Computational Physics – Lecture 9

Andrei Alexandru

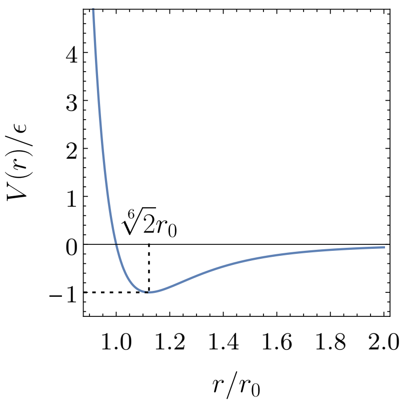
April 12, 2021

1 Introduction

This week we will discuss Monte-Carlo simulations for a model a gas of particles using the Lennard-Jones potential. From a statistical mechanics point of view, this type of simulation generate configurations (that is, particle positions and velocities) according to the *canonical ensemble*, a statistical ensemble corresponding to a system kept at constant temperature. Last week we discussed the same problem, a two-dimensional gas of particles, using the *micro-canonical ensemble*, corresponding to systems kept at constant energy. We will first review some details about the microscopic interactions for the system we plan to study and then discuss the relevant statistical mechanics details.

The principal interatomic attractive force in many gases is the van der Waals force, which follows an r^{-6} potential. However, if two atoms come too close together, this is outweighed by the repulsive force due to the overlap of the electronic orbitals. Often this is approximated by a repulsive r^{-12} potential. This potential is called the *Lennard-Jones potential*, and has the form

$$V(r) = 4\epsilon\left[\left(\frac{r_0}{r}\right)^{12} - \left(\frac{r_0}{r}\right)^{6}\right] \tag{1}$$



where ϵ governs the strength of the force and r_0 the length scale. As you can see from the figure, the minimum of the potential is at $r_{\min} = \sqrt[6]{2}r_0$ and its value there is $-\epsilon$. Thus, at zero temperature, when the system is assuming the lowest energy, the particles will try to arrange themselves in a structure that makes the distance between them close to $r_{\min}$. The total potential energy of the gas due to the Lennard-Jones potential is

$$U = \sum_{i<j} V(r_{ij}), \quad \text{where} \quad \boldsymbol{r}_{ij} = \boldsymbol{r}_i - \boldsymbol{r}_j\,. \tag{2}$$

Note that the sum above is restricted to $i < j$ such that every particle pair is counted only once.

To study the thermodynamics properties of the gas last week we used *molecular dynamics* simulations, that is we integrated the equations of motion for a system of N particles confined to a box that interact via the Lennard-Jones potential. We then took snapshots of the system at equal times intervals and computed the average of thermodynamic observables, like kinetic energy and pressure, over the snapshots. Taking a statistical point of view, the snapshots provide a probability distribution of configurations over the space of all configurations allowed for the system and the thermodynamics observables are viewed as averages over this distribution. The probability distribution generated by the dynamics of the system is confined to a subregion of the configuration space where the energy is fixed to the value set by the initial conditions. The probability distribution generated this way is called the *microcanonical ensemble* in statistical mechanics, where the system, in our case the gas of N particles, is assumed to be isolated from the environment.

Another useful probability distribution is the one generated by a system that is in *thermal contact* with a energy reservoir at fixed temperature T. The probability of one particular configuration (that is positions $\boldsymbol{x}_i$ and velocities $\boldsymbol{v}_i$) to occur is proportional to the Boltzmann factor that depends only on the energy of the configuration and the temperature of the reservoir

$$P(\boldsymbol{x}_1,\ldots,\boldsymbol{v}_1,\ldots) \propto e^{-E(\boldsymbol{x}_i,\boldsymbol{v}_i)/k_BT} \quad \text{with} \quad E = \sum_i \frac{1}{2}m\boldsymbol{v}_i^2 + U(\boldsymbol{x}_i)\,. \tag{3}$$

The goal for this week is to setup a process that generates configurations based on the probability functions above, called *canonical ensemble* in statistical mechanics. Once a set of such configurations is generated, the thermodynamic observables will be measured as averages over the generated set of configurations.

