Exercise 3

Due: Tuesday, 10/29/19

Objective: To learn how to write & compile Fortran libraries for Python, to perform a basic molecular dynamics simulation, and to compute thermodynamic and kinetic property averages from it.

A simple model of a linear polymer is the Lennard-Jones chain. An atomic system of such chains interacts according to the following (dimensionless) potential energy function:

$$U^* = \sum_{i<j\ ,ij\ \text{not bonded}} 4\left(r_{ij}^{-12} - r_{ij}^{-6}\right) + \sum_{i<j\ ,ij\ \text{bonded}} \frac{k}{2}\left(r_{ij} - r_0\right)^2$$

Here, the i and j are indices over *atoms* in the system. Make the following definitions:

M – number of monomers (LJ atoms) per polymer

N_{poly} – number of polymers in the simulation cell

$N = MN_{\text{poly}}$ – number of total atoms in the system

Notice that the potential energy function involves two parameters: k is the force constant for harmonic bonds between adjacent atoms in the same polymer and r_0 is the equilibrium bond distance. We will take $k = 3000$ and $r_0 = 1$, which are standard in the literature.

Lennard-Jones chains have been studied extensively as models of polyatomic molecules and very short polymers. These systems have suggested scaling laws for various properties as a function of polymer length. In particular, Reis and coworkers [Reis et al., Fluid Phase Equilibria 221, 25 (2004)] have evaluated the self-diffusion coefficient for systems of $M = 2,4,8,16$.

In this assignment, you will develop a molecular dynamics simulation of Lennard-Jones chains and use this to compute the self-diffusion coefficient for several polymer lengths and temperatures. To assist you, some of the code has already been written, but you are free also to develop your code from scratch should you choose to do so. The template files can be found on the course website as ex3lib.f90 and ex3.py.

Part a

Edit the ex3lib.f90 file to add a function `CalcEnergyForces` that computes the total potential energy and the force on each atom for a given configuration. Then add a second function `CalcEnergy` (copying and editing the code for the first) that only computes the total potential energy. The reason you will add two functions is that you will energy-minimize your initial configurations before performing molecular dynamics and the line search does not require the forces, which we want to avoid computing for efficiency reasons.

In the pairwise interaction loop, you can determine whether or not two atoms i and j are bound by the following kind of code:

```
do i = 0, NAtom - 1
    do j = i + 1, NAtom - 1
        if (j == i + 1 .and. mod(j, M) > 0) then
            !this is a bonded interaction;
            !use the harmonic bond potential here
        else
            !this is a nonbonded interaction;
            !use the Lennard-Jones potential here
        endif
    enddo
enddo
```

The code `mod(j, M)` will return 0 every time the i and j=i+1 particle are in different molecules. This is because when the modulo of j with M is zero, j is a multiple of M and it therefore is the first atom of a polymer while i is the last of the previous.

Your simulation will take place under periodic boundary conditions in a cubic box of side length L. Therefore, you will need to compute the minimum image distance. In Fortran,

```
rij = rij - L * dnint(rij / L)
```

where `rij` is the vector (a length-3 array) of position differences.

For the nonbonded interactions, you should cut and shift the Lennard-Jones potential at a distance of r_c. Do not apply a tail correction.

Keep in mind several efficiency considerations. You do not want to compute the square root of the distance unless you have to (e.g., the Lennard-Jones potential doesn't require it, but the harmonic potential does). Before the pairwise loop, you should precompute any constants, such as the Lennard-Jones shift value.

Compile your Fortran file into a Python library using f2py. You may want to compile it first using gfortran to detect any errors / bugs. This is covered in the handout on compiling routines using f2py.

Part b

Edit the Python file ex2.py to develop a molecular dynamics code. Several functions have been written for you already and are described in the docstrings and comments.

Your code for a molecular dynamics run should do the following:

- Initially place atoms on a cubic lattice. You can use the existing function `InitPositions` in the template file.
- Energy-minimize the initial configuration using the conjugate-gradient method. This is for numerical stability when starting the MD integration .
- Use the velocity Verlet integrator to perform a molecular dynamics run. The atomic velocities should be rescaled every **`RescaleFreq`** integration steps to achieve a target temperature T. At any step number i, you can test whether or not you should rescale the velocities using the modulo operator (e.g., **`i % RescaleFreq == 0`**).

Use the following settings:

- $N = 240$
- $\rho = N/V = 0.8$ so that $L = (N/\rho)^{1/3}$
- $\Delta t = 0.001$
- $T = 1.0$
- $r_c = 2.5$

Perform simulations for $M = 2, 4, 6, 8, 12, 16$, where $N_{\text{poly}} = N/M = 240/M$. On a single graph, add a series for each that gives the potential energy as a function of time for $t = 0$ to $t = 2$. (2,000 integration steps).

Part c

Consider the $M = 8$ simulation run. After $t = 2$, turn off velocity rescaling and monitor the total energy, which should be constant with time. Compute the fractional fluctuation in total energy $\sigma_E/\langle E \rangle$ as a function of the time step $\Delta t = 0.0001,\ 0.0002,\ 0.0004,\ 0.0008,\ 0.0016,\ 0.0032,\ 0.0064,\ 0.0128$. Place your results on a second graph that is a log-log plot of $\sigma_E/\langle E \rangle$ versus Δt.

