Building a molecular dynamics code

In this tutorial, you will build up a molecular dynamics code of a simple atomistic system – the famous Lennard-Jones model. This model captures the essential features of interactions between atoms that are not charged, and that do not form chemical bonds. (A noble gas like Argon is a real-world example of such a system.) Despite its simplicity, the Lennard-Jones model displays phase transitions between a gas, fluid, and solid, and is well suited to illustrate some of the questions that can be answered with computer simulations.

Detailed explanations of this material can be found in the book by Frenkel and Smit, "Understanding Molecular Simulation".

1 Create an initial configuration

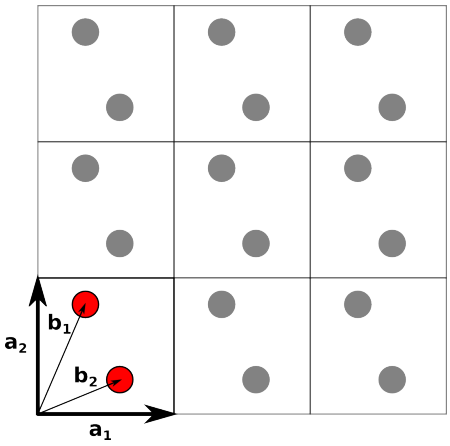

Figure 1: An example of a crystalline configuration in 2D. The unit cell is a rectangular region of space spanned by the lattice vectors $\mathbf{a}_1$ and $\mathbf{a}_2$. The unit cell contains two basis atoms (red circles), located at positions $\mathbf{b}_1$ and $\mathbf{b}_2$ with respect to the origin of the unit cell. An extended crystal configuration can be produced by translating the unit cell in space, along the two lattice vectors.

A crystal lattice is a periodic arrangement of points in space. The repeat unit is called a unit cell, a region of space spanned by three lattice vectors (in three dimensions). We are going to call them $\mathbf{a}_1$, $\mathbf{a}_2$, and $\mathbf{a}_3$. The unit cell contains one or more points that mark the position of particles (atoms, molecules, colloidal particles, etc.). These points can be specified by basis vectors $\mathbf{b}_i$, one for each particle. By periodically replicating the unit cell in space, one can creat extended crystalline arrangments. Figure 1 illustrates this concept for a particular lattice in two dimensions.

The simplest 3D lattice is the simple cubic lattice. It's basis vectors are $\mathbf{a}_1 = (a, 0, 0)$, $\mathbf{a}_2 = (0, a, 0)$, and $\mathbf{a}_3 = (0, 0, a)$, where a is the lattice constant. There is only one atom in the

unit cell. We can freely choose its location in the unit cell (why?). For instance, we can put it right in the origin of the unit cell, $\mathbf{b}_1 = (0, 0, 0)$.

A particulalry useful lattice to be familiar with is the face-centered cubic lattice (FCC). It represents the densest possible packing of spheres. Many metals crystallize in this structure. The unit cell of the FCC lattice has the same lattice vectors as the simple cubic structure. However, it has four atoms in the unit cell, located at $\mathbf{b}_1 = (0, 0, 0)$, $\mathbf{b}_2 = (a/2, a/2, 0)$, $\mathbf{b}_3 = (a/2, 0, a/2)$, and $\mathbf{b}_4 = (0, a/2, a/2)$.

Problem

Write a program that produces a set of atomic coordinates that represent a face-centered cubic (FCC) crystal, by replicating the FCC unit cell m times in each direction of space. (In Figure 1, $m = 3$.) This will result in a cubic "box" of edge length $L = ma$, with a total number of m^3 unit cells, and a total number of $N = 4m^3$ particles. (Why?)

The user should be able to specify two parameters: the lattice constant a and the number m. The program should store the positions of atoms in a two-dimensional array `double pos[N][3]`, a matrix with N lines (one for each particle) and three columns (one for each coordinate). It is usually best to define `pos` as a pointer (i.e., `double **pos`) and allocate memory dynamically using the function `malloc`.

To check your code, write the configuration to a file called "`pos.xyz`". The `.xyz` file format looks like this:

$$
\begin{array}{llll}
N & & & \\
 & & & \\
C & x_1 & y_1 & z_1 \\
C & x_2 & y_2 & z_2 \\
\vdots & & & \\
C & x_N & y_N & z_N
\end{array}
$$

The first line contains the number of particles N. The second line is blank. Starting with the third line, each line specifies the coordinates of a single atom, preceded by a character that indicates the elemental species of the atom. (This character, set to "C" for carbon in the example above, determines the color and size of spheres that a molecular viewing program might draw.)