For the case of no interactions, the gas has the equation of state of an *ideal gas*:

$$PV = NkT\,, \tag{4}$$

where P is the pressure, V is the volume of the system, N is the number of particles, T is the temperature, and k is the Boltzmann constant. We will study a two dimensional gas, so the role of the volume is played by the area of the box A and the pressure is ratio of the force the gas exerts on the wall, due to collision, on the length of the wall. The interactions lead to a change in this equation of state: for high densities where the distance between the particles is comparable with r_0, the short-range repulsion leads to an increase in pressure, whereas for low densities, the long-range attraction lowers the pressure compared to the ideal gas law.

2 Markov chains and the Metropolis algorithm

The basic task is to setup a process that generates configurations x with the desired probability distribution $P(x)$. Here x denotes a complete description of the system, for example for the two dimensional gas of N particles it would represent $4N$ parameters corresponding to positions and velocities. For clarity, we begin the discussion with a simple case where x takes values in a discrete set of variables or a continuous interval of the real axis.

A Markov chain represents a sequence of values x generated by a stochastic process controlled by the *transition probability* $T(x \to y)$. The sequence is generated starting with a initial value x_0. We then choose x_1 with probability $T(x_0 \to x_1)$, x_2 with probability $T(x_1 \to x_2)$, and continue indefinitely. Under certain conditions for T, the sequence generated by this process is distributed with probability $P(x)$ that is controlled by the choice of T. In most cases of interest we have a desired $P(x)$ and the goal is to design T accordingly. A sufficient condition on T is that it satisfies *detailed balance* with respect to $P(x)$

$$P(x)T(x \to y) = P(y)T(y \to x) \quad \text{for any } x \text{ and } y. \tag{5}$$

Let us look at a concrete example: take a coin that can be heads (h) or tails (t). In this case, a Markov chain is completely specified by listing $T(h \to h)$, $T(h \to t)$, $T(t \to h)$, and $T(t \to t)$. We can choose to represent this by a 2×2 matrix. One choice would be a forced flip:

$$T(x \to y) = \begin{pmatrix} h \to h & h \to t \\ t \to h & t \to t \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\,, \tag{6}$$

which basically force h to transition into t and t into h. Assuming that we start with h the sequence generated is

$$h \to t \to h \to t \to h \ldots \tag{7}$$

While this is not really a stochastic process, this clearly leads to 50% heads and 50% tails. Note that this transition matrix indeed satisfies detailed balance with respect to $P\{h,t\} \to \{0.5, 0.5\}$ since

$$P(h)T(h \to t) = P(t)T(t \to h) \quad \text{check this!}\,. \tag{8}$$

Detailed balance is automatically satisfied when $x = y$ so only $x \neq y$ cases need to be checked.

For a given probability $P(x)$ the Markov chain is not unique. For the example above consider the transition probability

$$T(x \to y) = \begin{pmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{pmatrix}\,. \tag{9}$$

In this case the sequence is stochastic, at each step h and t are chosen with probability 50% each, without regard of the previous state. This is similar to a sequence generated by a fair coin and one particular instance would look like

$$h \to h \to t \to h \to t \to t \to h \to t \to h \dots \tag{10}$$

This again generates the same 50-50 distribution as above. You can also easily check that detailed balance is satisfied.

In general it is not easy to come up with the right transition probability for a given P. There is however a rather general method that can be used when to generate the correct transition function proposed by Metropolis and collaborators [1]. Assuming that we have a *proposal transition* T_0 that is *symmetric*

$$T_0(x \to y) = T_0(y \to x)\,, \tag{11}$$

then we can generate the correct transition $T(x \to y)$ using the following algorithm:

- generate a proposal y_p using $T_0(x \to y_p)$ and compute $r = P(y_p)/P(x)$
- if $r > 1$ then $y = y_p$.
- if $r < 1$ then either *accept* the proposal $y = y_p$ or *reject* it $y = x$, with probabilities r and $1 - r$ respectively.

In summary the proposal is taken as the next step in the sequence, either when it increases the probability, or even when the probability is decreased the proposal is accepted stochastically at a lower rate. The other choice is to have as the new element in the sequence a copy of the previous value x.

