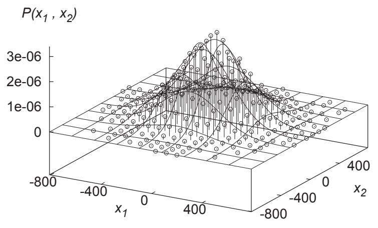

Fig. 12.9 $P\left(x_1(t), x_2(t)\right)$ as a function of $(x_1(t), x_2(t))$ at $t = 10^5$. The vertical lines ending in a circle represent numerical data, while the continuous grid represents the mathematical result we obtain in the large time limit.

12.3 A lattice gas

A first extension to the study of random processes, which is more interesting though still extremely simple, is a lattice gas of free particles. Variations of this model have given rise to a major research area. For instance, the continuous hydrodynamic equations which are used in meteorologic prediction models and in car and airplane design are very complex. Approximating them by relations defined on a lattice, of which the lattice gas model discussed in this section is the starting point, leads to remarkable results.

Consider a a two-dimensional square lattice, i.e., $D = 2$. A fraction ρ of the lattice sites are occupied by particles. For now we consider the case where each site is occupied by a particle with probability ρ. The total number of sites of the lattice is V. In two dimensions, we have $V = L^2$, where L is the number of sites in one dimension.

Initially, the particles are randomly distributed on the lattice. In this case, the lattice gas density itself is a random variable, determined by the distribution of the particles on the lattice. The dynamics we study here does not alter the density. Another possible approach is to first fix the number of particles on the lattice ($N = \rho\ V$) and then distribute them randomly among the sites, taking the following *exclusion principle* into account: if a site is already occupied by a particle, it cannot be occupied by a second one. The particles are numbered, i.e., they are characterized by a label, and during the dynamics they keep this original label. A typical example is given in Figure 12.10.

The problem can be defined in terms of *occupation variables* η_{x_1,x_2} which can assume values 0 and 1: $\eta_{x_1,x_2} = 0$ says that site (x_1, x_2) is unoccupied, while $\eta_{x_1,x_2} = 1$ says the site is occupied. Note that the case in which

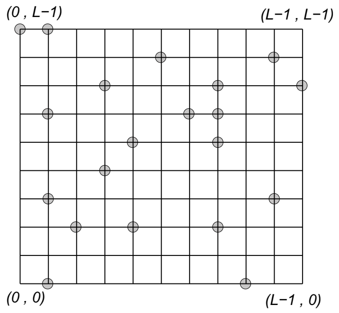

Fig. 12.10 A lattice gas of density $\rho = 0.2$. $D = 2$, $L = 10$ and $V = L^2 = 100$.

the variables take on values $+1$ or -1, i.e., $\sigma_{x_1,x_2} = \pm 1$, is equivalent. Due to the analogy with a two-state magnetic system, these variables are often referred to as *spin variables*. It is straightforward to verify that the transformation $\sigma_{x_1,x_2} = 2\,\eta_{x_1,x_2} - 1$ links σ_{x_1,x_2} to η_{x_1,x_2}.

The lattice we consider has *periodic boundary conditions*: a particle leaving the lattice on the right side border, reappears at the border on the left, one that leaves it on top, enters the lattice back at the bottom. In other words, the site with coordinates (i, j) is identical to the sites with coordinates $(i \pm L, j)$, $(i, j \pm L)$, $(i \pm L, j \pm L)$ and so on.

The dynamics which we are about to define conserves the particle density. Since the particles can only move to an adjacent empty site, and they are not allowed to disappear or reappear, the total number of particles

$$N = \sum_{x_1,x_2} \eta_{x_1,x_2}$$

remains invariant.

A time step is defined by various actions. First, randomly pick a particle. We already know how to do this, namely, we need a uniformly distributed integer random number between 1 and N. At this point, independently of whether the particle is eventually moved or not, we increase the elapsed time by one unit. Next, we randomly select one of the 4 nearest neighbors of the site occupied by the particle in question. If this neighboring site is empty, the particle moves onto this new site, otherwise a new step starts.

