Homework 2

Due 2019 September 27

Report instructions

Please upload *one* PDF file to Gradescope (Entry code: ME62YG).

For this homework you will implement code (on PrairieLearn) to perform molecular dynamics simulations using periodic boundary conditions. You specifically should implement functions that determine the coordinates of a particle, assuming periodic boundary conditions, that determine the vector that connects two particles (as well as its length), and functions for forces and energies (please see PrairieLearn for details and instructions).

Periodic Boundary Conditions and Minimum Image Convention

Often, the goal of atomic simulations is to measure bulk properties of a system. "Bulk" means that one wishes to describe the many-body interactions of an infinite system (the so-called thermodynamic limit). A particle in a bulk system should interact with an infinite volume of particles around it; it cannot be at a surface or boundary, where it will experience different interactions.

Of course, we can simulate only a finite system on a computer, so we must make an approximation to the thermodynamic limit. Typically, the solution is to implement Periodic Boundary Conditions (PBC). This is similar to old video games where, when you come out of one side of the screen, you re-enter on the other side.

The effect of PBC is that a particle in the simulation box "sees" all other particles replicated on all sides of the simulation box, thus creating the effective interactions of a bulk system. A technical limitation of PBC is that there will be artificial spatial correlations on the order of the box size. However, it is a standard simulation technique and this artifact can be analyzed and mitigated by studying the effect of systematically increasing the system size.

It is important to ensure that you are enforcing PBC correctly; it can be tricky to get this right!

The Lennard-Jones Fluid

Once you have implemented the corresponding functions correctly, you will implement a molecular dynamics code to model a Lennard-Jones gas. The Lennard-Jones potential is:

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

Here, there is no external potential; the only force you need to calculate is from the pairwise interactions given by the Lennard-Jones potential. (Recall that the force $\vec{F} = -\nabla V = -V'(r)\hat{\mathbf{r}}$.)

For simplicity, we will work in reduced units (see e.g., p.40 of Frenkel & Smit). This means lengths are expressed in terms of σ and the temperature is in units of the Lennard-Jones energy parameter, ϵ. Thus, the reduced temperature is $T_* = \frac{k_B T}{\epsilon}$. Then the Lennard-Jones potential simplifies to

$V(r) = 4[(\frac{1}{r})^{12} - (\frac{1}{r})^{6}]$ and we need only specify a density and temperature at which to perform our simulation.

We will simulate $N = 64$ particles at a number density of $\rho = 0.8442$ and a reduced temperature of $T_* = 0.728$. These values correspond to a box side length of approximately 4.232317. As described in Frenkel & Smit (p. 66), we initialize the particles with random velocities, adjust the system to have zero net momentum, then rescale all velocities so the system starts at the desired temperature of 0.728. These initial conditions are already set in the code provided below.

Note: You are welcome to spend some time experimenting with other ways to initialize your simulation, but it is non-trivial. For example, starting all your particles at the same spot is a bad idea. The Lennard-Jones potential has a repulsive core, so your system will start with effectively infinite potential energy. Because you are working in the micro-canonical ensemble, energy is conserved, so the kinetic energy will explode and the simulation will be unstable.

Computing the Energy

You'll also want to get some output from your simulation. We're usually less interested in the precise positions of all the atoms as we are in some physical properties. The simplest one to examine is the energy. In case you've forgotten, let us remind you precisely what the energy means in this context: First, the energy of the system is the sum of the kinetic and potential energies. The kinetic energy is simply the sum over all atoms of $\frac{1}{2}mv_i^2$, where v_i is the velocity of the i-th atom and m its mass. The potential energy is the sum over all distinct pairs of atoms of the Lennard-Jones potential (described above) between the atoms. Just to be clear, let's take an example: If we had 3 atoms in the box, we might write express the potential energy as

$$PE = V_{LJ}(\text{Distance}[R1, R2]) + V_{LJ}(\text{Distance}[R1, R3]) + V_{LJ}(\text{Distance}[R2, R3]),$$

where

$$V_{LJ}(r) = 4.0[1.0/(r^{12}) - 1.0/(r^{6})].$$

Since the total energy is a conserved quantity, we expect it will stay almost constant as our simulation proceeds.

