Computational Design of Materials Lab 4

Andrew Sullivan

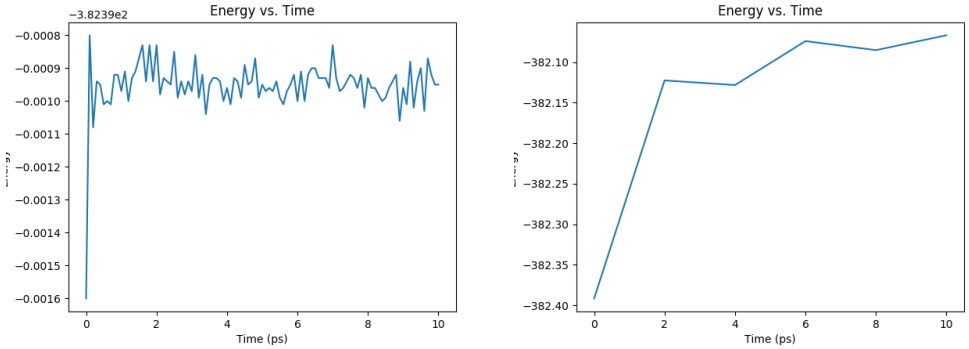
April 18, 2019

1 Problem 1: Integrator Accuracy

In this problem, we simulate bulk aluminum in an FCC cell with periodic boundary conditions and EAM interatomic potentials. A velocity Verlet integrator is employed in LAMMPS with a microcanonical (NVE) ensemble for molecular dynamics simulations. All thermodynamic quantities are returned every 100 steps, so all plots as a function of time are sampled using this rate.

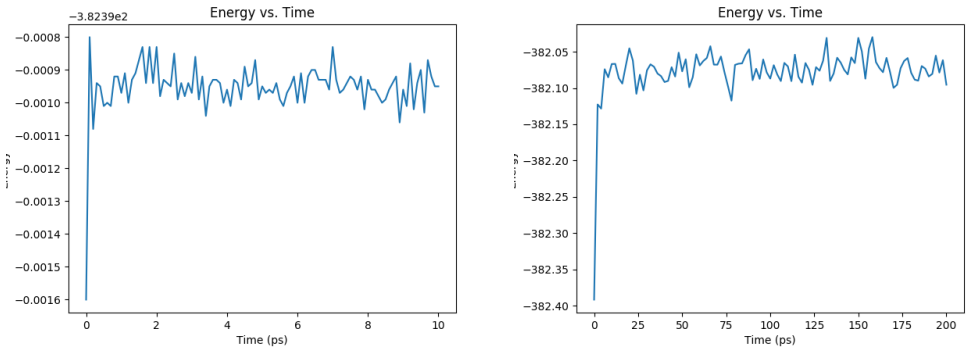
1.1 A: Optimization of Step Size

We begin by exploring optimization of the integrator step size by fixing the initial temperature at 300K and supercell size of 3x3x3 and vary the timestep size between 0.001 and 0.02 ps. The total number of steps is first varied to produce a constant simulation length of 20 ps, and then a fixed step size of 10,000 is explored. We first determine the proper way to take the average energy by plotting the average energy, calculated by averaging all values between the i^{th} point and the final point (i.e. $E_{avg} = \frac{1}{N-i+1}\Sigma_{j=i}^{N}E_j$), as a function of the first energy value included in the average. This calculation is performed at multiple timestep values. Representative plots of energy vs. time are presented below for values near either end of the range.



(a) Timestep = 0.001 ps. (b) Timestep = 0.02 ps.