Let us use Metropolis method to implement a Markov chain that mimics a unfair coin that has 99% chances of producing h and 1% t. If we use T_0 that produces 50% either h or t and assuming we start with h we have 50% chance to propose h in which case the proposal is accepted, and 50% to generate t which will be accepted with probability $P(t)/P(h) = 1/99$. Thus $T(h \to t) = 0.5/99$ and $T(h \to h) = 1 - 0.5/99$. When we start with t we have 50% chance to propose h and the proposal is accepted and 50% to propose t and the proposal is accepted. The generated transition probability is

$$T(x \to y) = \begin{pmatrix} 1 - 0.5/99 & 0.5/99 \\ 0.5 & 0.5 \end{pmatrix}\,. \tag{12}$$

This should satisfy the detailed balance with respect to $P\{h,t\} \to \{0.99, 0.01\}$:

$$\underbrace{0.99 \times 0.5/99}_{P(h)\times T(h\to t)} = \underbrace{0.01 \times 0.5}_{P(t)\times T(t\to h)}\,. \tag{13}$$

If we repeat the same calculation with T_0 corresponding to the forced flip case, then the transition matrix generated by the Metropolis method is

$$T(x \to y) = \begin{pmatrix} 1 - 1/99 & 1/99 \\ 1 & 0 \end{pmatrix}\,. \tag{14}$$

I leave it as an exercise for you to derive this and check that it satisfies the detailed balance.

One final point, when the proposal is rejected, this does not mean that the Markov chain does not advance. A rejection means that the next element in the sequence is a copy of the previous one. For example, if we consider T_0 as the force flip case, the correct sequence generated by Metropolis looks like

$$h \to h \to h \to h \dots \text{many rejections} \to h \to t \to h \to h \dots\,. \tag{15}$$

At every step when we start with h the proposal is t and it is rejected with high probability, 98/99. If we only advanced the Markov chain when the proposal is accepted that would lead to $h \to t \to h \to t \dots$, a chain that clearly does not have the desired probability distribution.

3 Implementation details

Before implementing the canonical ensemble for a gas, we will implement a simple code that uses Metropolis method to generate random numbers with arbitrary probability distribution functions. The listing for the code is given below.

```
 1  #include <stdio.h>
 2  #include <math.h>
 3  #include <stdlib.h>
 4
 5  double p(double x)
 6  {
 7    return exp(-x*x/2);
 8  }
 9
10  double getrnd()
11  {
12    static double x=10;
13    double step = 1.0;
14    double xp = x + step*2*(drand48()-0.5); // this is T0
15
16    double r = drand48();
17
18    if(p(xp)/p(x) > r) x = xp;
19
20    return x;
21  }
22
23  int main(int argc, char** argv)
24  {
25    int N = atoi(argv[1]);
26    for(int i=0; i<N; ++i) printf("%e\n", getrnd());
27    return 0;
28  }
```

The Metropolis algorithm appears in the `getrnd()` function. First a new proposal is generated on line 14. Note that this is done by shifting the current value of x by a random amount equally distributed between `-step` and `step`. This proposal is symmetric since two points x, y that are too far from each other, $|x-y| >$ `step`, have $T_0(x \to y) = T_0(y \to x) = 0$. If they are close by, then $T_0(x \to y) = T_0(y \to x) = 1/2$`step` [1]. We accept the proposal based on the probability ratio $p(xp)/p(x)$ on line 18. Note that there we fused two cases: where the ratio is greater than 1 the condition is satisfied independently of the value of r since r is always smaller than 1. If $p(xp) < p(x)$ then the acceptance is stochastic with the correct probability.

One coding details to note is that we use `static` variables on line 12 as a way to retain state between different calls to the `getrnd` functions. The value of x is set to 10 only the first time the function is called. On subsequent calls the value of `x` is left unchanged between calls.

Using Mathematica we produce histograms for the $100,000$ numbers generated by this code and compare them with the expected PDF. The results are shown in Fig. 1, for $p(x)$ corresponding to uniform distribution between 0 and 1, for the normal distribution that appears in the listing above, and for $p(x) = \exp(-x^2/2) + \exp(-(x-8)^2/8)$. Note that for the uniform distribution starting at `x=10` produces the wrong results and we need to place the starting point somewhere within the range $\texttt{x} \in (-1, 2)$. Do you know why?

