CHE384T: Molecular Dynamics

1 PS4: Molecular Dynamics at the surface of a solid and liquid

1.1 Context and Motivations

1.1.1 Introduction

Molecular dynamics is often used to study the thermodynamic and time-dependent structural properties of solids and liquids. It has been used to study various types of materials, including noble gases, ionic materials, covalent materials, and metals. Although simple in form, the interatomic potentials used in this class have been used to study various materials as well. The purpose of this assignment is to introduce you to methods used in molecular dynamics (MD) and to gain some experience in it use.

Perhaps the most well-studied material with atomistic simulations is "Lennard-Jonesium," a material descrbied with a Lennard-Jones interatomic potential. We will perform molecular dynamics simulations with such a system. While we use the Lennard-Jones potential here, it would be straightforward to use more sophisticated potentials such as the embedded-atom model or a bond-order potential.

This problem set will involve several incremental changes to the given function files. You may find it useful to work in terms of python scripts or an IDE for the coding and preparing the report in a Jupyter notebook. Note that you may import user-made python scripts as modules as long as that script is listed in your PYTHON_PATH.

You may also find useful making a visual representation showing the flow of routine calls used in the code, particularly as you modify small portions of the code.

1.1.2 The Lennard-Jones potential

The Lennard-Jones potential is defined as

$$\phi_{LJ}(r) = 4\varepsilon\Big[\Big(\frac{\sigma}{r}\Big)^{12} - \Big(\frac{\sigma}{r}\Big)^{6}\Big].$$

We will use reduced units, i.e., $E^* = E/\varepsilon$ for energy, $U^* = U/\varepsilon$ for potential energy, $r^* = r/\sigma$ for distance, $V^* = V/\sigma^3$ for volume, and $T^* = k_B T/\varepsilon$, where k_B is the Boltzmann constant. While kinetic energy has units of energy, it is proportional to mv^2, which has units of mass(distance/time)^2. For consistency with the remaining reduced units, we will defined a reducted time as $t^* = t/t_o$ where $t_o = \sqrt{m\sigma^2\varepsilon}$.

The Lennard-Jones potential in reduced units then becomes (dropping the *)

$$\phi_{LJ}(r) = 4\Big[\Big(\frac{1}{r}\Big)^{12} - \Big(\frac{1}{r}\Big)^{6}\Big].$$

Recall in the last few problem sets we used cutoff distances. Using cutoffs introduce discontinuities in the potential and force at the cutoff distance r_c. We may eliminate this discontinuity of the potential by *shifting* the energy by the value of the true potential at the cutoff so that it goes to zero at the cutoff distance, i.e.,

$\phi_{\text{shift}}(r) = \phi_{LJ}(r) - \phi_{LJ}(r_c)$.

We then add a correction to the energy at the end of the calculation. This does not eliminate the discontinuity in the force, however, which can lead to issues in energy convergence. For these exercises, *we will ignore both such discontinuities.* A serious production run calculation wuold require additional methods, e.g., introducing an interpolating function in the force and potential so that both go smoothly to 0 at r_c.

1.1.3 Forces and Pressure

The Lennard-Jones potential is what is known as a central force potential. The force on a particle i for the Lennard-Jones potential is given by

$\vec{F}_i = \sum_{j\neq i} \vec{f}_i = \sum_{j\neq i} \Big(-\frac{1}{r_{ij}} \frac{d\phi}{dr_i} \Big) \vec{r}_{ij} = \sum_{j\neq i} \frac{24}{r_{ij}^2} \Big(2 \Big(\frac{1}{r_{ij}} \Big)^{12} - \Big(\frac{1}{r_{ij}} \Big)^{6} \Big) \vec{r}_{ij},$

where the sum is over all neighbors if particle i is within the cutoff distance. Note that again we have dropped the *.

For a central force potential, the pressure may be calculated by

$P = \frac{N}{V} k_B T + \frac{1}{3V} \Big\langle \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \frac{24}{r_{ij}^2} \Big(2 \Big(\frac{1}{r_{ij}} \Big)^{12} - \Big(\frac{1}{r_{ij}} \Big)^{6} \Big) \Big\rangle,$

where the angle brackets $\langle\ \rangle$ indicates a thermodynamic average.

1.1.4 Solving the equations of motion with the Verlet algorithm

We will use the Verlet algorithm to propogate the equations of motion. While many choices are possible, the Verlet algorithm is simple to implement and for the purposes of these exercises will provide satisfactory results.