Install the program VMD on your computer and visualize the configuration, for instance by starting VMD with the command `vmd pos.xyz`. Familiarize yourself with VMD and its different representation styles; "VDW" and "Dynamic Bonds" are among the most useful ones. Render a snapshot of the configuration with `tachyon`, which comes with `VMD`.

2 Calculating the energy between two particles

Consider two particles at positions $\mathbf{r}_1 = (x_1, y_1, z_1)$ and $\mathbf{r}_2 = (x_2, y_2, z_2)$. The Lennard-Jones model assigns these particles a potential energy of

$$u(r) = 4\epsilon \left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right).$$

Here, $r = |\mathbf{r}_2 - \mathbf{r}_1|$ is the distance between the particles, ϵ is the attraction strength, and σ is the particle diameter. The force on particle 1 described by this potential can be calculated by

taking the negative gradient,

$$\mathbf{F}_1 = -\nabla_{\mathbf{r}_1} u(r),$$

where $\nabla_{\mathbf{r}_1} = \left(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial y_1}, \frac{\partial}{\partial z_1}\right)$. The force on particle 2 can be similarly obtained by taking derivatives with respect to the coordinates of particle 2. According to Newton's laws, $\mathbf{F}_1 = -\mathbf{F}_2$.

Problem

Plot $u(r)$ using a plotting tool like gnuplot or python's matplotlib. Determine analytically the position of the minimum $r_{\min}$ and the corresponding energy value $u(r_{\min})$. Compute the analytic expression for the force. Write a function `double compute_pair_en(double *p1, double *p2, double eps, double sig, double *f1, double *f2)` that takes as arguments the positions of two particles (in the form of the two pointers `p1` and `p2`) and returns the corresponding potential energy. The function should aslo calculate the forces on particle 1 and 2 and store them using the two pointers `f1` and `f2`. To test your function, calculate the energy and forces between two particles at distances between 0.9σ and 3σ, for $\epsilon = 1$. (A good choice is to put particle 1 in the origin and particle 2 on the x-axis. That way, there is only one non-zero component of the force, which is convenient for plotting.) Write the data into a file and plot it, comparing it to your analytical solution.

3 Calculating the total potential energy

Once you are confident that your function for calculating the energy and forces for a pair of particles is working, we are ready to calculate the total potential energy of a larger system. The potential energy U of a system of N Lennard-Jones particles is given by the sum over all pair-energies,

$$U = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} u(r_{ij}),$$

where r_{ij} is the distance between particles i and j. The total force $\mathbf{F}_i$ on particle i is just the sum of all pair forces involving particle i,

$$\mathbf{F}_i = \sum_{j=1,\, j\neq i}^{N} -\nabla_{\mathbf{r}_i} u(r_{ij})$$

Problem

In your `main` function, allocate memory for a two-dimensional array `double force[N][3]`. Write a function `double compute_en(double **pos, double eps, double sig, double **force)` that computes and returns the total potential energy, and stores all forces in the array `force`. When the function returns, `force[3][0]`, for instance, should have the value of the x-component of the total force on particle 4 (remember, we count from 0 in C!).

Apply this function to calculate the energy of FCC crystal configurations as obtained from the function you previousl wrote. Do two things: First, plot the energy per particle, in units of epsilon, for a fixed number of particles $N = 500$, as a function of the lattice constant a. Vary a in the interval $[0.9\sigma/\sqrt{2}, 3\sigma]$. Plot the results and explain them! Second, for fixed $a = 2\sigma$, vary the number of particles between 4 and 13500. Plot the energy per particle as a function of N. Explain your result!

4 Simulating a bulk system

Often, we are interested in the behavior of atoms in the bulk, far removed from any surfaces. There is a simple trick of the trade which allows one to simulate such behavior without the need for extremely large particle numbers. This trick is called "nearest image convention" (NIC).

Assume that we are dealing with a cubic simulation box filled with particles. Every particle in the box interacts with all other particles via the pair potential $u(r)$. However, while particles close to the center of the box are in an isoptropic environment, a particle close to the boundary of the box experiences forces that predominantly come from particles located towards the center of the box. This is the essence of surface tension and the origin of much interesting behavior in nanoscience. To avoid surface effects and equip a surface particle with bulk behavior, we need to supplement forces that seem to come from particles outside of the simulation box. To this end, we imagine all of space filled with periodic replicas of our simulation box, as illustrated in Figure XXX. When calculating the force on particle A that stems from particle B, we don't