The squared distance the particle α has traveled can be expressed in function of its coordinates as

$$\Delta R^{(2)}_{\alpha}(t) \equiv \left(x_1^{\alpha}(t) - x_1^{\alpha}(0)\right)^2 + \left(x_2^{\alpha}(t) - x_2^{\alpha}(0)\right)^2 \ .$$

Considering the average with respect to the positions of the N particles, the *mean square distance* of the system is defined as

$$\Delta R^{(2)}(t) \equiv \frac{1}{N} \sum_{\alpha=0}^{N-1} \Delta R^{(2)}_{\alpha}(t) \ . \tag{12.16}$$

Note a first technical problem with the definition of the mean square distance (12.16). A direct application of (12.16) is not appropriate when periodic boundary conditions are used. Suppose a particle leaves at time zero from site $(0, 0)$, and at the first step moves in the $-\hat{x}$-direction, thus reaching site $(L - 1, 0)$. In this particular case, the displacement of one lattice step appears to be of $L - 1$ steps instead! In order to avoid this type of problem, we should keep track of two types of lattice positions. A first type which refers to the position on the basic lattice defined with its periodic boundary conditions, and a second type which does not take the latter into account, but simply increase or decrease by $+1$ if a displacement in the positive or negative direction, respectively, occurred. The latter type of positions are used in (12.16), while the former type are used to verify at each step whether a particle neighboring sites are occupied or not.

The most important quantity characterizing the system (diffusive) behavior is the *diffusion coefficient*, defined as

$$\mathcal{D}(\rho) = \lim_{t\to\infty} \mathcal{D}(\rho, t) = \lim_{t\to\infty} \frac{1}{2Dt} \left\langle \Delta R^{(2)}(t) \right\rangle \ , \tag{12.17}$$

where the expected value is computed by averaging over several histories of the system motion. Each time the system is reinitialized and the dynamics takes place starting at time zero up to a time which is large enough to allow an accurate prediction of the limit in (12.17). The diffusion velocity decreases with increasing density as in this case it becomes more and more difficult for the particles to move around. Obviously, a fully occupied lattice is completely blocked, while a single particle on a lattice behaves at each step exactly like a normal random walk. Note that also for the dynamics of a lattice gas, the square distance divided by a time asymptotically tends to a constant: this fact is the most important property of random walks.

Hands on 12.5 - The diffusion coefficient

Compute the diffusion coefficient $\mathcal{D}(\rho)$ for $\rho = 0.2$, 0.4, 0.6 and 0.8. Repeat the same computation for lattices of different sizes (for example with $L = 10$, 20, 40 and 80) and verify that for larger lattice volumes the results are independent from the volume. The computation has to be performed in the region where $\langle \Delta R_\alpha^{(2)}(t) \rangle^{\frac{1}{2}} < \frac{L}{2}$, as finite size effects may occur in the region where the square root of the mean square displacement becomes of the order of the size of the lattice. However, the finite size effects on this lattice are rather limited: try to quantify them. How does $\mathcal{D}(\rho)$ depend on ρ?

Study a two-dimensional triangular lattice, with connectivity six (each site has six nearest neighbors). How does this case differ from the previous one?