According to the Verlet algorithm, the equations to advance the position of the ith particle from time t to $t + \delta t$ is

$\vec{r}_i(t + \delta t) = 2\vec{r}_i(t) - \vec{r}_i(t - \delta t) + \vec{a}_i(t)\delta t^2,$

where the acceleration $\vec{a}_i(t) = \vec{F}_i(t)/m_i$ depends on the forces F_i as computed above and m_i the mass of the particle i. The velocities are not computed or used directly in the Verlet algorithm but are needed to compute the kinetic energy, temperature, and other thermodynamic quantities. We use the central difference formulat to estimate the velocities,

$\vec{v}_i(t) = \frac{\vec{r}_i(t+\delta t) - \vec{r}_i(t-\delta t)}{2\delta t}$.

The time step δt is chosen to ensure that the total energy $E = U + K$ is conserved.

The positions at times t and $t - \delta t$ are needed to find the positions at $t + \delta t$. However, one time we do no have such information is at $t = 0$. The initialization of positions and velocities are discussed in the next section. We require the positions for $\vec{r}_i(-\delta t)$ to obtain $\vec{r}_i(+\delta t)$, which may be determined using

$\vec{r}_i(-\delta t) = \vec{r}_i(0) - \vec{v}_i(0)\delta t + \frac{1}{2}\vec{a}_i(0)\delta t^2$

where the forces and accelerations are computed using the initial positions $\vec{r}_i(0)$.

1.1.5 Initialization of positions and velocities

We require the initial positions and velocities. The atomic positions for N atoms in the simulation are chosen as needed for the problem of interest. For simulating condensed systems (i.e., solids and liquids), we usually start from a solid structure (e.g., FCC or BCC structure).

So far we have used scaled coordinates (i.e., fractional coordinates scaled to the size of the simulation cell). Additional care will need to be used when dealing with molecular dynamics simulations as discussed below.

The velocities may be chosen in a number of ways. For this exercise, we choose velocities sampled from the Maxwell-Boltzmann distribution, as discussed in the text. We first choose an initial temperature T_{init} and then select each component of the velocity $\vec{v}_=(v_x, v_y, v_z)$ for each atom from a distribution of the form

$\rho(v_x) = \frac{1}{\sigma\sqrt{2\pi}} \exp(-v_x^2/2\sigma^2)$,

where $\sigma = \sqrt{k_B T/m}$.

We can generate a set of random numbers from a normal (Gaussian) distribution with a zero mean and unit standard deviation $\sigma = 1$ by choosing

$\tau = (-\ln \mathscr{R}_1)^{1/2} \cos(\pi \mathscr{R}_2)$,

where $\mathscr{R}_1$ and $\mathscr{R}2$ are real random numbers generated between (0,1). We assign velocities based on the random numbers generated. The net linear momentum is conserved, which we may compute from our initial set of velocities. By substracting the average from each velocity in the initial set, we ensure that the total linear momentum is zero at the beginning of the calculation and thus eliminate the possibility that the entire set of atoms drifts during the simulation (which makes analysis easier). Based on these velocities, we calculate the kinetic energy

$K = \frac{1}{2}\sum_{i=1}^{N} m_i v_i^2$

Sincer $K = 3Nk_BT/2$, the kinetic energy for the selected initial temperature is $K_{init} = 3Nk_BT_{init}/2$. By rescaling each velocity by the factor $\sqrt{K_{init}/K}$, we may ensure that the kinetic energy from our input velocities equals that associated with the initial, prescribed temperature. This is the simplest strategy for enforcing a particular temperature. Several other strategies exist, with varying degrees of complexity and extents to which they satisfy thermodynamic relations.

These velocities to the time rate of change of the *real* coordinates of the atoms, not the scaled coordinates used in the simulation. However, the molecular dynamics code we have below, the atom positions are in scaled coordinates. Thus, we need to be careful to make sure that the velocities and forces are scaled appropriately.

