Computational Physics – Lecture 8

Andrei Alexandru

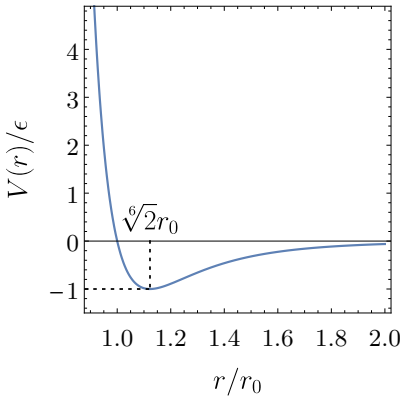
April 8, 2019

1 Introduction

This week we will discuss molecular dynamics simulations: we will model a gas of particles using the Lennard-Jones potential and integrate the equations of motion. From a statistical mechanics point of view, this type of simulation generate configurations (that is, particle positions and velocities) according to the *microcanonical ensemble*, a statistical ensemble corresponding to a system kept at constant energy. Next project we will attack the same problem, a two-dimensional gas of particles, using the *canonical ensemble*, corresponding to systems kept at constant temperature.

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. The force on particle i is then

$$\boldsymbol{F}_i = -\nabla_i U = \sum_{j \neq i} \boldsymbol{F}_{i \leftarrow j} \tag{3}$$

where $\boldsymbol{F}_{i \leftarrow j}$ is the force due to particle j on particle i:

$$\boldsymbol{F}_{i \leftarrow j} = 24\epsilon \left[2 \left(\frac{r_0}{r_{ij}}\right)^{12} - \left(\frac{r_0}{r_{ij}}\right)^{6} \right] \frac{\boldsymbol{r}_{ij}}{r_{ij}^2} \,. \tag{4}$$

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

$$PV = NkT \,, \tag{5}$$

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. In our simulations, we measure the pressure by monitoring the particles that bounce off the wall and computing the linear momentum imparted to the walls per unit time. This corresponds to the average force on the wall and the pressure can be easily evaluated from this information.

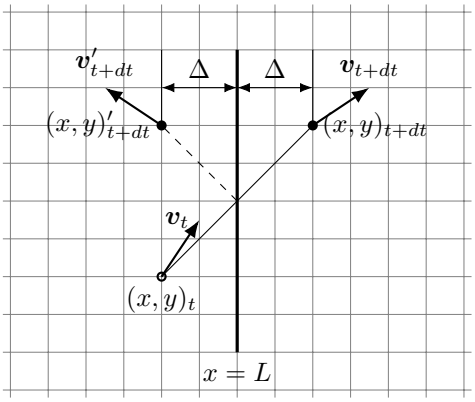
Our interest is in the properties of the system in the *thermodynamic limit*, that is the limit where the bulk effect due to the volume of the gas are dominant. The presence of the walls introduces *finite volume* effects that only vanish when the size of the box is taken to infinity. More precisely, the finite volume effects actually scale up with the size of the walls (proportional with L in the case of an $L \times L$ box) but this become subdominant when compared with the effects due to the volume which scale up as L^2. Thus, if we look at *intensive parameters*—like pressure, temperature, particle and energy density—the finite volume corrections vanish in the limit $L \to \infty$. Since we cannot do simulations in the infinite volume, the usual strategy is to carry out simulations on ever increasing volumes, until the parameters of interest converge to their infinite volume limit within the desired precision.

Before discussing the implementation details, we note that for molecular dynamics simulations we fix the total energy of the system. The initial energy remains the same, as long as we integrate the equations of motion correctly. During evolution the potential and kinetic energy change, but their sum remains fixed. If we want to measure the temperature of the system, we can determine it form the average kinetic energy. A theorem of statistical mechanics states that the average kinetic energy for a d-dimensional gas is $\langle K \rangle = \frac{d}{2} kT$. For two dimensions this implies that $kT = \langle K \rangle$.

2 Implementation details

To implement molecular dynamics evolution we use the integrators we developed in week 4. The only changes have to do with how we initialize the state of the system and the implementation of the force term.

