AtomicScaleSimulations_HW2

Questions, clarifications, comments, etc. about this homework can be posted on the course forum on Piazza.

Skeleton code: please download both particleset.py mdsim.py. You do NOT need to edit particleset.py.

mdsim.py is the main code where the functions need to be implemented. particleset.py should be in the same directory as mdsim.py so that it can be imported.

Periodic Boundary Conditions

Introduction

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 like old video games where when you come out of one side of the screen, you 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. You will implement PBC in a simple molecular dynamics code.

Now, you want to confine the particle to a box of side length L centered about the origin (i.e. the simulation box is defined from x = -L/2 to x = L/2 and y = -L/2 to y = L/2 and z = -L/2 to z = L/2. If the particle's literal position is outside the box, each coordinate should be wrapped at its respective boundary so that the position is mapped back into the box; this is the meaning of Periodic Boundary Conditions.

Implement this by writing the following function:

```
def pos_in_box(mypos, box_length):
    """ Return position mypos in simulation box using periodic boundary conditions.
The simulation box is a cube of size box_length^ndim centered around the origin
vec{0}. """

    new_pos = mypos.copy()
    ndim    = mypos.shape[0]
    # calculate new_pos

    return new_pos
# def pos_in_box
```

What is the output from the following commands?

(You may replace the "__main__" block in mdsim.py or start a new script and import the necessary objects)

```
box_length = 1.0
pset = ParticleSet(1)
pset.change_pos(0,np.array([0.501,0.499,0.501]))
print( pos_in_box(pset.pos(0),box_length) )
pset.change_pos(0,np.array([1.000, -.501, 4.501]))
print( pos_in_box(pset.pos(0),box_length) )
```

Minimum Image Convention

The other prerequisite for performing bulk simulations using PBC is the minimum image convention. Take an example where there are particles at R_1 = [-0.4,0.0,0.0] and R_2 = [0.4,0.0,0.0] in a box of L=1. Naïvely, you would think the distance between these two points would be 0.8 (if you are working in open boundary conditions this would be correct).

With periodic boundary conditions, though, remember that images of particle 1 are present at R_1' = [-0.4 + n_1, n_2, n_3] for n_1, n_2, n_3 being integers. Similarly, images of particle 2 are present at R_2' = [0.4 + n1, n2, n3]. Considering all possible pairs of particle 1 images and particle 2 images, the minimum inter-particle distance occurs for R_{12} = min |R_2' - R_1'| = |[0.6,0.0,0.0] -[0.4,0.0,0.0]| = 0.2.

The minimum image convention states that, under PBC, when we ask for the distance between particles R_1 and R_2 we should use the distance for the nearest two images (i.e. the minimum distance between any pair of images). For our box, this means that R_{12} = min |R_2' - R_1'| will never be larger than sqrt(3) L/2.

Once the concept is clear to you, implement a distance function:

```
def distance(iat, jat, pset, box_length):
    dist = 0.0
    # calculate distance with minimum image convention
    return dist
# end def distance
```

that returns the distance between the ith and jth atom in the simulation WITH the minimum image convention.

What is the output for the following commands?

```
box_length = 1.0
pset = ParticleSet(2)
pset.change_pos(0,np.array([ 0.499, 0.000, 0.000 ]) )
pset.change_pos(1,np.array([ -.499, 0.000, 0.000 ]) )
print( distance(0,1,pset,box_length) )

pset.change_pos(0,np.array([ 0.001, 0.000, 0.000 ]) )
pset.change_pos(1,np.array([ -.001, 0.000, 0.000 ]) )
print( distance(0,1,pset,box_length) )
```

Verify your output is always within the range [0, sqrt(3) L/2].