The code simulating a lattice gas is probably a bit more complicated with respect to the ones we have dealt with so far. This is a good reason to subdivide it into several functions. The prototypes of the functions we use are:

```
double myInit(void);
long int initLattice(double rho);
void updateLattice(long int thisNumberOfParticles);
double measure(long int thisNumberOfParticles);
void myEnd(double averageRho);
```

The function `myInit` is called one time when the execution starts. It carries out the necessary initialization procedures. For example, it opens the output files, which are controlled by some global variables. Further on, we discuss how to identify the nearest neighboring sites in an efficient way, and how to efficiently initialize an array. The function `initLattice` is called each time the lattice is initialized to extract a new initial configuration, simulate its dynamics and extract the corresponding sample properties which allows us to evaluate the corresponding averages and their statistical errors. The function `updateLattice` is the motor of the program. It lets the time increase by one unit, during which we try to move a particle. The total number of times we try to do this is equal to the number of particles in the lattice. The function `measure` computes the relevant quantities (in our case this is the mean square distance) for which we then compute the averages. The function `myEnd` performs operations which need to be executed only once at the end of the code (for example, normalize average values, calculate their errors and close open files). The more important arrays we choose to use are

```
long int particleOfSite[L][L];
long int positionOfParticle[VOLUME][DIM];
long int truePositionOfParticle[VOLUME][DIM];
long int zeroPositionOfParticle[VOLUME][DIM];
```

where `DIM` is the spatial dimension of the system, which in our case is 2. We are defining a double link between particles and sites: each particle is linked to a site and each site is linked to a particle. When a particle moves from one site to another, we need to adjust the vector containing the particle coordinates, and the two arrays indicating which particle was at the old site and which one is at the new site. The array `particleOfSite[x][y]` keeps track of the label of the particle that occupies the site with coordinates `x` and `y` at a given time. If the corresponding site is empty, it is given the value `MY_EMPTY`. We use three vectors for the particles positions: `zeroPositionOfParticle[VOLUME][DIM]` contains the position at time zero, `positionOfParticle[VOLUME][DIM]` keeps track of the position at the current time (taking the periodic boundary conditions into account, i.e., these coordinates have values from 0 up to $L - 1$, and a particle leaving the lattice, reappears at the opposite side of the lattice), and `truePositionOfParticle[VOLUME][DIM]` which keeps track of the "absolute" position obtained by neglecting the periodic boundary condition and used to compute the mean square displacement. The average values at various time instances calculated together with their respective statistical errors are kept in two arrays. We do not calculate the expected values at all times, but only at some selected time instances. To this end, we define the following two arrays

```
double averageDeltaR2[NUM_MEASUREMENTS];
double errorDeltaR2[NUM_MEASUREMENTS];
```

The constant `NUM_MEASUREMENTS` is defined as following

```
#define NUM_SWEEPS 1000
#define NUM_MEASUREMENTS 100
#define MEASUREMENT_PERIOD (NUM_SWEEPS / NUM_MEASUREMENTS)
```

In this way the total number of steps is guaranteed to be a multiple of the number of measurements. In the function `init` we include

```
if ((NUM_MEASUREMENTS * MEASUREMENT_PERIOD) != NUM_SWEEPS) {
  printf("Error: number of steps is not a multiple\n");
  printf("       of number of measurements\n");
  exit(MY_EXIT_FAILURE);
}
```

The measurements therefore only takes place after every MEASUREMENT_PERIOD time steps. After calling the function updateLattice in the main function, we write

```
if ((sweep % MEASUREMENT_PERIOD) == 0) {
  double deltaR2 = measure(trueN);
}
```

where trueN contains the exact number of particles composing the considered sample. The function measure computes and returns the mean square displacement at the considered time. The omitted code includes the update of the averages and the errors.

In a code which is slightly more complex than the ones we considered so far, it is useful to include and maintain some lines of code which verify its internal coherency. These are pieces of code which do not need to be executed in case we are sure everything works fine, but which help to identify incoherences or signs of programming errors. They prove to be very useful when changing the code in order to generalize it or to include other functions. However, the coherence checks slow down the execution of the code. A good possible solution to this dilemma is based on using the *preprocessor* of the C language, discussed in Section 3.5. In this way certain lines of code are only conditionally included depending on whether a certain constant has been defined or not. By including at the beginning of our C code,

```
#define MY_DEBUG
```

we define a constant MY_DEBUG to which we do not attribute a specific value. The important thing is to define and include it in our program dictionary. For example, in the function main, we write

```
#ifdef MY_DEBUG
if (numMeasure >= NUM_MEASUREMENTS) {
  printf("Error: numMeasure too large\n");
  exit(MY_EXIT_FAILURE);
}
#endif
averageDeltaR2[numMeasure] += deltaR2;
```