A code to create the initial distribution of velocities may look like

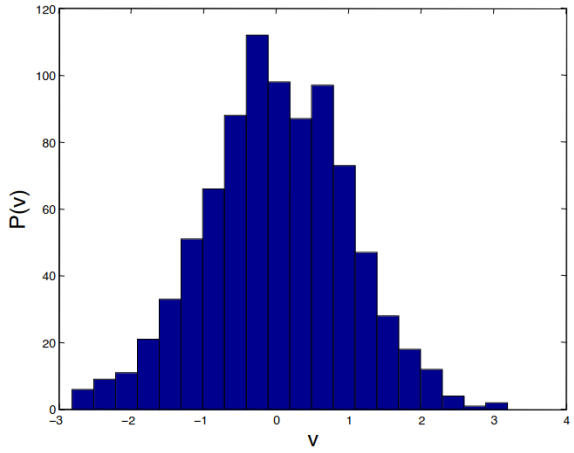
[1]:
```
%pycat ./code_exercises/init_vel_3D.py
```

```
import numpy as np
```

```
# initialize velocities and momentums using Maxwell-Boltzmann distribution
# rescale velocities to prescribed temperature Tin

def init_vel_3D(n, Tin):
    """
    Pick velocities from Maxwell-Boltzmann distribution
    for any temperature we want.
    Then we will calculate the kinetic energy and thus
    the temperature of these atoms and then we will
    rescale the velocities to the correct temperature

    Input:
       n (integer): number of steps in trajectory
       Tin (float): initial temperature
    Output:
       vx, vy, vz (float): initial velocities
       px, py, pz (float): initial momentums
    """
    k = 0
    px = 0
    py = 0
    pz = 0

    vx = np.zeros(n)
    vy = np.zeros(n)
    vz = np.zeros(n)

    for i in range(n):
        vx[i] = np.sqrt(-2 * np.log(np.random.rand())) * np.cos(2 * np.pi * np.
↪random.rand())
        vy[i] = np.sqrt(-2 * np.log(np.random.rand())) * np.cos(2 * np.pi * np.
↪random.rand())
        vz[i] = np.sqrt(-2 * np.log(np.random.rand())) * np.cos(2 * np.pi * np.
↪random.rand())

        px += vx[i]
        py += vy[i]
        pz += vz[i]

    # Find average momentum per atom
    px /= n
    py /= n
    pz /= n

    # Set net momentum to zero and calculate K
    for i in range(n):
        vx[i] -= px
```

```
        vy[i] -= py
        vz[i] -= pz

        k += vx[i]**2 + vy[i]**2 + vz[i]**2

    k *= 0.5

    # Kinetic energy of desired temperature (Tin)
    kin = 1.5 * n * Tin

    # Rescale velocities
    sc = np.sqrt(kin / k)
    for i in range(n):
        vx[i] *= sc
        vy[i] *= sc
        vz[i] *= sc

    return vx, vy, vz, px, py, pz
```

We plot below a histrogram distribution of the velocities sampled in an example calculation. Note that aside from fluctuations from being a finite sample, the velocities follow a normal distribution centered around 0.



1.1.6 Boundary conditions

The next ingredient needed is boundary conditions. Consider a typical calculation that uses periodic boundary conditions, which would be used to simulate a solid or fluid. Other boundary conditions may be used in different contexts, e.g., for one of the Bravais lattices, free surfaces in surface structures.

In these exercises we consider bulk simulations and assume the *minimum image convention*. Implementation of the minimum image convention is discussed in the text and in the prior problem set. We emphasize that the minimum image convention was popular in the early days of computer simulations when systems had limited memory and were less powerful than today's machines. Since

we use small systems here, the minimum image approximation is appropriate. For research purposes that often involve supercomputers, a different strategy for lattice sums would likely be employed.

1.2 Implementation of force and energy calculations

At each time step, we have a new set of atomic positions $\vec{r}$. To calculate the positions at the next time step requires the accelerations of each atom, and thus the forces on that atom from the other particles in the system. We also would like to calculate the potential energy, kinetic energy, pressure, and other thermodynamic quantities of interest.

We may write the potential energy as

$U = \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \phi_{ij}(r_{ij})$.

where terms for $i = j$ are omitted. Equivalently, to avoid double counting interations and to also cut the computation time in half, we can also write the potential energy as

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

To be computationally efficient, we will calculate U and the force with the same lattice sum strategy. The second form of U is more computationally efficient than the first form. However, for forces, we need $\vec{f}_{ij}$ the force exerted by atom j on atom i and the force exerted by atom i on atom j, $\vec{f}_{ij} = -\vec{f}_{ji}$. We need to accumulate both terms as we sum over the interactions, of which one method is demonstrated in the following code where we invoke the minimum image convention and a distance cutoff. We additionally accumulate values for computing the pressure.

[3]:
```
%pycat ./code_exercises/forces_LJ.py
```

```
import numpy as np

def forces_LJ(a, n, x, y, z):
    """
    Simple lattice sum for force with cutoffs and
    minimum image convention

    We calculate force (fx, fy, fz), energy (u), and
    part of the pressure (w).

    """
    fx = np.zeros(n)
    fy = np.zeros(n)
    fz = np.zeros(n)
    u = 0
    w = 0

    for i in range(n - 1):  # Note limits
        ftx = 0
        fty = 0
        ftz = 0
        for j in range(i + 1, n):  # Note limits
            # Minimum image convention
```

```
            dx = x[j] - x[i]
            dy = y[j] - y[i]
            dz = z[j] - z[i]
            dx -= round(dx)
            dy -= round(dy)
            dz -= round(dz)
            dist = a * np.sqrt(dx**2 + dy**2 + dz**2)
            if dist <= rc:
                dphi = (2 / dist**12 - 1 / dist**6)
                ffx = dphi * a * dx / dist**2
                ffy = dphi * a * dy / dist**2
                ffz = dphi * a * dz / dist**2
                ftx += ffx
                fty += ffy
                ftz += ffz
                phi = (1 / dist**12 - 1 / dist**6)
                u += phi
                w += dphi

                # Add -f to sum of force on j
                fx[j] -= ffx
                fy[j] -= ffy
                fz[j] -= ffz

        # Sum up force on i (fi)
        fx[i] += ftx
        fy[i] += fty
        fz[i] += ftz

    # Need to multiply LJ by 4 and force and pressure by 24
    # Also need to correct sign in f
    u *= 4
    w *= 24
    fx *= -24
    fy *= -24
    fz *= -24

    return u, w, fx, fy, fz
```