One important difference is how we deal with the *boundary conditions*. For the case studied in week 4 the interaction was attractive and we used initial conditions that led to a bound system, that is a system where the motion stays in a finite range around the center of mass. For the case studied this week, we will use initial conditions that lead to unbounded motion. This is not a problem for the system we try to simulate, because the particles are reflected back by the walls. This is accomplished by interleaving the regular time integration for time step `dt`, with a process where all particles that have moved outside the box, are "reflected" inside the box. A typical situation is shown in the graph on the right where the particles crosses the wall at $x = L$. To fix by hand the reflection we move $\boldsymbol{x}_{t+dt} \to \boldsymbol{x}'_{t+dt}$ and $\boldsymbol{v}_{t+dt} \to \boldsymbol{v}'_{t+dt}$ with



$$x'_{t+dt} = L - \Delta = L - (x_{t+dt} - L)\,, \qquad v'_{t+dt} = -v_{t+dt}\,, \tag{6}$$

with the y components for both position and velocities left unchanged. This also allows us to compute the impulse imparted by this collision on the wall: $\Delta I = -(m\boldsymbol{v}' - m\boldsymbol{v}) = -2mv'_x$. Similar relations can be derived for the other three walls.

To make the particles collide elastically with the walls, we have to modify the integration loop from the `nbody.c` code we wrote in week 4. Below is the relevant code listing. We note that in the listings included here we assume that the configuration of the system is encoded in an array of length $4 \times$ NBODY with the layout: $x_0, y_0, x_1, y_1, \ldots, v_{x0}, v_{y0}, v_{x1}, v_{y1}, \ldots$, where (x_i, y_i) is the position for particle i and (v_{xi}, v_{yi}) its velocity.

```
1  double pressure[4] = {0,0,0,0};
2
3  init_positions(y, L);
4  init_velocities(y, EperN);
5
```

```
6     double t=0;
7     for(int i=0; i<nstep; ++i)
8     {
9       integrate_rk2(y, 4*NBODY, t, dt, yn, F);
10
11      // check if any particle is outside the box and fix it
12      for(int i=0; i<NBODY; ++i)
13      {
14        if(yn[2*i+0] < 0)
15        {
16          yn[2*i+0] *= -1;
17          yn[2*NBODY+2*i+0] *= -1;
18          pressure[0] += 2*yn[2*NBODY+2*i+0];
19        }
20        if(yn[2*i+0] > L)
21        {
22          yn[2*i+0] = L - (yn[2*i+0] - L);
23          yn[2*NBODY+2*i+0] *= -1;
24          pressure[1] += -2*yn[2*NBODY+2*i+0];
25        }
26        if(yn[2*i+1] < 0)
27        {
28          yn[2*i+1] *= -1;
29          yn[2*NBODY+2*i+1] *= -1;
30          pressure[2] += 2*yn[2*NBODY+2*i+1];
31        }
32        if(yn[2*i+1] > L)
33        {
34          yn[2*i+1] = L - (yn[2*i+1] - L);
35          yn[2*NBODY+2*i+1] *= -1;
36          pressure[3] += -2*yn[2*NBODY+2*i+1];
37        }
38
39      }
40      if((i+1)%nskip == 0)
41      {
42        printf("% .5f ", t+dt);
43        for(int k=0; k<4*NBODY; ++k) printf("%20.15e ", yn[k]);
44        for(int k=0; k<4; ++k) printf("%20.15e ", pressure[k]);
45        printf("\n");
46      }
47      for(int k=0; k<4*NBODY; ++k) y[k] = yn[k];
48      t = t + dt;
49    }
```

The calls to `init_positions` and `init_velocities` sets the initial configuration of the system such that the total energy of the system is E. We will discuss this procedure in detail latter. The relevant part for boundary conditions appears on lines 11–37: after the configuration of the system is advanced in time with dt we check whether any particle is outside the box and loop it back in. Note that we assume here that the time interval dt is small enough that if any particle is out of the box, the distance from the edge is smaller than L.

If the equations of motion are integrated exactly, the total energy of the system is conserved. We only have to arrange the initial state to have the desired energy. One ingredient is the routine that computes the potential energy of the system. To compute it we sum up the potential due to all distinct pairs. The code is listed below.

```
1  double pot(double r)
2  {
3    return 4*eps*(pow(r0/r,12)-pow(r0/r,6));
4  }
5
6  double force(double r)
7  {
8    return 24*eps*(2*pow(r0/r,12)-pow(r0/r,6))/r;
9  }
10
11 double compute_pot(double* y)
12 {
13   double res=0;
14   for(int i=0; i<NBODY; ++i)
15   for(int j=i+1; j<NBODY; ++j)
16   {
17     double r[2] = {y[2*i+0]-y[2*j+0], y[2*i+1]-y[2*j+1]};
18     double d = hypot(r[0], r[1]);
19     res += pot(d);
20   }
21   return res;
22 }
```

We note that we loop over all possible pairs with $i < j$ and that we compute the energy as the sum of each pair contribution according to the Lennard-Jones formula in Eq. 1. The proper distance is evaluated on line 18, where the routing `hypot(x, y)` is used that returns $\sqrt{x^2 + y^2}$.