One important aspect in Monte-Carlo simulation is that successive numbers generated by the process are not statistically independent, the process has *memory*. For example in Fig. 2 we plot the numbers produced by our code when the probability $p(x)$ is a Gaussian. We can easily see the memory effect that is due to the fact that the step size in the proposal step is of max size 1. Since the value we start at $x = 10$ is

[1] In this case we are looking at transition probability *densities* since the probability outcomes are continous.

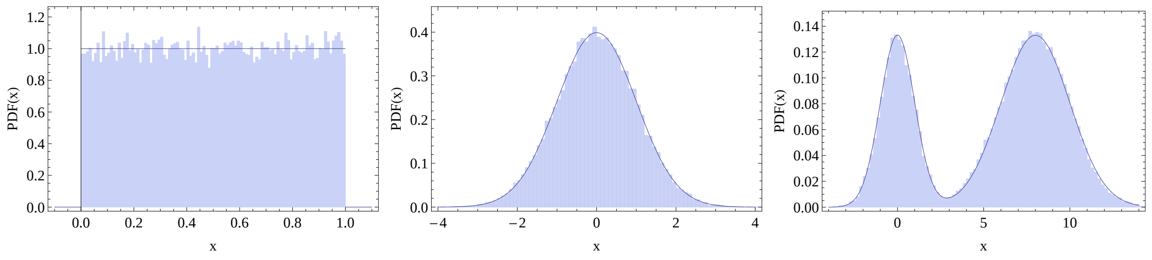


Figure 1: The histograms for the PDFs generated by our implementation of the Metropolis algorithm compared to expectations.

atypical for this distribution, in the beginning the process drifts towards the region in the parameter space that is typical. In the figure on the right we see that it takes of the order of 50 iterations to drift from the initial state to the typical region. This is usually called the *thermalization* time. We also note that even after we get to the typical region the process continues to exhibit a memory. This can be easily seen when comparing this with the right panel in the figure where successive data is uncorrelated. Metropolis process takes a few steps, of the order 10, to become statistically independent (this is called the *auto-correlation* time.) To confirm this, in the center panel we plot the Metropolis data where we skipped the first 100 steps, to allow the system to thermalize, and then plotted every tenth update; the plot looks much closer to the one generated by a statistically independent process. Note that the thermalization time and autocorrelation times depend on the system that is simulated, the proposal mechanism, and the observable that we monitor. For your problem you should monitor the observables of interest and skip over the measurements produces doing thermalization, and also only skip over a number of steps between measurements to insure that they are statistically independent.

Note that the thermalization time and autocorrelation time are related to the size of the proposal steps. It seems clear that larger step sizes should move you more effectively in the configuration space. Unfortunately, if the steps are too large then the majority of proposals would lead to large changes in energy and then most of the proposals will be rejected. This will lead again to an increase in correlation, since for most iterations the configuration is unchanged. As a rule of the thumb, you should chose a proposal mechanism that has about 50% chance of being accepted. In fact, anywhere between 20% and 80% is fine, so do not spend too much time optimizing this. You should include a counter in your code to measure the acceptance rate and make sure it is in the interval above.

4 Metropolis for interacting gas

The strategy to sample the interacting gas according to the canonical partition function, $P(y) \propto \exp(-E(y)/kT)$, is going to be similar to the one used above. The state of the vector is encoded by the vector `y` that holds

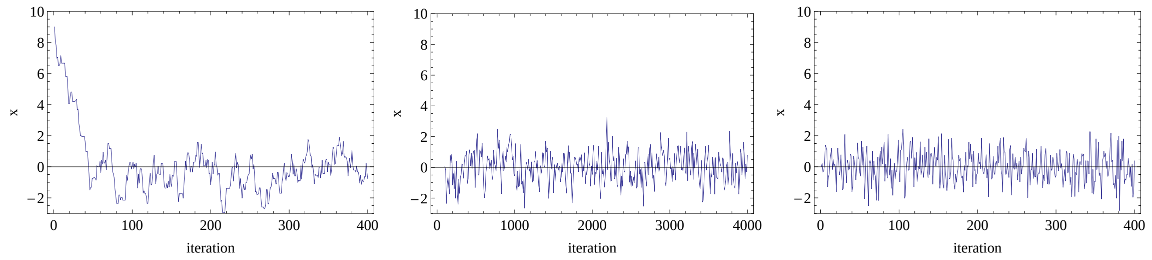


Figure 2: The numbers generated by the Metropolis process for the normal distribution when no intermediate steps are skipped (left), when the first 100 steps are skipped for *thermalization* and 10 steps are skipped between each measurement for *decorrelation* (center), and the numbers produced by an efficient algortihm that produces statistically independent samples (right).