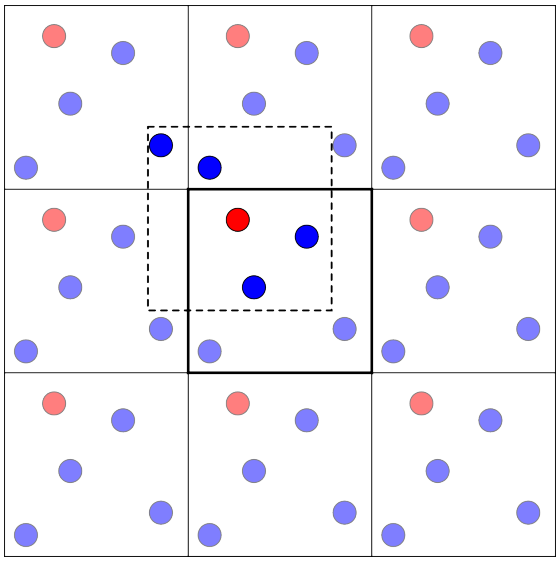

Figure 2: Nearest image convention. We imagine the original simulation box (shown as the central bold square) to be surrounded by exact copies of itself. A given particle in the box (red) interacts with the nearest images of the other particles. In essence, we imagine each particle to be located in the center of its own simulation box (dashed square), interacting with all particles that are located inside that box.

necessarily take the copy of particle B that is in the original box, but the copy that is closest to particle A, the "nearest image". Practically, this is achieved by making sure that in the force calculation all components of the distance vector $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ lie in the interval $[-L/2; L/2]$, where L is the edge length of the simulation box. This can be compactly written as (for the x-component of $\mathbf{r}_{ij}$)

$$x = x - L\,\mathrm{rint}(x/L).$$

Here, the function `rint()` returns the value of a floating point number rounded to the nearest integer. (For instance, $\mathrm{rint}(0.7) = 1.0$.) Sometimes it is computationally cheaper to implement the NIC with an `if-else`-clause. Try to rewrite the code above using `if-else`.

Problem

Augment your function `compute_pair_en()` with code that implements the NIC. Repeat the calculation from the previous example, that is, for fixed $a = 2\sigma$, vary the number of particles N between 4 and 13500 and plot the energy per particle as a function of N. Compare your results with NIC and without NIC. Explain!

5 Dynamics!

It is finally time to make our particles jiggle. We can follow the particles' dynamics by numerically integrating Newton's equations of motion. That means that from knowledge of the positions $\mathbf{r}(t)$, velocities $\mathbf{v}(t)$, and forces, $\mathbf{f}(t)$ of all particles at a given time t, we are able to approximately calculate positions and velocities at time $t + \Delta t$. A simple numerical recipe that is widely used is the velocity-Verlet algorithm

$$\mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \mathbf{v}(t)\Delta t + \mathbf{f}(t)\frac{\Delta t^2}{2m} \quad (1)$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \frac{\Delta t}{2m}(\mathbf{f}(t) + \mathbf{f}(t + \Delta t)). \quad (2)$$

Here, Δt is called the time step, and m is the mass of the particle. Note that while integration of the positions $\mathbf{r}$ is straightforward, integration of the velocities $\mathbf{v}$ requires knowledge of the forces at time $t+\Delta t$, which can only be computed once we know the new positions $\mathbf{r}(t+\Delta t)$! In practice we thus need to implement the integration in two separate steps, with a force calculation in between. It is convenient to proceed in this manner, given $\mathbf{r}(t)$, $\mathbf{v}(t)$, and $\mathbf{f}(t)$:

1. Calculate the new positions $\mathbf{r}(t + \Delta t)$.
2. Calculate the "half-step" velocities $\mathbf{v}_{\frac{1}{2}} = \mathbf{v}(t) + \frac{\Delta t}{2m}\mathbf{f}(t)$.
3. Calculate the new forces $\mathbf{f}(t + \Delta t)$, using the new positions.
4. Calculate the new velocities $\mathbf{v}(t + \Delta t) = \mathbf{v}_{\frac{1}{2}} + \frac{\Delta t}{2m}\mathbf{f}(t + \Delta t)$.

Problem

Write a function `double md_step(...)` that implements the four steps described above! As arguments to the function, pass the fields `pos`, `vel`, and `force` (as well as any constant you might need), which hold the positions, velocities, and forces at the current time t. The function should overwrite these fields with the respective values at time $t + \Delta t$, and return the potential energy at time $t + \Delta t$.

6 Periodic boundaries

To be able to test your integrator, we need to put in place a few more pieces. One issue we need to fix is what happens to particles when their dynamics takes them out of the simulation box. Figure 2 has the answer to this question: If a particle leaves the box, one of its periodic images enters the box from the other side at the same time! In practice that means we need to check after each integration step if all particles are still in the box. If a particle has left, we put it back in on the other side of the simulation box, by adding or subtracting L from one or more of its coordinates.