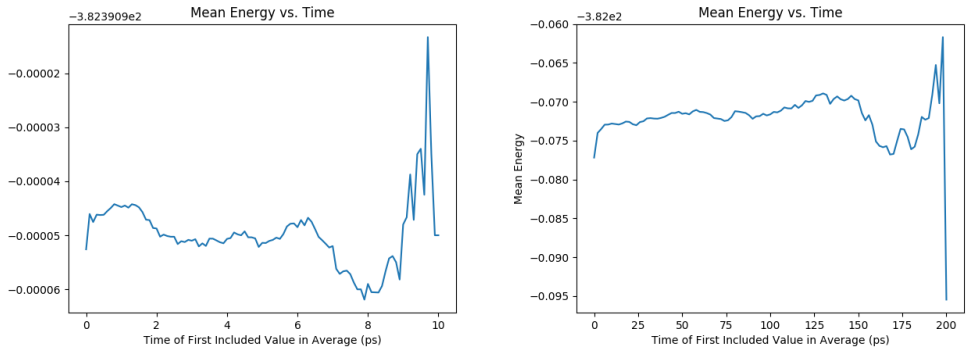
Figure 1: Timestep dependence of energy for fixed simulation time.



(a) Timestep = 0.001 ps. (b) Timestep = 0.02 ps.

Figure 2: Timestep dependence of energy for fixed number of iterations.

In Figure 1, the evolution of the system energy is plotted as a function of time. We can see clear convergence to an energy around which the value fluctuates in the first subfigure, but as the timestep is increased, this convergence is less obvious. This is a result of keeping the total simulation time fixed. Using a length of 20 ps allows for the energy to converge for small timesteps, since these will go through more iterations as the simulation progresses. To address this issue, we next look at the same trends with the



(a) Timestep = 0.001 ps. (b) Timestep = 0.02 ps.

Figure 3: Timestep dependence of average energy.

number of steps fixed at 10,000, allowing the total simulation time to vary. In Figure 2, the evolution of

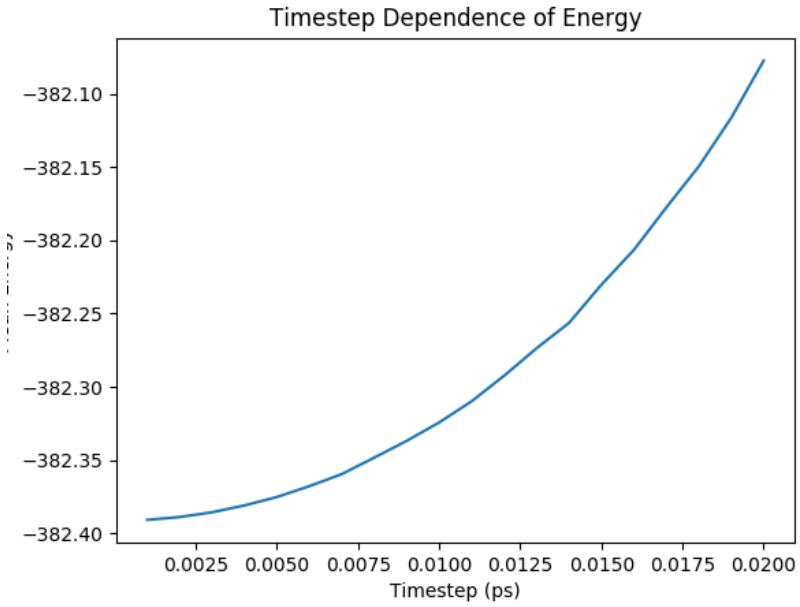


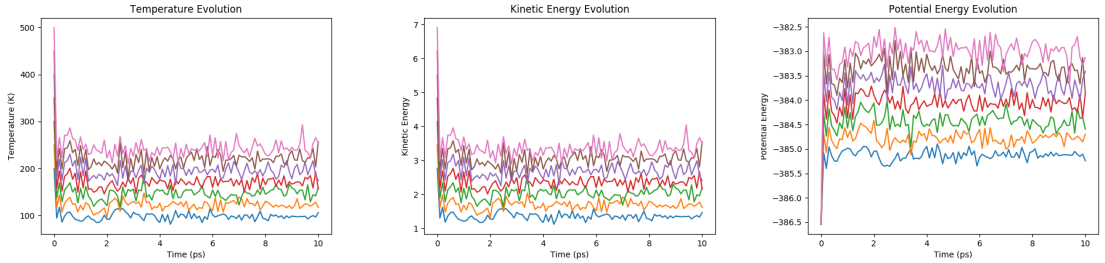
Figure 4: Timestep dependence of average energy for fixed number of iterations.