The Actual Molecular Dynamics Code

To save you some effort, you will not have to implement the entire molecular dynamics code. You should peruse the following code to see the flow of logic in a simple molecular dynamics simulation:

```
import numpy as np

# Everyone will start their gas in the same initial configuration.
# ------------------------------------------------------------------
def InitPositionCubic(Ncube, L):
    """Places Ncube^3 atoms in a cubic box; returns position vector"""
    N = Ncube**3
    position = np.zeros((N,3))
    rs = L/Ncube
```

```
    roffset = L/2 - rs/2
    n = 0
    # Note: you can rewrite this using the `itertools.product()` function
    for x in range(0, Ncube):
        for y in range(0, Ncube):
            for z in range(0, Ncube):
                if n < N:
                    position[n, 0] = rs*x - roffset
                    position[n, 1] = rs*y - roffset
                    position[n, 2] = rs*z - roffset
                n += 1
    return position

def InitVelocity(N, T0, mass=1., seed=1):
    dim = 3
    np.random.seed(seed)
    # generate N x dim array of random numbers, and shift to be [-0.5, 0.5)
    velocity = np.random.random((N,dim))-0.5
    sumV = np.sum(velocity, axis=0)/N  # get the average along the first ax
    velocity -= sumV  # subtract off sumV, so there is no net momentum
    KE = np.sum(velocity*velocity)  # calculate the total of V^2
    vscale = np.sqrt(dim*N*T0/(mass*KE))  # calculate a scaling factor
    velocity *= vscale  # rescale
    return velocity

# The simulation will require most of the functions you have already
# implemented above. If it helps you debug, feel free to copy and
# paste the code here.

# We have written the Verlet time-stepping functions for you below,
# `h` is the time step.
# ------------------------------------------------------------------

def VerletNextR(r_t, v_t, a_t, h):
    """Return new positions after one Verlet step"""
    # Note that these are vector quantities.
    # Numpy loops over the coordinates for us.
    r_t_plus_h = r_t + v_t*h + 0.5*a_t*h*h
    return r_t_plus_h

def VerletNextV(v_t,a_t,a_t_plus_h,h):
    """Return new velocities after one Verlet step"""
    # Note that these are vector quantities.
    # Numpy loops over the coordinates for us.
    v_t_plus_h = v_t + 0.5*(a_t + a_t_plus_h)*h
```

```
    return v_t_plus_h


# Main Loop.
# ----------------------------------------------------------------------

# R, V, and A are the position, velocity, and acceleration of the atoms
# respectively. nR, nV, and nA are the next positions, velocities, etc.
# There are missing pieces in the code below that you will need to fill in.
# These are marked below with comments:

def simulate(Ncube, T0, L, M, steps, h):
    """Initialize and run a simulation in a Ncube**3 box, for steps"""
    N = Ncube**3
    R = InitPositionCubic(Ncube, L)
    V = InitVelocity(N, T0, M)
    A = np.zeros((N,3))

    E = np.zeros(steps)

    for t in range(0, steps):
        E[t] = my_kinetic_energy(V, M)
        E[t] += ## calculate potential energy contribution
        F = ## calculate forces; should be a function that returns an N x 3
        A = F/M
        nR = VerletNextR(R, V, A, h)
        my_pos_in_box(nR, L)  ## from PrairieLearn HW

        nF = ## calculate forces with new positions nR
        nA = nF/M
        nV = VerletNextV(V, A, nA, h)

        # update positions:
        R, V = nR, nV
    return E

# You may adjust the gas properties here.
# ---------------------------------------

# mass
M = 48.0

# number of Particles
Ncube = 4
N = Ncube**3
```

```
# box side length
L = 4.2323167

# temperature
T0 = 0.728
```

Note: The slowest part of a molecular dynamics simulation is computing the distances and the forces. If you are interested in speeding up your code, consider the following tips.

- Before trying to speed up your code, make sure it works as is. A slow function is better than a broken function. Ideally, you should profile your code and find out which parts are slow—there is no point in spending time speeding up one function when your program spends all its time in another. In Jupyter, you can use `%lprun` (an example is here) to see how much time is spent on individual lines in a function.
- Python `for` loops can be slow. Whenever possible, allow `numpy` (or python builtin's) to manipulate vectors for you rather than looping over all elements. Often, using vectors in `numpy` results in much neater looking code as well.
- Notice that in our (straightforward) implementation of the Verlet algorithm, two evaluations of force are used. You might notice that each iteration, the first for loop computes the same acceleration that the 'next' loop computed previously. You can use this to essentially cut your computation in half.
- Calculating internal forces and potential both require a) looping over all the particles and b) computing distances. You might save computation time by writing/modifying a function that calculates both together.

Your report

1. You can vary the time step by changing the variable h just above the Verlet time-stepping functions. Using a time step of 0.01, run your simulation for at least 1000 steps (a run of 1000 steps should take about 10 minutes to complete). Plot an energy trace of the run (the total energy at each time step in the simulation) and include in your report. (As a check on your energy trace, look to see if your total energy remains approximately constant.)

2. Initially, we used a time step of 0.01, which is probably not optimal and may not even be stable! It is important to be in the habit of ensuring that your time step is neither too small (your system will never evolve from its initial configuration) nor too large (your numerical integration will be unstable and the velocities will explode). Here, our goal is to find the largest time step that gives a stable mean and keeps total energy fluctuations within 5% of the mean. This would be the largest acceptable time step, and you will generally want to work with a time step that is an order of magnitude smaller than this to mitigate the time step error.

   Make a graph of the standard deviation of the total energy versus time step size. In other words, you are trying to see how the energy fluctuates with respect to your choice of time step. For this you need to make several simulations using as many as 10 different time steps.

3. What happens if you pick a very large time step? Why?

4. From your graph, determine the largest time step one can use and still maintain energy conservation (a stable mean and fluctuations within 5% of the mean).

5. So far, we have primarily used the velocity Verlet integrator. This isn't the most obvious choice. Instead, one might imagine that we could simply Taylor expand

$$r(t+\delta t) = r(t) + r'(t)\delta t + \frac{1}{2}r''(t)\delta t^2 + \cdots \rightarrow r(t+\delta t) = r(t) + v(t)\delta t + \frac{1}{2}a(t)\delta t^2 + \cdots$$

   Try using this. Discuss what happens (look at the energy)! Why don't we use this integrator instead of Verlet? *Tip:* Write separate functions for this integrator rather than changing the Verlet functions.

6. Prove that the Verlet algorithm is time-reversal invariant. That is, show that if we use $r(t+h)$ and $r(t)$ as inputs we get $r(t-h)$ (up to round-off errors).

7. State two possible problems with finite-precision arithmetic about the Verlet time-stepping algorithm.