Note that the data structure is somewhat different in this version of the code In earlier codes, we defined a vector `s(i,alpha)`, which returned the components x of `s` for `alpha = 0`, y components for `alpha = 1`, and z components for `alpha = 2`. We use the variables `x(j), y(j), z(j)` for each atom to make the code easier to follow.

1.3 Implementation of Verlet algorithm

By choice, we write all atomic positions in fractional coordinates. Here, we show an implementation of the Verlet algorithm for the time propagation of an MD simulation involving a *cubic* simulation cell with lattice parameter a. Note where all the places a appears.

The relevant parameters are: - `nc`: number of fcc unit cells, as described in our previous set of exercises - `density`: volumetric density in reduced units - `tin`: input temperature in reduced units - `nsteps`: total number of time steps in a calculation - `dt`: time step in reduced units

The basic code is shown below.

[4]:
```
%pycat ./code_exercises/MD_LJ.py
```

```
import numpy as np

def MDLJ(density, tin, nsteps, dt):
    """
    Initialize positions and velocities.
    Calculate some useful quantities.
    Calculate initial energy and forces.
    Now start the time stepping with the Verlet algorithm.
    """
    # Initialize positions and velocities
    n, x, y, z, vx, vy, vz = initLJMD(nc, tin)

    # Calculate some useful quantities
    vol = n / density
    a = vol**(1/3)
    rc = a / 2

    # Calculate initial energy and forces
    u, w, fx, fy, fz = forces_LJ(a, n, rc, s)

    # Initialize variables
    xold = np.zeros(n)
    yold = np.zeros(n)
    zold = np.zeros(n)
    xnew = np.zeros(n)
    ynew = np.zeros(n)
    znew = np.zeros(n)

    # Time series arrays
    un = np.zeros(nsteps)
    kn = np.zeros(nsteps)
    en = np.zeros(nsteps)
    tn = np.zeros(nsteps)
    pn = np.zeros(nsteps)
```

```
    # First find the positions at t-dt
    for i in range(n):
        xold[i] = x[i] - vx[i]*dt/a + 0.5*fx[i]*dt**2/a
        yold[i] = y[i] - vy[i]*dt/a + 0.5*fy[i]*dt**2/a
        zold[i] = z[i] - vz[i]*dt/a + 0.5*fz[i]*dt**2/a

    # Start the time steps
    for j in range(nsteps):
        k = 0
        # Find positions for time t + dt and velocities for time t
        for i in range(n):
            xnew[i] = 2*x[i] - xold[i] + fx[i]*dt**2/a
            ynew[i] = 2*y[i] - yold[i] + fy[i]*dt**2/a
            znew[i] = 2*z[i] - zold[i] + fz[i]*dt**2/a
            vx[i] = a*(xnew[i] - xold[i]) / (2*dt)
            vy[i] = a*(ynew[i] - yold[i]) / (2*dt)
            vz[i] = a*(znew[i] - zold[i]) / (2*dt)
            k += vx[i]**2 + vy[i]**2 + vz[i]**2

        k *= 0.5
        temp = 2*k / (3*n)

        # Create time series of values
        e = k + u
        un[j] = u / n
        kn[j] = k / n
        en[j] = e / n
        tn[j] = temp
        pn[j] = density*temp + w / (3*vol)

        # Reset positions for next time step
        for i in range(n):
            xold[i], yold[i], zold[i] = x[i], y[i], z[i]
            x[i], y[i], z[i] = xnew[i], ynew[i], znew[i]

        # Calculate force and energy at new positions for next cycle
        u, w, fx, fy, fz = forces_LJ(a, n, rc, x, y, z)

    return un, kn, en, tn, pn
```

1.3.1 Determine if the MD simulation is properly working

With this bare bones implementation, we need to see if our MD simulation works appropriately. Usually, one has to consider if: 1) the code is written correctly or 2) if the user is using the code correctly. We can answer both to a certain degree with test against physically known principles.