Problem

Implement periodic boundary conditions. The natural place to do this is in your `md_step()` function, right after you obtain the new positions.

7 Initialization and First Run

If you haven't done so already, allocate memory for a two-dimensional array `double vel[N][3]`, which stores the velocities of the particles. Initialize the system by creating an FCC crystal with 256 particles and a lattice constant of $a = 2\sigma$. Set $\epsilon = 1$, $\Delta t = 0.001$, and $m = 1$ for all particles.

Initialize the velocities by assigning each of the $3N$ velocity components a random number in the interval $[-1; 1]$. A random number x between 0 and 1 can be produced by writing `x = (double) rand()/RAND_MAX`. Note that the numbers produced by `rand()` are not really random. While `rand()` produces a sequence of numbers from the right distribution and with very little correlation, the sequence is identical between different simulation runs! This arises because the random number generator is a deterministic function just like any other. This function takes a single argument, the "seed". Each value of the seed produces a different sequence of random numbers. If the user does not specify a seed, the generator uses a default value. To actually get different trajectories for simulations that start from identical initial conditions, we therefore need to use different seeds. These can be provided manually, but it is more convenient to use the time to achieve this. (Functions in the header file `<time.h>` might come in handy.) Go to the internet to find out how to seed your random number generator!

Calculate the initial energy and forces of this configuration. Write a loop in your main function that performs $n = 10000$ time steps, by repeatedly calling your function `md_step()`. After each time step, write the potential energy, the kinetic energy $E_{\rm kin} = \sum_i^N m_i \mathbf{v}_i^2/2$, and the sum of potential and kinetic energy to a file.

Problem

Implement theses instructions! Then, run your code and plot the energies as a function of time. While you should observe fluctuations in both the kinetic and potential energy, the sum of the two should stay constant to high accuracy. If your code does not work properly (which is highly likely for any code that runs for the first time!), see if you can spot the problem by "watching a movie", i.e., by writing the full configuration of the system into a `xyz`-file every time step and visualizing it with VMD.

8 Setting the center of mass momentum to zero

By randomly assigning velocities, the velocity of the center of mass (which is a constant of motion!) will generally have a non-zero value, $\mathbf{v}_{\rm CM} = \frac{1}{M}\sum_{i=1}^N m_i \mathbf{v}_i \neq 0$, where M is the total mass of all particles in the system. In other words, the particles in the simulation box will undergo a constant drift. To remove this drift, we need to set the center of mass motion to zero. This can be achieved by subtracting from all particle momenta the N-th part of the center of mass momentum,

$$\mathbf{v}_i \to \mathbf{v}_i - \frac{M}{N m_i}\mathbf{v}_{\rm CM}$$

Problem

Augment your code with a procedure that sets the center of mass motion to zero, right after you assign initial velocities.

9 Setting the temperature

Since we are dealing with Hamiltonian dynamics, the total energy of the system is conserved. It is frequently more convenient, however, to specify a desired temperature rather than total energy. The instantaneous temperature of the system can be computed from the kinetic energy via the equipartition theorem,

$$E_{\mathrm{kin}} = \sum_{i=1}^{N} \frac{m_i}{2} \mathbf{v}_i^2 = \frac{3}{2} N k_{\mathrm{B}} T.$$

Since the kinetic energy fluctuates, so does the temperature in the system. Therefore we can only set the average temperature in the system. Practically, we can achieve this by rescaling the velocities of all particles every few time steps at the beginning of the simulation, until equilibrium has been established. Let T_{inst} denote the instantaneous temperature in the system at a given instance, as computed via the equipartition theorem. You can convince yourself that T_{inst} can be set to be exactly equal to the desired temperature T by multiplying all velocity components by a suitable factor α,

$$\mathbf{v}_i \to \alpha \mathbf{v}_i$$

Problem

Determine the factor α! Write a function `void set_temp(double **vel, double temp)` that sets the instantaneous temperature to a desired temperature `temp` by implementing the procedure outlined above. To set the *average* temperature in your system to a value close to `temp`, you will need to call your function a few times during the first few thousand time steps of your simulations. A reasonable procedure might be to rescale the temperature every 100 time steps, for the first 10000 time steps. For $N = 256$ particles, a temperature of $T = 2.0$, and a density of $\rho = 0.6$, plot the kinetic, potential, and total energy per particle, as well as the instantaneous temperature for a trajectory of 20000 time steps length.

Let me know if you have any questions!