both particle positions and velocities, using the same layout as before $x_1, y_1, x_2, y_2, \ldots, v_{x1}, v_{y1}, v_{x2}, v_{y2}, \ldots$. We will update the state one particle at time, and accept/reject the proposal based on the probability ratio.

The update process for velocity for particle `i` is the following. Propose

$$v'_{xi} = v_{xi} + \delta v_x\,, \qquad v'_{yi} = v_{yi} + \delta v_y\,, \tag{16}$$

with $\delta v_{x,y}$ two random numbers from a uniform distribution in the interval $[-\Delta v, \Delta v]$. The accept/reject step needs the probability ratio

$$\frac{P(y')}{P(y)} = \frac{e^{-E(y')/kT}}{e^{-E(y)/kT}} = e^{-(E(y')-E(y))/kT}\,. \tag{17}$$

Since we only changed the velocity of one particle the change in energy is simply $E(y') - E(y) = \frac{m}{2}(\boldsymbol{v}'^2 - \boldsymbol{v}^2)$.

To update the positions we propose

$$x'_i = x_i + \delta x\,, \qquad y'_i = y_i + \delta y\,, \tag{18}$$

with $\delta x, \delta y$ two random numbers from a uniform distribution in the interval $[-\Delta x, \Delta x]$. The accept/reject depends on the change in energy, which in this case is related to the potential energy alone, since the velocities do not change. We have

$$E(y') - E(y) = \sum_{j<k}[V(r'_{jk}) - V(r_{jk})] = \sum_{j\neq i} V(r'_{ij}) - \sum_{j\neq i} V(r_{ij})\,. \tag{19}$$

Note that since we only changed the position of particle `i`, the potential energy due to pairs that do not involve particle `i` is unchanged. It is then advisable to compute only the contribution to the system potential energy of the pairs that involve particle `i`. This calculation involves only $N-1$ steps, compared to the $N(N-1)/2$ for the full potential, and it is much quicker to compute. The implementation of this contribution is given in the listing below:

```
double compute_pot_one(double* y, int nbody, int j)
{
  double res=0;
  for(int i=0; i<nbody; ++i) if(i!=j)
  {
    double r[2] = {y[2*i+0] - y[2*j+0], y[2*i+1] - y[2*j+1]};
    double d = hypot(r[0], r[1]);
    res += pot(d);
  }
  return res;
}
```

The parameters Δv and Δx are inputs to the code and are adjusted so that the acceptance rate is reasonable. The main routine will then read in the parameters, update the configuration, and print out the state of the system every few steps. The code is similar to the one we implemented last week for molecular dynamics

```
int main(int argc, char** argv)
{
  if(argc < 6)
  {
    printf("Usage: %s N L kT dx dv nmeas nskip\n", argv[0]);
    return -1;
  }
  NBODY = atoi(argv[1]);
  double L = atof(argv[2]);
  double kT = atof(argv[3]);
11    double dx = atof(argv[4]);
12    double dv  = atof(argv[5]);
13    int nmeas = atoi(argv[6]);
14    int nskip = atoi(argv[7]);
15
16    posprop = posacc = velprop = velacc = 0;
17
18    double *y;
19    y = malloc(NBODY*4*sizeof(double));
20
21    init_positions(y, L);
22    init_velocities(y, NBODY, mass, kT);
23
24    for(int i=0; i<nmeas; ++i)
25    {
26      for(int j=0; j<nskip; ++j) update(y, dx, dv, kT, L);
27
28      for(int k=0; k<4*NBODY; ++k) printf("%20.15e ", y[k]);
29      printf("\n");
30    }
31
32    free(y);
33
34    fprintf(stderr, "velocity acc: %g  position acc: %g\n",
35            ((float) velacc)/velprop, ((float) posacc)/posprop);
36
37    return 0;
38  }
```