This stops the execution of the program with a message and an error code if numMeasure (a counter of the actually performed number of measurements) is too large. We know that if the program is correct this error should not occur as numMeasure cannot get to NUM_MEASUREMENTS. When writing or improving the program, we want to verify this condition to check whether we did not introduce any errors (e.g., logic or typographical) in the program. Program *debugging*, i.e, locating these errors, is a crucial aspect of programming. At the end of the debugging, we can simply remove the line which defines the variable MY_DEBUG, or add a line which cancels out its definition:

```
#define MY_DEBUG
#undef MY_DEBUG
```

The myInit manages (one single time, at the beginning) the *periodic boundary conditions*. Usually the nearest neighbor in the $+\hat{x}$-direction of the site with coordinates (x, y) has coordinates $(x+1, y)$. However, for the site with coordinates $(L-1, y)$ this becomes the site with coordinates $(0, y)$. Managing this problem with if statements is extremely inefficient as it requires many operations for each site. A good solution consists in defining two vectors plusNeighbor[L] and minusNeighbor[L], of type long int, which are initialized by the following statements

```
for (i = 0; i < L; i++) {
  plusNeighbor[i] = i + 1;
  minusNeighbor[i] = i - 1;
}
plusNeighbor[L - 1] = 0;
minusNeighbor[0] = L - 1;
```

Now, the neighbor of (x, y) in the $+\hat{x}$-direction has coordinates (plusNeighbor[x], y).

The code defining the particles initial configuration is executed in initLattice, each time a new sample is started. After the vectors have been initialized to MY_EMPTY, the following lines of code are executed:

```
trueN = 0;
for (x = 0; x < L; x++) {
  for (y = 0; y < L; y++) {
    randomNumber = a * randomNumber;
    r = (double)randomNumber * iDoubleM;
    if (r < rho) {
      particleOfSite[x][y] = trueN;
      positionOfParticle[trueN][0] = x;
      positionOfParticle[trueN][1] = y;
      zeroPositionOfParticle[trueN][0] = x;
      zeroPositionOfParticle[trueN][1] = y;
      truePositionOfParticle[trueN][0] = x;
      truePositionOfParticle[trueN][1] = y;
      trueN++;
    }
  }
}
```

The function creates samples of variable density which on average is equal to ρ. trueN is the actual number of particles of the sample under consideration, and thus varies from sample to sample. A function initializing the configuration while keeping the number of particles fixed would be different. It is a good exercise for the reader to write such a function.

The earth of the updating procedure is contained in `updateLattice`, which manages the updates of the particles positions. First, one of the `trueN` particles is randomly chosen with a uniform probability and the time is increased by one unit, independently of whether a change in the particle position will occur. Then, after a possible direction in which the particle could move has been chosen (for example, if `thisDirection` equals 0 we move in the positive direction of the x-axis), we write

```
if (particleOfSite[nX][nY] == MY_EMPTY) {
  particleOfSite[nX][nY] = particleOfSite[x][y];
  particleOfSite[x][y] = MY_EMPTY;
  positionOfParticle[thisParticle][0] = nX;
  positionOfParticle[thisParticle][1] = nY;
  if (thisDirection == 0) {
    truePositionOfParticle[thisParticle][0]++;
  } else if (thisDirection == 1) {
    truePositionOfParticle[thisParticle][0]--;
  } else if (thisDirection == 2) {
    truePositionOfParticle[thisParticle][1]++;
  } else if (thisDirection == 3) {
    truePositionOfParticle[thisParticle][1]--;
  }
}
```

If the randomly chosen neighbor (`nX` stands for *neighbor x*) is empty, the particle moves to this neighboring site, and the previous site it occupied is emptied. Correspondingly, we update the particle position vectors, both the one which does as the one taking periodic boundary conditions into account as the one which does not.