system energy is plotted as a function of time over 10,000 iterations. It is clear from these figures that the energy will evolve with qualitative similarity for varying timesteps given a sufficient number of iterations. Similarly, Figure 3 shows the evolution of the average energy, as defined above. We can see that all cases begin with a value far from the converged average, and tend to deviate from the average as less points are included. This latter behavior is expected, since including only ten or fifteen points in the average will make it more susceptible to fluctuations than including a fifty or sixty. By examining all three plots above, along with others within the designated range that are not pictured, we can observe that an approximately constant average energy is obtained by including approximately the last 60% of the energy values in the average. To form a more robust detection of converged average energy, we compute the median of all points along the average energy curve, and beginning at the first value, which includes all points in the simulation, we scan through until we find the first point which deviates by less than two standard deviations from the median. Comparing the returned value to the energy vs. time traces, we find good agreement between the detected

values and the points at the center of the energy fluctuations after convergence. These energy values are then used to examine the energy vs. timestep trend presented in the above figures, which exhibits a quadratic-like dependence. Though it is apparent that decreasing timesteps to values around 0.001 ps produce the most accurate energy values, as expected, we can also observe that the gain in accuracy becomes consistently smaller as this value is approached. Depending on the required accuracy, values close to this minimal cutoff but not exactly at the minimum are likely to suffice, especially if energy differences are of interest rather than absolute magnitudes. For example, timestep values of 0.0025 ps are likely to give sufficient accuracy for many problems, and will improve computational efficiency over using smaller timesteps.

1.2 B: Temperature Evolution

In this section, we fix the timestep to be 0.001 ps, maintain the above supercell size of 3x3x3, and vary the temperature between 200K and 500K for simulations of length 10000 steps.



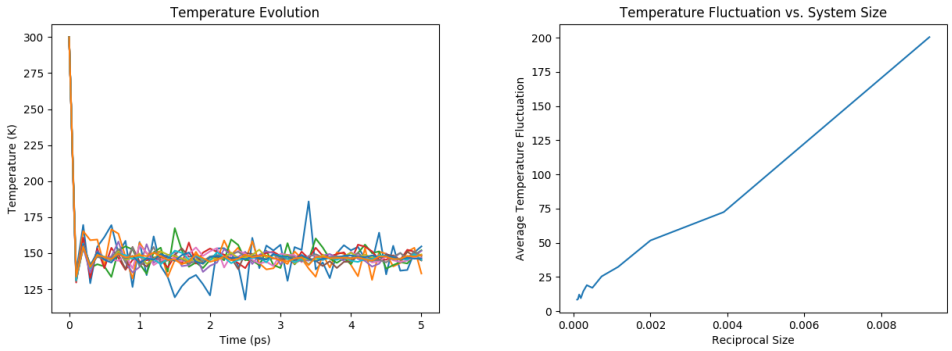
(a) Temperature Evolution. (b) Kinetic Energy Evolution. (c) Potential Energy Evolution.

Figure 5: Time evolution of temperature, and kinetic and potential energies.

We next fix the timestep to be 0.001 ps and keep the supercell size the same at 3x3x3. The above figure shows the evolution of instantaneous temperature as a function of time at multiple temperature inputs (between 200 and 500 K in increments of 50 K). If we plot the kinetic and potential energies as well, we can see the thermodynamic origin of the initial change in temperature- in the microcanonical ensemble, energy is a fixed quantity, but may be partitioned differently between kinetic and potential spaces. The kinetic energy is directly proportional to the temperature, and thus these two quantities change with identical fluctuations. As the total energy is constant, these fluctuations directly oppose those in the potential energy. Depending on the initial configuration of position and velocity, the potential energy and kinetic energies are likely to not be at their equilibrium value. As the simulation progresses, energy is transferred between kinetic and potential forms as the system stabilizes to its equilibrium value from its initial non-equilibrium conditions as a consequence of the random sampling of velocities from a Boltzmann distribution. As the initial temperature input increases, the equilibrium value of the temperature increases as well (the pink curve denotes temperature evolution for an initial condition of 500 K, while the blue curve is for 200 K), but using this ensemble makes it difficult to set the desired temperature, since it will typically change from the initial value quite substantially. This behavior may be better controlled by introducing a thermostat and thereby using a canonical (NVT) ensemble rather than an NVE ensemble. A number of different thermostats are available and have their own set of advantages and disadvantages. Velocity rescaling directly scales the velocity by the square root of the ratio of the desired and instantaneous temperature values, while weakly coupled methods like the Berendsen thermostat use a fictitious external heat bath to incrementally adjust the temperature towards the target. The Nosé-Hoover thermostat, which we use in the following question, the Lagrangian of the system is modified by introducing an extra variable, s, which represents a frictional force acting to accelerate particles towards the target temperature.