Note that we only print the state of the system every `nskip` steps. The total number of measurements is `nmeas` and the parameters `dx` and `dv` correspond to Δx and Δv. The first three parameters fix the physical properties of the system. On line 16 we initialize update and acceptance counters for the velocity and position proposal. These counters will be incremented in the update routine and a final diagnostic, the acceptance rate, is printed on line 34. Note that we print this diagnostic on `stderr`, the error stream. When running from terminal this text will be printed on the console, but when we read the output of the program in Mathematica, or when we redirect to a file, the standard error stream is separated and does not interfere. The initial state of the system can be set arbitrarily, since the update is expected to eventually make the system assume typical gas configurations. We set the system initially in a state that has the positions on a grid, a crystal like structure, and velocities typical what you expect for a gas at temperature T.

```
void init_positions(double* y, double L)
{
  int nside = ceil(sqrt(NBODY));

  double step = pow(2,1.0/6)*r0;
  if(step*nside > L) step = L/nside;

  for(int i=0, n=0; i<nside; ++i)
  for(int j=0; (j<nside) && (n<NBODY); ++j, ++n)
  {
    y[2*n+0] = i * step;
    y[2*n+1] = j * step;
  }

}
```

```
16
17  void init_velocities(double* y, int nbody, double mass, double kT)
18  {
19    double vel = sqrt(2*kT/mass);
20    for(int i=0; i<nbody; ++i)
21    {
22      double ang = 2*M_PI*drand48();
23      y[2*nbody+2*i+0] = vel*sin(ang);
24      y[2*nbody+2*i+1] = vel*cos(ang);
25    }
26  }
```

Finally, the update code is given below. Note that we loop over particles first and apply a Metropolis step for each particle velocity. A second loop updates particles positions. One thing to discuss is the code on line 96–102: the proposals could move particles outside the box. In that case, the proposal is rejected, and we move on updating the next particle.

```
1   int velprop, velacc, posprop, posacc;
2
3   void update(double* y, double dx, double dv, double kT, double L)
4   {
5     for(int i=0; i<NBODY; ++i)
6     {
7       velprop++;
8       double oldke = (pow(y[2*NBODY+2*i+0],2)+pow(y[2*NBODY+2*i+1],2))*
           mass/2;
9       double nvx = y[2*NBODY+2*i+0] + dv*2*(drand48()-0.5);
10      double nvy = y[2*NBODY+2*i+1] + dv*2*(drand48()-0.5);
11      double newke = (pow(nvx,2)+pow(nvy,2))*mass/2;
12      double r = drand48();
13      if( exp(-(newke-oldke)/kT) > r ) //accept
14      {
15        y[2*NBODY+2*i+0] = nvx;
16        y[2*NBODY+2*i+1] = nvy;
17        velacc++;
18      }
19    }
20
21    for(int i=0; i<NBODY; ++i)
22    {
23      posprop++;
24      double oldpot = compute_pot_one(y, NBODY, i);
25      double ox = y[2*i+0];
26      double oy = y[2*i+1];
27      y[2*i+0] = ox + dx*2*(drand48()-0.5);
28      y[2*i+1] = oy + dx*2*(drand48()-0.5);
29      if( (y[2*i+0] < 0) || (y[2*i+0] > L) ||
30          (y[2*i+1] < 0) || (y[2*i+1] > L) ) // out of box. reject
31      {
32        y[2*i+0] = ox;
33        y[2*i+1] = oy;
34        continue;
35      }
36      double newpot = compute_pot_one(y, NBODY, i);
37      double r = drand48();
```