Let us analyze the results one obtains with this code. Figure 12.11 shows the diffusion coefficient $\mathcal{D}(\rho, t)$ (defined in (12.17)) as a function of time t for several values of the density ρ. These data were obtained for $L = 80$ and a density which varies from 0.1 up to 0.9 at steps of 0.1. $\mathcal{D}(\rho, t)$ monotonously decreases with ρ. As expected, $\mathcal{D}(\rho, t)$ tends to a constant at large times. Instead, at smaller times there is a systematic variation, that can already be seen on the scale used in the figure. It is worth focusing on the behavior occurring at smaller times: this allows us to get a better understanding of the convergence of the diffusion constant.

Figure 12.11 is the result for a system with $L = 80$, i.e, a volume composed of 6400 sites organized on a two-dimensional square lattice with

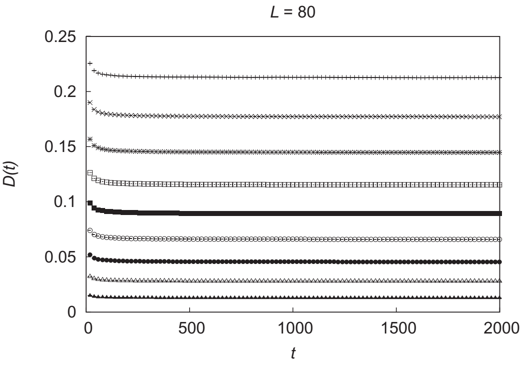

Fig. 12.11 The diffusion coefficient $\mathcal{D}(\rho, t)$ as a function of time t. $L = 80$ and $V = L^2 = 6400$. The larger coefficients correspond to the systems with lower density (from top to bottom: $\rho = 0.1$, 0.2, 0.3 up to 0.9, at steps of 0.1).

periodic boundary conditions. Why did we chose $L = 80$? Before opting for this choice, we studied various lattice sizes to ensure *finite size errors* were under control. Namely, the fact that we are dealing with a lattice of finite size brings about that the estimate for the coefficient depends on L. Only in the limit $L \to \infty$, we obtain the result typical of an infinite volume (which is the one we are interested in). In Figure 12.12 we show the diffusion coefficient $\mathcal{D}(\rho, t)$ as a function of time t for $\rho = 0.6$, computed on lattices of size $L = 10$, 20, 40 and 80. There is a 3 percent difference between the value obtained on a lattice composed of 100 sites and the correct one! Also the value obtained on a lattice with $L = 20$ is visibly different from the asymptotic value, while the values obtained on lattices with either $L = 40$ or $L = 80$ coincide (considering our statistical precision). This indicates that by using the lattices of this size we have reached a sufficient systematic precision. Note that this systematic precision is strictly related to the statistical precision of our numerical quantities. Therefore, it is useless to set one of these uncertainties a lot smaller than the other. Also note that the numerical results obtained for a lattice of a particular size can never provide us with any indications on how precise these results are. The precision can only be deduced by observing how the numerical results change when changing the size. Finite size effects are analyzed by comparing several lattice sizes and checking the behavior of the measured quantities as L increases. Finally, the scale used in Figure 12.12 allows us to observe a small remnant drift, indicating that the system has not yet fully reached the asymptotic regime at large times, where $\mathcal{D}(\rho, t)$ no longer depends on the time t. This drift becomes even more obvious on a plot where the time is represented on a logarithmic scale. We leave this as an exercise to the reader.