Similarly, implement a displacement function:

```
def displacement(iat, jat, pset, box_length):
    """ Return the displacement of the iat th particle relative to the jat th
particle. Unlike the distance function, here you will return a vector instead of a
scalar
"""
    posi = pset.pos(iat)
    posj = pset.pos(jat)
    disp = posi.copy()
    # calculate displacement of the iat th particle relative to the jat th particle
    #  careful with minimum image convention!
    # i.e. r_i - r_j
    return disp
# end def distance
```

The Lennard-Jones Fluid

Calculating Pair Potentials

You will now implement a molecular dynamics code to model a Lennard-Jones gas. If you chose not to use the provided skeleton code mdsim.py, you should model your code as close as you can to the skeleton code structure. In particular you need to choose the same initial conditions and use the Verlet time-stepping algorithm. In either case, you should peruse the script to see the flow of logic in a simple molecular dynamics simulation.

The Lennard-Jones potential is (is the distance at which the potential reaches 0, r_m is the distance at which the potential reaches its minimum, is the magnitude of the potential minimum)

the only force you need to calculate is from the pairwise interactions given by the Lennard-Jones potential. Recall the relation between force and potential is

For simplicity, we will work in reduced units (see p. 40 of Frenkel & Smit). This means lengths are expressed in terms of (the parameter in the Lennard-Jones potential above) and the temperature is in units of the Lennard-Jones energy parameter . Thus the reduced temperature T* = kT/. **This means we could "set" =1 and =1 in the Lennard-Jones potential**. 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 = 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 for you in "mdsim.py".

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.

To complete the molecular dynamics code provided you need to code in the Lennard-Jones interaction:

```
def internal_force(iat,pset,box_length):
    """
    We want to return the force on atom 'iat' when we are given a list of
    all atom positions. Note, pos is the position vector of the
    1st atom and pos[0][0] is the x coordinate of the 1st atom. It may
    be convenient to use the 'displacement' function above. For example,
    disp = displacement( 0, 1, pset, box_length ) would give the position
    of the 1st atom relative to the 2nd, and disp[0] would then be the x coordinate
    of this displacement. Use the Lennard-Jones pair interaction. Be sure to avoid
    computing the force of an atom on itself.
    """
    pos = pset.all_pos()  # positions of all particles
    mypos = pset.pos(iat) # position of the iat th particle
    force = np.zeros(pset.ndim())
    return force
# end def internal_force
```

Integrator

Now that we have the force, we will be able to get the acceleration, which then allows us to time evolve the particles. To numerically time evolve the particles, we have to choose a time step t. Basically, we need to determine the state of any particle at time t+t based on the state of the particle at time t.

The algorithm we are going to use is called Verlet time-stepping algorithm, which has two versions:

(1) Verlet (Without velocities)

$$r(t+\Delta t) = 2r(t) - r(t-\Delta t) + a(t)\Delta t^2$$

(2) Velocity Verlet (With velocities)

Notice that the velocity at the next moment needs the acceleration at the next moment, which is not a typo. a(t+t) can be obtained after r(t+t) is calculated.

- 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).
- State two possible problems with finite-precision arithmetic about the Verlet time-stepping algorithm.
- The Velocity Verlet integrator is algebraically equivalent to Verlet but avoids taking differences between large numbers. Prove that velocity Verlet is equivalent to Verlet. Which is preferable?

For your molecular dynamics code, you need to choose Velocity Verlet. Complete the Velocity Verlet Integrator in the molecular dynamics code provided or in your own code.

```
def verlet_next_pos(pos_t,vel_t,accel_t,dt):
    """
    We want to return position of the particle at the next moment t_plus_dt
    based on its position, velocity and acceleration at time t.
    """
    pos_t_plus_dt = pos_t.copy()
    return pos_t_plus_dt
# end def verlet_next_pos
def verlet_next_vel(vel_t,accel_t,accel_t_plus_dt):
    """
    We want to return velocity of the particle at the next moment t_plus_dt,
    based on its velocity at time t, and its acceleration at time t and t_plus_dt
    """
    vel_t_plus_dt = vel_t.copy()
    return vel_t_plus_dt
# end def verlet_next_vel
```

