){    // STOP if outer circle
                                         //    hits world's end
         cout << "# last rmax = " << rmax << '\n';
         outfile1.close();
         return sum_particles;
      }                               // end of merge-IF
   }                                  // end of RW-WHILE
```

```
    rxm=0.; rym=0.; r2m=0.;               // calculate CM position and Rg
    if( OUTNXT == sum_particles ){        //    if next output time
       OUTNXT += DNOUT;                   // increase for the next output
       for(int i=1; i<=NL; i++ ){
         for(int j=1; j<=NL; j++ ){
           if(world[i][j] > 0){
             rxm+=i;                      // x-mean
             rym+=j;                      // y-mean
           }
         }
       }
       rxm /= sum_particles;              // center of mass position x
       rym /= sum_particles;              // center of mass position y

       for(int i=1; i<=NL; i++ ){
         for(int j=1; j<=NL; j++ ){
           if(world[i][j] > 0){
             rij=(i-rxm)*(i-rxm)
                +(j-rym)*(j-rym);
             r2m+=rij;                   // average of the variance
           }
         }
       }
       r2m = sqrt(r2m/sum_particles);     // compute the radius of gyration

       cout << sum_particles << ' ' << rxm << ' ' << rym << ' '
            << r2m << ' ' << rmax << ' ' << '\n';
    }
  }
  outfile1.close();
  return 0;
}
```