Here, we use the most fundamental of physically known principles, the conservation of total energy

over time, i.e., $E = K + U$. We will expect some fluctuations in the energy over time (and in certain cases these fluctuations contain important thermodynamic information). However, the fluctuations should be relatively small and the time-average of the total energy $\langle E \rangle$ should remain constant and not drift with time. A rule of thumb is

$\frac{\max(E) - \min(E)}{\langle E \rangle} \sim 10^{-4}$

where $\max(E) - \min(E)$ is the range of values of E. If $\langle E \rangle$ is small, then you must correpsondingly adjust this criteria.

If your total energies begin drifting with time, there are a few places to look first. 1) There is an error in code. You will need to debug the code. 2) Your timestep dt is too large, which leads to inaccuracies in the solution to the constitutive equations (here, Newton's equations) that can lead to large fluctuations in E and drifiting over time. As a rule of thumb, your dt should be small enough able to capture the highest frequencies of vibrations to capture the physics of the system but not too small that you are wasting computational time. 3) Your potential has large gradients and the algorithm you have chosen to solve the equations of motion are unstable and/or inadequate. Here we choose the Verlet algorithm, which for toy systems performs suitably well. However, more complex potentials or simulations require additional considerations.

Other fundamental thermodynamic or physical relationships may also need to be enforced in the MD simulation. At minimum, the conservation of energy must be conserved.

It is worth noting that substantial work often goes into coding the potential function in research-level calculations. Checking for errors in the coding of the potential is also an important task. Such errors may not be obvious to detect, so one often turns to established knowledge or well-studied model systems (i.e., replicating prior literature results from someone else's calculation or some result from experiment).

As computational scientists, we think about - verifying the code: the code works correctly - validating the code: the code gives a reliable description of the physical system of interest

Beyond checking that the code works, one may think of code dissemination along the principle of FAIR (https://www.go-fair.org/fair-principles/). At the basic level, this consists of e.g., having readable code, thoroughly commenting your code, and making sure each version passes a set of pre-determined tests. In Python, there are recommended style guides and conventions (https://peps.python.org/pep-0008/). For large and mature codes, this often involves developing extensive user and developer documentation, including user tutorials and hosting user forums.

2 ========== Exercises ==========

We will study two types of systems: - a simple model of atoms on a surface, starting with a single atom - a bulk system described by an empirical potential (i.e., Lennard-Jones potential)

2.0.1 Atoms on a surface

We start with a simple toy model that will help demonstrate the basics of molecular dynamics simulations. Let's suppose we have a rigid surface that contains some atoms (starting with one) on the surface. There is an exchange of energy between the surface and atoms moving on top of the surface. The interaction between the surface and atom(s) is described with a simple potential that

is not designed to represent a realistic material. Regardless, there are several interesting states that are possible to drive the system into that are analogous to realistic systems.

We define our potential to be

$\phi(x, y) = A \sin(\pi x) \cos(\pi y)$

where A is some adjustable parameter that represents the strength of interaction. As usual, the force is the negative of the gradient of the potential.

We can then easily implement the potential and force.

We still start with the simulation of a single atom. Since "temperature" is a thermodynamic statistical average, it will be difficult to define an input target temperature. Instead, we will input the total energy, `ein`. We need to somehow initialize the surface before letting the molecular dynamics simulation to proceed.

First, we will first a random position near a well, calculate the potential and use $K = E - U$ to define a kinetic energy. Random velocities will be generated and then rescaled to give the appropriate K.

[5]:
```
%pycat ./code_exercises/2Dsurf.py
```

```
import numpy as np

# 2D surface

def phisurf(acon, x, y):
    """
    Surface potential
    """
    phi = acon * np.sin(np.pi * x) * np.sin(np.pi * y)
    return phi

def fsurf(acon, x, y):
    """
    Surface force
    """
    fx = -acon * np.pi * np.cos(np.pi * x) * np.sin(np.pi * y)
    fy = -acon * np.pi * np.sin(np.pi * x) * np.cos(np.pi * y)
    return fx, fy


def initsurf(ein, acon):
    """
    Pick x and y so they sit near a well
    """
    x = -0.75 + 1.5 * np.random.rand()
    y = -0.75 + 1.5 * np.random.rand()

    # Calculate potential at x, y
    phi = phisurf(acon, x, y)
```

```
    # Find kinetic energy such that total energy is ein
    ki = ein - phi

    # Pick random velocity components
    vxs = np.random.rand()
    vys = np.random.rand()

    # Find kinetic energy from random velocities
    ks = 0.5 * (vxs**2 + vys**2)

    # Scale velocities so total energy is equal to ein
    vx = vxs * np.sqrt(ki / ks)
    vy = vys * np.sqrt(ki / ks)

    return x, y, vx, vy
```

We will use the Verlet algorithm to solve the equations of motion in two dimensions. The discussion of the code involved so far is sufficient to start testing all of the functions with our simple 2D potential model.