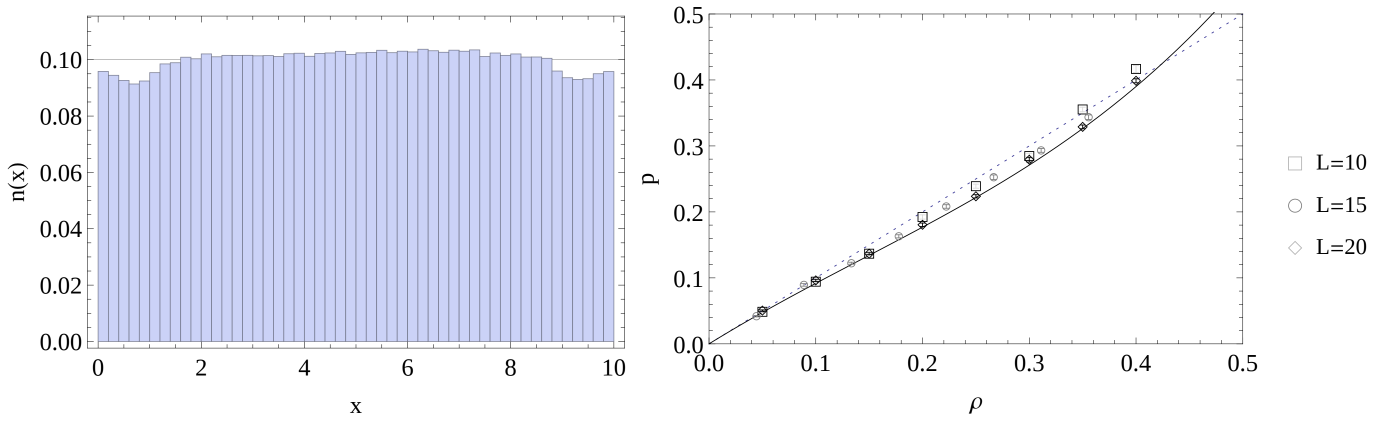


Figure 3: Left: histogram of particle position for a simulation of 10 particles in a 10×10 box at temperature $kT = 1$. The horizontal line is the expected value for uniform distribution $n = N/A = 10/10^2$. Right: Pressure as a function of density ($\rho = n$) for $kT = 1$ from numerical simulation. The dashed line is the ideal gas result and the solid line is the virial expansion.

```
      if(r > exp(-(newpot-oldpot)/kT)) // reject
      {
        y[2*i+0] = ox;
        y[2*i+1] = oy;
      }
      else posacc++;
    }
}
```

Before we conclude, we would like to show a bit of physics that you can study using this code. The ideal gas equation of state is $pA = NkT$ (A plays the role of volume in two dimensions.) For an interacting gas this equation receives corrections which we can systematically compute, at least for small densities $n = N/A$. We have

$$\frac{p}{kT} = n + \alpha n^2 + \beta n^3 + \gamma n^4 + \dots\,, \tag{20}$$

with coefficients α, β, γ being functions of temperature. For $kT = 1$, for the potential we used in this assignment, we can evaluate these *virial* coefficients using Mayer cluster expansion to be $\alpha = -1.07347$, $\beta = 2.427$, $\gamma = 0.25$.

We plan to compare this predictions with our numerical results. The first question is how to measure the pressure. One way to do it is to compute the *virial* which produces accurate results, but requires a bit of justification. A more straightforward way to do it, that also contains an important bit of physics, is by measuring the density. To evaluate the pressure imparted on a surface at any point in the gas, we can take a snapshot of the system and evolve it for a short time dt. Since the velocity distribution is known and does not depend on position, what you find is that the pressure the surface experiences is just $p = nkT$. If the density is uniform in the gas, then $n = N/A$ and we get the ideal gas law. In Fig. 3, left panel, I plot the histogram of particle position along the x direction for a gas of 10 particles in a 10×10 box at temperature $kT = 1$. As we can see, the density is not uniform and it changes close to the walls. This has to do with the fact that the particles next to the walls experience a pull (or push) from the particles in the bulk. For particles in the middle of the box, they are pulled equally in all directions. At low densities, the particles are far from each other and the most dominant interaction piece is the attractive tail. In this case the density close to the walls is lower than in the middle. At higher densities, when the particles are close to each other and the repulsive cores dominate, the density close to the walls will be larger.

We can use the state of the system to figure out what the density is close to the walls. I did this for a small region within a distance of 0.5 from the wall. Using this density I can compute the pressure for different densities and compare it with the virial expansion. This is shown in right panel of Fig. 3. Note that for a 10×10 box the data does not agree very well with the virial expansion. This has to do with the fact that on small boxes finite-volume effects are large. If we repeat the calculation with a larger box, and a correspondingly larger number of particles, the agreement improves. By the time we use a 20×20 box the agreement between our simulation result and the theoretical expectations is very good.

References

[1] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.