AtomicScaleSimulations_HW3

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

Measuring Physical Properties

Introduction

Now that you have a functional molecular dynamics code, you want to add the capability to measure important physical and statistical properties of your system. In the previous homework, you already added code to measure the system energy. Now we will implement other observables, including several commonly studied correlation functions.

Refer to Section 4.4 of Frenkel & Smit, p.84.

In this section, we will ask you to produce plots of your new observables from a simulation of a Lennard-Jones system using the same parameters from HW2. Those parameters are

- temperature: 0.728
- box length: 4.2323167
- number of particles: 64
- time step size: 0.01
- number of time steps: 1000

Instantaneous Temperature

You should already be measuring the kinetic energy, which is directly proportional to the instantaneous temperature T, defined by the relationship

where the quantity on the left side is just the kinetic energy. N is the total number of particles. In the reduced units we've been working with, the Boltzmann constant k = 1.

Add code to output the instantaneous temperature and output it along with the kinetic energy.

This is a simple modification to the kinetic energy observable, but when we work with thermostats, it will be useful to be able to study how the energy of your system evolves.

Momentum

Measure the total momentum of your system as it evolves in time. Is momentum conserved? Do you expect it to be conserved?

Pair Correlation Function

The pair correlation function (also known as g(r) or RDF, radial distribution function) is a staple of condensed matter physics. It characterizes the spatial distribution of particles in your system and thus provides a signature of the spatial structure.

A typical way to define the pair correlation function is

where r_{ij} denotes the distance between the particles i and j, and we sum over all N(N-1)/2 pairs in the system. (r) denotes the normalization, the exact form of which you will have to determine.

Conventionally, g(r) is normalized such that g(r)=1 in a system with randomly distributed particles (i.e. there is no spatial correlation among particles).

> This means the normalization is a function of radial distance r. Think about computing **un-normalized** g(r) for an infinite system of randomly distributed particles, and convince yourself that the number of particles a distance r away will grow in proportion to the volume of a sphere of radius r. Thus, to impose g(r)=1 for a randomly distributed system, we must include in the normalization a term that compensates for this, i.e. that is inversely proportional to the volume as a function of r.

The crux of this task is understanding how to implement g(r) in a discrete, numerical setting. That is, you will not literally perform the integral and you will not end up with g(r) as a continuous function of r. Instead, you want to generate a histogram of g(r). The idea is to look at the distance between each pair of particles, r_{ij}, at a given MD time step. You should declare an array with each element initialized to 0, and you need to define a distance dr which is your grid resolution. Each element i of the array represents the range of distances from i dr to (i+1) dr. Then for each pair distance r_{ij} , you should "bin" that distance by identifying which array element i contains this value of r_{ij} and then increasing the value of that element by one.

See p.86 of Frenkel & Smit for an implementation of this function.

Implement the calculation of g(r) in your code and submit a plot of g(r) for the Lennard-Jones simulation conditions described above. Be sure to normalize your function correctly!

If your system takes time to equilibrate from its initial conditions (ie. if you see a transient in your energy trace), you should not collect g(r) data until your system is well equilibrated.

Structure Factor

Next, implement a calculation of the structure factor:

where

and the summation is over all particle position vectors.

S(k) can be cast as a Fourier transform of the pair correlation function g(r), and ρ_k is the spatial Fourier transform of the number density. In a periodic box, we may only use wavevectors k that are commensurate with the periodicity of the box, i.e.

We will implement this calculation by writing three functions. To begin with, write a function

```
def legal_kvecs(maxk):
 kvecs = []
 # calculate a list of legal k vectors
 return np.array(kvecs)
# end def legal_kvecs
```

that computes a list of the legal k-vectors. "maxk" is an integer specifying the largest value of (n_x, n_y, n_z) that is allowed. The larger k-vectors you use, the smaller length scales you can probe. However, the number of k-vectors you sample scales as "maxk^3", so you should start by choosing "maxk=5".

Next write a function to compute the Fourier transform of the density for a given wavevector, _k

You may use fast fourier transform (np.fft), but be sure to implement this by hand first to check that you know how to use np.fft correctly.

```
def rhok(kvec, pset):
    value = 0.0
 #computes \sum_j \exp(i * k \dot r_j)
 return value
# end def
```

Finally, write a function

```
def Sk(kvecs, pset):
 """ computes structure factor for all k vectors in kList
  and returns a list of them """
 return sk_list
# end def Sk
```

Now, you should be able to plot your results in the following way:

```
kmags = [np.linalg.norm(kvec) for kvec in kvecs]
plt.plot(kmags,sk_list)
```

Implement the calculation of S(k) in your code and submit a plot of S(k) from a run using the Lennard-Jones system parameters described above. Be sure to normalize your function correctly! Describe qualitatively how S(k) and g(r) should look for a liquid and a solid.

Velocity-Velocity Correlation and Diffusion Constant

Add an observable to your code that measures the velocity-velocity correlation function:

You should compute this expectation value by averaging over each particle. You will have to store

previous velocities, where t_max is the largest value of t for which you want to compute this correlation.

Implement a calculation of this quantity as a function of time \(t\); be sure to normalize your function.

Add an observable to your code that calculates the diffusion constant, which is simply an integral over the velocity-velocity autocorrelation function,

Implement a diffusion constant observable. Using the simulation parameters described above, produce a plot of the velocity-velocity correlation as a function of time. Also report your value for the diffusion constant.

Canonical Ensemble and Thermostatting

So far, we have performed molecular dynamics in the micro-canonical ensemble, i.e. at constant energy. We are now going to modify our molecular dynamics code to work in the canonical ensemble, ie. constant temperature. There are a number of ways to modify the code to simulate a canonical ensemble; generally this involves coupling the system to a heat bath which allows it to sample different energy states. Some methods are simpler than others and some methods are more accurate then others.