2.1 ==== Single atom on surface ====

1. Using the Verlet algorithm, implement an MD code to explore the "atom on a surface" problem. Do not use periodic boundary conditions so that you may monitor large motions of your atom(s). Use E as an input parameter. You may use the provided code in constructing the MD code or implement the routines yourself.

a) Vary the time step and comment on the numerical error. Show plots of the calculated total energy. Is the total energy being integrated properly?

b) After determining the appropriate time step, run calculations with several values of E. What is the relationships you observe among E, U, and K?

c) Modify your code to calculate $\langle U \rangle$ and $\langle K \rangle$. What is relationship between temperature and $\langle K \rangle$?

d) For the same calculations, examine the behavior of the atom motion as a function of the energy of the atom. Comment on the relationship between the input energy and the ability of the atom to jump out of the well.

In all cases in d), plot the trajectory of the atom relative to the underlying potential.

You may visualize the potential as a contour plot. In python, you may find it helpful to define a grid of `(x,y)` coordinates in the following manner. First, define a vector x and y values

```
import numpy as np
x = np.arange(-10,10,0.1)
y = np.arange(-10,10,0.1)
```

Then, use the x and y vectors to define a two-dimensional grid using the built-in `numpy` function `meshgrid`

```
xmesh, ymesh = np.meshgrid(x,y)
```

You may then easily evaluate a function Z on the grid of coordinates generated

```
Z = f(xmesh,ymesh)
```

and use `matplotlib` built-in function `countour` plot.

2.2 Multiple atoms on the surface

2. Modify the code to include multiple atoms on the surface and to include periodic boundary conditions using the minimum image convention. Choose a system size that includes at least 5 surface unit cells in each direction. Having multiple atoms on the surface means interaction between the atoms is possible. We may choose to model that interaction with the Lennard-Jones potential (or any other appropriate potential).

We showed above how to compute energy and forces with the Lennard-Jones potential. In the above implementation, it is assumed that the periodic surface is a square lattice with lattice parameter 1. However, the Lennard-Jones potential in distance reduced units has its energetic minimum at $r_m = 2^{1/6} \approx 1.12$, which means that there would be a 12% mismatch between the assumed periodic surface and the interaction potential. We can vary this mismatch with a simple modification of the surface potential

$$\phi(x, y) = A \sin(\pi x/a) \cos(\pi y/a)$$

where a is the lattice parameter of the surface atoms, which can be varied.

a) Switch from an energy-based input to a temperature-based input for the initial velocities. Choose the initial velocities from either a normal distribution or pick a set of velocity components over a small range centered on zero and scale to the initial temperature chosen. Choose initial atomic positions near minimum(s) of the surface potential.

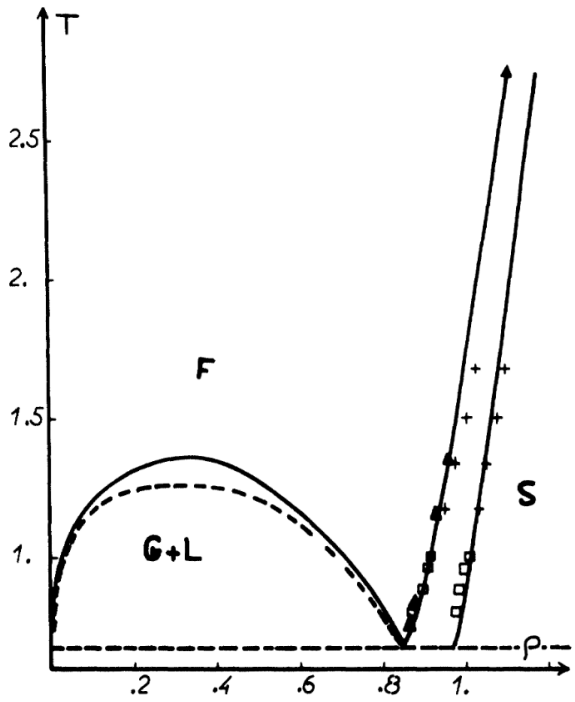
b) Pick two temperatures and two values for the interaction strength A. Run calculations until you observe equilibration in the energies and then compare the structure of the atoms on the surface. Do you observe clustering, repulsion, or something else? Note to take the minimum-image approach to scale back atom positions that fall outside of the central cell.

---

3. Repeat part 2b) with a different density of atoms on the surface.

2.3 Bulk Material

The hypothetical material "Lennard-Jonesium" is well studied (of course only by computation). A "phase diagram" of this material is presented in the figure below.



For the following exercises, you will want to restrict the densities (ρ) and temperatures (T) to remain in the solid (S) or liquid (L) regions. For example, at $\rho = 0$, the melting point occurs at $T = 0.7$.