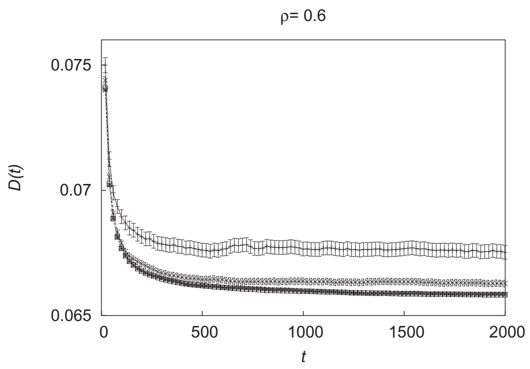

Fig. 12.12 The diffusion coefficient $\mathcal{D}(\rho, t)$ as a function of time t, for $\rho = 0.6$. The series of points from top to bottom correspond to lattices of size $L = 10$, $L = 20$, $L = 40$ and $L = 80$, where the latter two essentially overlap each other (the finite size effects are negligible).

Figure 12.11 shows how the diffusion coefficient decreases monotonously as a function of the density ρ, for each value of t. This is rather intuitive: when few particles are present on the lattice, it should be easier for them to move around. Instead, if many particles are present, often a nearest neighboring site will be occupied and therefore the particles are expected to move less. To study the behavior of $\mathcal{D}(\rho)$, we select those numerical data that have reached the asymptotic regime. Another more accurate, but also more complex approach consists in extrapolating the desired data at finite times using an appropriate functional form. We here avoid this more complicated approach as we assume to have gathered enough evidence that the asymptotic regime has indeed been reached. The data of Figure 12.13 are the result of an averaging procedure over times ranging form 16 000 up to 20 000, where each time step or unit consists in $\rho\ V$, i.e., `trueN`, attempts of moving a randomly chosen particle. This time scale is clearly larger than the one reported in Figure 12.11 and 12.12, in which we were mainly considered with demonstrating the existence of a transient phase. Figure 12.13 illustrates the behavior of the diffusion coefficient computed in this way, as a function of the density ρ.

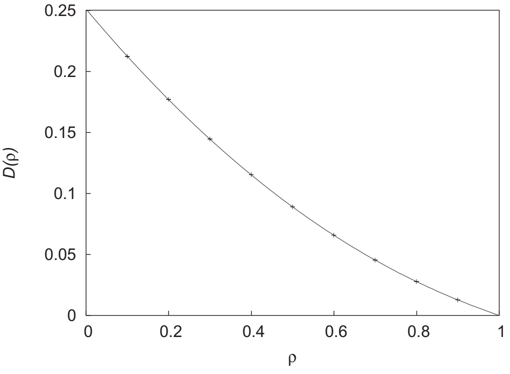

Fig. 12.13 The diffusion coefficient $\mathcal{D}(\rho)$ as a function of the density ρ, computed in the infinite time limit.

Which function describes the behavior of the data in Figure 12.13 in function of ρ analytically? A first, too simple attempt could be a linear behavior of $\mathcal{D}(\rho)$ in function of $(1-\rho)$. Indeed, as a first approximation, increasing the density causes a decrease in the mean displacement of a particle at a given time step, thus effectively reducing a lattice step from a to $(1-\rho)a$. This is a reasonable first approximation, given that a single particle on a lattice moves a distance a, while a particle with on average two neighbors moves on average one out of two times, i.e., a distance $\frac{a}{2}$.

However, this does not cover all aspects of the dynamics. Let us consider a rather dense system. A particle residing in a region with lower density (with respect to the average density), has a higher probability to remain there rather than moving to a denser region. There is a strong *correlation* correcting the first linear approximation. Namely, there exists a secondary effect connected to the case when two particles are close to each other, which must be dependent on ρ squared, or better on a term $(1-\rho)^2$. The continuous curve in Figure 12.13 is a best fit to a functional form with four free parameters (two multiplicative factors and two powers of $(1-\rho)$). We find two very similar coefficients close to the value $\frac{1}{8}$, an exponent close to 1 and another one close to 2: this is fully consistent with our previous reasoning.