If you are using mdsim.py, in the "main function", you need to call velocity Verlet in the main loop

```
# molecular dynamics simulation loop
    for istep in range(nsteps):
        # calculate properties of the particles
        print(istep, compute_energy(pset))
        # update positions
        for iat in range(num_atoms):
            my_next_pos = verlet_next_pos( pset.pos(iat), pset.vel(iat),
pset.accel(iat), dt)
            new_pos = pos_in_box(my_next_pos,box_length)
   pset.change_pos(iat,new_pos)
        # end for iat
        # Q/ When should forces be updated?
        new_accel = pset.all_accel()
        # update velocities
        for iat in range(num_atoms):
            my_next_vel = verlet_next_vel( pset.vel(iat), pset.accel(iat),
new_accel[iat] )
            pset.change_vel( iat, my_next_vel )
        # end for iat
    # end for istep
```

Examining the System Energy

Now you've written enough code to get the atoms flying around in a box, with PBC. But you'll also want to get some feedback 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: Firstly, the energy of the system is the sum of the kinetic and potential energies. The kinetic energy itself is simply the sum over all atoms of m v_i^2 / 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_Lennard(Distance[R1,R2]) + V_Lennard(Distance[R1,R3]) + V_Lennard(Distance[R2,R3])" where "V_Lennard(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.

Finish the routine to monitor the energy:

```
def compute_energy(pset):
    natom = pset.nat     # number of particles
    pos = pset.all_pos() # all particle positions
    vel = pset.all_vel() # all particle velocies
    tot_kinetic   = 0.0
    tot_potential = 0.0

    # calculate energy
    tot_energy = tot_kinetic + tot_potential
    return tot_potential, tot_kinetic, tot_energy
# end def compute_energy
```

If you want to write this output to a file, you can redirect it when you run your simulation:

```
python mdsim.py > my_output.dat
```

will write all print statements to "my_output.dat".

Optional: Also useful are the following functions

(1) If you have saved the R at every step into a list called coordList (which is now of dimension num_of_step * num_of_particle * 3 dimensions), then you could use the following function to write that list to file

```
def VMDOut(coordList):
 numPtcls=len(coordList)
 outFile=open("myTrajectory.xyz","w")
 outFile.write(str(numPtcls)+"\n")
 for coord in coordList:
  for i in range(0,len(coord)):
   outFile.write(str(i)+" "+str(coord[i][0])+" "+str(coord[i][1])+"
"+str(coord[i][2])+"\n")
 outFile.close()
```

which will write your trajectories to a file that is readable by VMD, a useful visualization tool. The input parameter for this function is a nested array of these trajectories, ie. each element of the array coordList is an Nx3 array indexed by particle number and (3D) coordinate. You are encouraged to try visualizing your simulation with VMD; **instructions are here**. The .xyz file type is necessary for using VMD.

(2) If you do not want to save that gigantic list, you could append the R at each step to the same file in the following way

```
def VMDOut(R):
    outFile=open("myTrajectory.xyz","a")
    for i in range(0,len(R)):
        outFile.write(str(i)+" "+str(R[i][0])+" "+str(R[i][1])+" "+str(R[i][2])+"\n")
    outFile.close()
```

the "a" option tells "outFile" to append data to the file, not to overwrite it. **Just remember that you need to delete the file before each run, otherwise this "myTrajectory.xyz" file will keep getting longer and longer. Also, before you import it into VMD, add a line containing the total number of particles to the top of "myTrajectory.xyz".**

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). Submit your code, an energy trace of the run (the output of the ComputeEnergy function at each time step in your simulation), and a file containing the configuration of particle 0 (the data stored in position[0]) for the first 10 time steps. (As a check on your energy trace, look to see if your total energy remains approximately constant.)

Time Steps

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