A similar procedure is used to compute the distance used in the definition of the force. Note that the function `F` listed below computes the right hand side of the first order differential equation $\dot{\boldsymbol{y}} = \boldsymbol{F}(\boldsymbol{y}, t)$ that is required by our integrator, not the total force of the system (see discussion in week 4 about how to turn second order differential equations into a system of first order ones).

```
1  void F(double* y, double t, double* res)
2  {
3
4    for(int i=0; i<2*NBODY; ++i) res[i] = y[2*NBODY+i];
5
6    for(int i=0; i<NBODY; ++i)
7    {
8      res[2*NBODY+2*i] = 0;   // vx_i
9      res[2*NBODY+2*i+1] = 0; // vy_i
10     for(int j=0; j<NBODY; ++j) if(j!=i)
11     {
12       double r[2] = { y[2*j] - y[2*i], y[2*j+1]-y[2*i+1] };
13       double magr = sqrt(r[0]*r[0]+r[1]*r[1]);
14       double f = force(magr);
15       res[2*NBODY+2*i]   -= f*r[0]/magr/mass;
16       res[2*NBODY+2*i+1] -= f*r[1]/magr/mass;
17     }
18   }
19 }
```

The force on object i is computed in the inner for loop, by summing the contribution due to all other particles ($j \neq i$).

We turn now to discussing how to set the initial conditions. In these simulations it is desirable to select the total energy of the system as an input parameter. Our only goal is to find a configuration that has the desired energy. The particular initial configuration is not very important since the system will evolve toward

equilibrium. Our strategy is the following: we will try to arrange the particles in a configuration that has close to minimal potential energy and then put the remainder energy into the kinetic energy. To get the particles in a state with close to minimal energy we will arrange them initially on a two dimensional grid with step-size $r_{\min} = \sqrt[6]{2}r_0$. The remainder energy $K = E - U$ is then divided equally over all particles and the corresponding initial velocity is then $v = \sqrt{2(K/N)/m}$. For every particle we chose an random direction for this velocity. The code implementing this strategy is listed below.

```
1  void init_positions(double* y, double L)
2  {
3    int nperside = ceil(sqrt(NBODY));
4    double step = r0*pow(2.0,1.0/6);
5    if(r0*nperside > L) step = L/nperside;
6
7    for(int i=0, n=0; i<nperside ; ++i)
8    for(int j=0; (j<nperside) && (n<NBODY); ++j, ++n)
9    {
10     y[2*n+0] = i*step;
11     y[2*n+1] = j*step;
12   }
13
14 }
15
16 void init_velocities(double* y, double EovN)
17 {
18   double U = compute_pot(y);
19   double KovN = EovN - U/NBODY;
20   if(KovN < 0)
21   {
22     printf("Cannot find a state with EovN: %e ... exiting\n", EovN);
23     abort();
24   }
25
26   double v = sqrt(2*KovN/mass);
27
28   for(int i=0; i<NBODY; ++i)
29   {
30     double ang = drand48()*2*M_PI;
31     y[2*NBODY+2*i+0] = v*cos(ang);
32     y[2*NBODY+2*i+1] = v*sin(ang);
33   }
34 }
```

The last aspect to discuss here is the calculation of the pressure. We compute the momentum imparted to the wall by each particle that collides with it. Note that in the case of a bounce, it is only the component perpendicular on the wall that would flip, and the momentum flip is due to the force imparted on the wall. We then create four counters, one for each wall, and modify the wrap around the box code to keep track of the bounces. The listing is given above, when presenting the main integration loop. In the routine that "fixes" the position of the particles that crossed the walls, the imparted momentum is accumulated in 4 different `pressure` counters, that keep track of each wall separately. These 4 `pressure` counters are printed at the end of each status line. Our Mathematica code needs to be adjusted accordingly, to make sure each status line is parsed correctly.