Part d

Modify your molecular dynamics code to perform a series of steps that will allow you to compute the self-diffusion coefficient as a function of temperature and chain length:

- First perform equilibration for **`NStepsEquil1`** integration steps using velocity rescaling to the target temperature T every **`RescaleFreq`** steps.
- Perform a second equilibration for **`NStepsEquil2`** integration steps using velocity rescaling every **`RescaleFreq`** steps. At the end of this period, the average total energy for the **`NStepsEquil2`** steps should be computed. The velocities should then be rescaled such that the current kinetic energy plus the *current* (instantaneous, not average) potential energy equals the target average total that you computed.
- Copy the positions of the particles into a reference array for computing the mean-squared displacement, e.g., **`Pos0 = Pos.copy()`**.
- Perform a production run for **`NStepsProd`** integration steps without velocity rescaling, that is, for constant NVE dynamics. During the run, periodically record the time and the mean-squared displacement of the atoms from their initial positions.

Use the settings

- **`NStepsEquil1`** = 10,000
- **`NStepsEquil2`** = 10,000
- **`NStepsProd`** = 100,000

Perform runs for $M = 2, 4, 6, 8, 12, 16$. Create a third graph that gives the mean-squared displacement for each as a function of time for $t_{\mathrm{tot}} = 100$. Estimate the diffusion coefficient from the slopes and plot these on a fourth graph as a function of chain length. Does the diffusion coefficient appear to obey any obvious scaling law, $D \sim M^{\nu}$?

You may want to benchmark your results with those of Reis et al.

Part e

Make a short movie of a production run for the case $M = 16$. Color one chain differently so that you can visually follow its evolution in time (HINT: give this chain unique atom names using the AtomNames option of atomwrite.py). Print out a screenshot of your movie.

Part f – ADVANCED TRACK

For $M = 4, 8, 16$, compute the self-diffusion coefficient for $T = 1.0, 1.5, 2.0, 2.5, 3.0$. Make a fifth plot of $\ln(D/T)$ as a function of $1/T$ with each M as a separate series. If the diffusion constant follows an Arhennius relationship, to first order the slope of this line should be linear:

$$D \sim T \exp\left[\frac{-E_a}{k_B T}\right]$$

How does the activation energy compare for the different chain lengths? Note: you may want to automate the computation of the self-diffusion coefficient in your code. NumPy comes with a function for performing least-squares regression:

```
Slope, Intercept = np.polyfit(xvals, yvals, 1)
```

Part g – ADVANCED TRACK

Are system size effects significant? Consider the case $M = 16$. How does the computed diffusivity vary as the total number of atoms (and hence polymers) increases? You may want to make a plot of the calculated diffusivity as a function of $1/L$ where L is the box dimension, such that extrapolation to $1/L = 0$ estimates the infinite system size limit.

What to turn in

For the assignment, submit a short, typed summary of your results, clearly indicating the problem components. In addition, print a copy of your Python code for reference and attach it to the back of these.

Template overview

The following is a list of functions already implemented for you in the ex3.py template:

```
Help on module ex3:

NAME
    ex3 - #Exercise 3 template for CHE210D

FILE
    ex3.py

FUNCTIONS
    LineSearch(Pos, Dir, dx, EFracTol)
        Performs a line search along direction Dir.
        Input:
            Pos: starting positions, (N,3) array
            Dir: (N,3) array of gradient direction
            dx: initial step amount
            EFracTol: fractional energy tolerance
```

```
        Output:
            PEnergy: value of potential energy at minimum along Dir
            Pos: minimum energy (N,3) position array along Dir

    ConjugateGradient(Pos, dx, EFracTolLS, EFracTolCG)
        Performs a conjugate gradient search.
        Input:
            Pos: starting positions, (N,3) array
            dx: initial step amount
            EFracTolLS: fractional energy tolerance for line search
            EFracTolCG: fractional energy tolerance for conjugate gradient
        Output:
            PEnergy: value of potential energy at minimum
            Pos: minimum energy (N,3) position array

    InitAccel(Pos)
        Returns the initial acceleration array.
        Input:
            Pos: (N,3) array of atomic positions
        Output:
            Accel: (N,3) array of acceleration vectors

    InitPositions(N, L)
        Returns an array of initial positions of each atom,
        placed on a cubic lattice for convenience.
        Input:
            N: number of atoms
            L: box length
        Output:
            Pos: (N,3) array of positions

    InitVelocities(N, T)
        Returns an initial random velocity set.
        Input:
            N: number of atoms
            T: target temperature
        Output:
            Vel: (N,3) array of atomic velocities

    RescaleVelocities(Vel, T)
        Rescales velocities in the system to the target temperature.
        Input:
            Vel: (N,3) array of atomic velocities
            T: target temperature
        Output:
            Vel: same as above

    InstTemp(Vel)
        Returns the instantaneous temperature.
        Input:
            Vel: (N,3) array of atomic velocities
        Output:
            Tinst: float
```