We will start by implementing the Andersen thermostat, which is the most straightforward algorithm to implement. This method ensures that you will get the correct properties for static equilibrium observables (i.e. pair correlation functions, energy, etc.) but the dynamics are artificial and you should not expect your simulation to provide reliable information about dynamical properties of the system (i.e. diffusion constant).

Andersen Thermostat

Algorithm:

1. Start with an initial set of configurations and momenta and integrate for t (your code already does this)
2. Loop over all particles and for each one, select it to undergo a collision with probability t where t is the time step and is a parameter defining the coupling strength to the bath (you should choose the value of so you get a collision probability around 1%).
3. For each particle that is selected to undergo a collision, its new velocity is drawn from the Maxwell-Boltzmann distribution (i.e. select a new velocity for each component from a Gaussian distribution with =sqrt(T/M) and mean 0.

You may want to refer to the documentation on Python's "random" module to see how to generate random numbers drawn from a Gaussian distribution.

(You can implement this algorithm by adding about 5 lines of code to your existing MD code).

You can also refer to Frenkel & Smit, p.140 for a discussion and implementation.

Implement this new technique in your code. To check that it is working, re-run the Lennard-Jones simulation described above but with your thermostat, and produce a plot of the instantaneous temperature as a function of simulation time.

 Now, increase the temperature to T=1.0, but still pass the value 0.728 as an argument to the "InitVelocity" function. Re-run and produce a plot of temperature vs. time for this simulation.

A limitation of the Andersen algorithm is that it uses randomized velocities, which corrupt the dynamics of the system. Static equilibrium properties are still accurate, but you will no longer be simulating the physical dynamics of your system.

To see this quantitatively, adjust the value of so that your collision probability is around 50%. Produce a plot of the velocity-velocity correlation

function with thermostatting on and report your calculation of the diffusion constant. Compare these results with your data from the microcanonical simulation (above). Specifically, you should discuss what differences you expect to see in these observables and whether or not you do, in fact, see them.

Nosé–Hoover thermostat (optional)

A more effective technique is to use Nosé–Hoover thermostats. In this method, we couple our system to a new fictitious particle that acts like a friction term on the system. Concretely, this causes us to change the equations of motion.

> Unless you have written your Anderson thermostat in a modular way, you may want to implement this thermostat in a **separate copy** of your code.

1. Start changing your code by adding a friction term of the form bv to your acceleration. In a little bit, we will dynamically change b as your molecular dynamics progresses. First, though, let's test that if you set b to some positive value and run your simulation the particles slow down and eventually halt.
2. Now, we wish to change b as the simulation evolves. We will let

$$\dot{b} = \frac{1}{Q} \left[\sum_{i=1}^{N} \frac{p^2}{m} - \frac{g}{\beta} \right]$$

where Q is a mass scaling parameter that you have to choose, =1/kT and g=3N+1.

Calculating phase transitions

In this problem, you will be using your molecular dynamics code to estimate the transition temperature at which argon turns from a solid to a liquid.

To begin with, let's specify the relevant parameters for our argon simulation, which again uses the Lennard-Jones potential in reduced units (ie. ==1):

- mass: 48.0
- density: 1.0
- timestep: 0.032
- particle number: 64
- using Andersen thermostat for canonical ensemble

Make sure you initialize the particleset with the correct **mass** and pass **temperature** to the velocity initialization routine.

```
mass = 48
num_atoms = 64
mytemp = 2.0 # for example
box_length = # calculate from density

pset = ParticleSet(num_atoms,mass)
pset.init_pos_cubic(box_length)
pset.init_vel(mytemp)
```

> To this point, we have worked with a particle mass of 1, and you may not have used this value explicitly in your code. Now, it is crucial that the mass of 48 appears in the correct equations, namely the relationship between force and acceleration, in the Andersen thermostat algorithm, and in computing the kinetic energy.

Choose 5 temperatures in the range between 0 and 4 and run MD simulations at those temperatures.

From this, you should be able to roughly identify the location of the melting/freezing transition.

Now, try to refine your estimate by choosing three more temperature points in this vicinity.

You should study all of the observables you have implemented to determine the phase of your system at a given temperature.

Examining the structure factor is likely the most useful way to determine the phase of your system.

Submit your estimate of the transition temperature along with data that support this claim. Minimally, you should include graphs of the pair correlation function and structure factor for temperatures above and below your estimated transition temperature.

Finite Size Effects (optional)

> Nothing in this section is required. Instead it is for your edification only.

We are interested in calculating the properties of systems at the thermodynamic limit (i.e. the number of particles goes to infinity). Since we are currently only using a system of 64 particles, we will see some effects related to this finite size. One of these effects is that your estimate of the transition temperature will be subject to systematic errors.
One source of error is hysteresis (ie. if your system starts in a solid, it is more likely to stay solid to a higher temperature then if it started as a liquid). We would like to see how much this problem has contaminated our result. To do this one would ...

1. Choose a temperature near but below the transition temperature.
2. Initialize your system as a liquid instead of a solid. For example, you could take your final coordinates from a simulation above the phase transition, and use them as initial conditions for your simulation below the transition temperature.
3. Find the temperature of the phase transition again.

You may note that the temperature at which you see a phase transition is now different. Hysteresis is often described as the effect of the system having "memory" of its initial conditions, meaning that the phase transition will occur at a different temperature depending on how the system was initialized. This means that you aren't able to decisively locate the transition point. One solution to this problem is to simulate larger systems.