---

4. Our first order of business is to determine a time step, which can often be estimated from what we know of the system being studied. Assume reduced units.

a) Estimate the approximate time step for the simulation in reduced units assuming a time step of 5×10^{-15} s. Use the mass of one of the rare gas atoms and the appropriate potential parameters as given in the text.

b) The period of an atom in a harmonic oscillator is $2\pi/\omega$, where $\omega = \sqrt{k/m}$. For a Lennard-Jones system at zero pressure and temperature, one can show that the force constant $k = 377\varepsilon/\sigma^2$. Assuming 50 time steps per period, calculate the time step in reduced units and compare with (a).

---

5. Assume some density and initial temperature for a Lennard-Jones system. Use a small system (i.e., $2 \times 2 \times 2$ supercell of fcc cells (N = 32) or $3 \times 3 \times 3$ supercell (N=108). Assume the time step in Problem 4. Run the system for a few hundred steps. Is this time step small enough, based on the energy criterion above? Your goal is the biggest time step possible that still gives a good solution. Note that for small systems it may be difficult to reach energy fluctuations $< 10^{-4}$.

---

6. Pick a density and temperature such that the material is in the **solid** phase. Using the selected

time step, run a simulation long enough to reach equilibration (typically a few thousand time steps).

a) Plot the various thermodynamic quantities (E, T, K, P). By what time step has the system reached equilibrium?

b) Average the various thermodynamic quantities over the equilibrated run. Note, it may be easier to output the calculated quantities as time sequences and then average everything in a separate routine. Remember to omit the initial part of the time sequence when the system has not quite equilibrated.

---

7. You may have noticed that the equilibrilation time can take awhile. The code in its current form takes a while to equilibrate. Currently, the code is written such that it *always* starts from a perfect lattice and a set initial temperature. Thus the system must equilibrate each time it is run, which is rather inconvenient. Instead, one can run the initialization routines that sets initial positions and velocities first. Then you can change the input and outputs of the main MD routine as folows

```
def MDLJi(density, nsteps, dt, natoms, x, y, z, vx, vy, vz)
    """
    MD code for LJ atoms with initialization modification

    Input:
        density (float): density of Lennard-Jonesium
        nsteps (integer): number of time steps in MD run
        dt (float): time step
        natoms (integer): number of atoms in cell
        x, y, z (array of floats): atomic positions
        vx, vy, vz (array of floats): atomic velocities
    Output:
        un (array of floats): time series of potential energy per atom
        kn (array of floats): time series of kinetic energy per atom
        en (array of floats): time series of total energy per atom
        tn (array of floats): time series of temperature
        pn (array of floats): time series of pressure
        a (float): cell dimension
        x, y, z (array of floats): time series of atomic positions
        vx, vy, vz (array of floats): time series of atomic velocities

    """
    return un, kn, en, tn, pn, a, x, y, z, vx, vy, vz
```

then remove the call to `initLJMD` in the MD code. As the inputs have changed from before, you may find that renaming the function may be of help to distinguish it from previous versions.

Now the MD code reads in the positions and velocities, follows through the calculation and outputs the *final* positions and velocities.

When running the code, the MD could look like

```
# nc (integer) = number of repeated cells
# tin (float) = desired temperature for velocities
natoms, x, y, z, vx, vy, vz = initLJMD(nc, tin)
un,kn,en,tn,pn,a,x,y,z,vx,vy,vz = MDLJi(density, nsteps, dt, natoms, x, y, z, vx, vy, vz)
```

After an MD calculation, we can take `x,y,z,vx,vy,vz` (i.e., the final positions and velocities) and restart a new MD calculation without the need for equilibrating by calling our MD routine again

```
un,kn,en,tn,pn,a,x,y,z,vx,vy,vz = MDLJi(density, nsteps, dt, natoms, x, y, z, vx, vy, vz)
```

Modify the code accordintly and redo the calculations from Problem 4, restarting the calculation until you have a stable and equilibrated system. Compute averages. Analyze whether there is a drfit in the calculation as discussed in lecture. Have you made a long enough run (i.e., have you used enough time steps) to get good statistical averages?

---

8. Pick a density and temperature that woud put the system in the **liquid** phase.

a) Using the optical time step, run a few thousand time steps (this should take a few minutes) and plot the various thermodynamic quantities.

b) By what time stpe has the system equilibrated?

c) Find average quantities over the equilibrated part of the run. Analyze whether there is a drift in the calculation as discussed in lecture. Have you have a long enough run (i.e., have you used enough time steps to get good statistics?

---

9. Using the data you have generated in part 6., calculate the fluctuations for the various thermodynamic quantities (e.g., $\langle U\rangle^2 - \langle U^2\rangle$). Note: these fluctuations are not an estimate of the error of the calculation, but rather to related to other thermodynamic quantities. To estimate error, we need to follow the procedure of binning the data and finding the standard deviation of the averages in the bins. You may find it useful to create separate, reusable routines for evaluating statistics of trajectories.

---

10. Compare your calculations for the solid and liquid phases. In order to determine whether you are in the solid or liquid phase, you need to monitor the structure of the material. For this, we turn to the radial distribution function $g(r)$. The radial distribution function gives the probability that an atom is located within a thin shell $r + dr$ away from a reference atom. It is customary to normalize the integral of $g(r)$ to give the number of atoms within that distance. The appendix to Ch. 6 of Lesar discusses how to calculate $g(r)$.

Below we present a simple code to compute $g(r)$, using the normalization given in the text appendix. Such a code could be used as a post-processing step once a set of atomic positions is known.

[ ]:
```
import numpy as np

def gr(a, n, x, y, z, nbin):
    """
    Calculate radial distribution function.
```

```
    Input:
        a (float): simulation cell dimension
        n (integer): number of atoms
        x, y, z (array of floats): atomic positions
        nbin (integer): number of bins
    Output:
        bp (array): binning used in g(r)
        ng (array): frequencies corresponding to g(r)
    """
    # cutoff, bin size
    rc = a / 2
    xb = rc / nbin
    g = np.zeros(nbin)
    bp = np.zeros(nbin)
    ng = np.zeros(nbin)

    for i in range(n - 1):  # Note limits
        for j in range(i + 1, n):  # Note limits
            # Minimum image convention
            dx = x[j] - x[i]
            dy = y[j] - y[i]
            dz = z[j] - z[i]
            dx -= round(dx)
            dy -= round(dy)
            dz -= round(dz)
            dist = a * np.sqrt(dx**2 + dy**2 + dz**2)
            if dist <= rc:
                ib = int(dist / xb)
                if ib < nbin:
                    g[ib] += 1

    # For computing g(r)
    # Normalize and create proper distances
    factor = 2 * a**3 / (4 * np.pi * n**2 * xb)
    for i in range(nbin):
        bp[i] = (i + 0.5) * xb
        ng[i] = factor * g[i] / ((i * xb)**2) if i != 0 else 0

    return bp, ng
```

From the final configuration of Problems 8 and 6, calculate $g(r)$ for the solid and liquid phases. For example, a command could look like

```
r,g = gr(a, n, x, y, z, nbin)
plt.plot(r,g)
```

Can you tell the difference between the structures of the solid and liquid phases?

One of the uncertainties that you will face is that the data is noisy when you use only the final configuration. In reality, $g(r)$ should be averaged over a long (equilibrated) time sequence, as it is also an average quantity. One way to do this would be to modify the force routine (where you already compute distances; you may have noticed the code is similar to the force routines) and then do the binning operations as shown in the above code. Accumulate those values and then average at the end of the run, apply the normalization factor, and plot. Try this.

---

11. As discussed in class, we may also define a structural order parameter that can discriminate between solids and liquids

$\rho(\vec{k}) = \frac{1}{N}\sum_{i=1}^{N}\cos(\vec{k}\cdot\vec{r})$.

For an fcc lattice, one may choose $\vec{k} = (2\pi/a_{fcc})(-1, 1, -1)$ where a_{fcc} is the lattice parameter of the unit cell (not the simulation cell). Implement a routine that computes this structural order parameter for a final configuration of a run. Does your computed structural order parameter agree with whether the cell is in a solid or liquid state? How does the information from the structure order parameter compare with $g(r)$?

---

12. Next, we turn to the *velocity autocorrelation function*, one of the the most important quantities one can compute in an MD calculation. The velocity autocorrelation function is defined as

$c_{vv}(t) = \frac{m}{3k_BT}\langle\vec{v}(t)\cdot\vec{v}(0)\rangle$

where the average $\langle\rangle$ is taken over all particles. One reason the velocity autocorrelation function is important is that it is related to the diffusion constant. Compute the diffusion constant of the fcc lattice for the solid and the liquid. What differences do you see in the diffusion constant of the solid versus the liquid?

Hint: you will need to make the following modifications to the code. Store the initial velocities v_{x0}, v_{y0}, v_{z0}. At each time step k calculate

```
cvv[k] = 0

for i in np.range(natoms):
    cvv[k] = cvv[k] + vxo[i]*vx[i] + vyo[i]*vy[i] + vzo[i]*vz[i]

cvv[k] = cvv[k]/natoms
```

where i the index of the index over atoms. Note that this is a very rudimentary wayto calculate the autocorelation function. Other approaches are discussed in the text/lecture.

References

1. J.P. Hansen and L. Verlet, "Phase transitions of the Lennard-Jones system." Phys. Rev. 184, 151-161 (1969)
2. R. LeSar and J.M. Rickman, "Finite-temperature properties of materials from analytical statistical mechanics." Philosophical Magazine B 73, 627-639 (1996).