1.3 C: Supercell Dependence

We now explore the dependence on system size of energy and its fluctuations in the microcanonical ensemble. The supercell size is varied between 3 and 15, and the temperature and timestep are fixed at 300 K and



(a) Temperature vs. Size.

(b) Fluctuations vs. Reciprocal of Size.

Figure 6: Dependence of temperature on system size.

0.001 ps, respectively. Due to high computational cost for large system sizes, we restrict simulation length to 5 ps. We compute the average energy as before, and we estimate the fluctuations in the system as the standard deviation of the instantaneous energy vs. time curve for all values above the detected cutoff using the median method described above. In "Thermodynamic theory of equilibrium fluctuations" (*Annals of Physics* 2015), Mishin shows that, for a system with fixed volume and number of particles, the stiffness matrix reduces to a scalar value $K_{11} = \frac{1}{Nkc_v}$. The covariance of two thermodynamic properties is given by $\overline{\Delta x \Delta y} = \Sigma_{i,j=1}^{m}(\frac{\partial x}{\partial Z_i})(\frac{\partial y}{\partial Z_j})K_{ij}^{-1}$, where thermodynamic variables x and y are functions of the extensive system variables $\{Z_i\}$. Evaluating the variance of energy (which in our microcanonical ensemble instead refers to kinetic energy, since total energy does not change) by inputting E for both thermodynamic variables and S as the extensive quantity Z, we obtain $\overline{(\Delta E)^2} = (\frac{\partial E}{\partial S})_{V,N}^2 K_{11}^{-1} = NkT_0^2 c_v$. As in "Temperature fluctuations in canonical systems: Insights from molecular dynamics simulations" (2016), the non-equilibrium temperature can be defined as $T - T_0 = \frac{E-E_0}{Nc_v}$, or $\Delta T = \frac{\Delta E}{Nc_v}$. Inserting this equality into the above expression for the energy variance yields the variance in temperature, $\overline{(\Delta T)^2} = \frac{kT_0}{Nc_v}$. This suggests that the average squared deviation of temperature should be proportional to the reciprocal of the system size, N. In Figure 6b, we plot the average squared deviation, calculated by finding the mean temperature using the median method defined above and squaring the difference between this value and each instantaneous temperature value with an index greater than or equal to the corresponding cutoff for the mean temperature, averaged over these time points. The reciprocal size is defined as $\frac{1}{N}$, where N is the number of atoms in the system, calculated as $4s^3$, where s is the input size parameter, used to create an s x s x s supercell with each FCC unit cell containing 4 atoms. We can see from the plot that this yields an approximately linear relationship, as we expect.

2 Problem 2: Melting Temperature of Bulk Al

In this problem, we consider the same system as above (bulk FCC Al) in a canonical ensemble (NVT) and adjust the temperature to simulate melting, which is observed using a variety of readily available system parameters relating to energetics, structural correlation, and atomic kinetics. As in problem 1, all thermodynamic quantities are returned every 100 steps in order to reduce correlation between adjacent points and to reduce the required memory.

2.1 A: Parameter Optimization Using NVT Ensemble

In order to simulate the melting of bulk FCC Al, we first search the temperature space to find the approximate region in which melting occurs. To do so, we begin by simulating with a supercell size of 3x3x3 and timestep of 0.001 ps as before for 20 ps simulations over a temperature range from 200 to 1300 K in 100 K increments. One way of detecting melting is the presence of an initial upswing in the energy vs. time plot corresponding

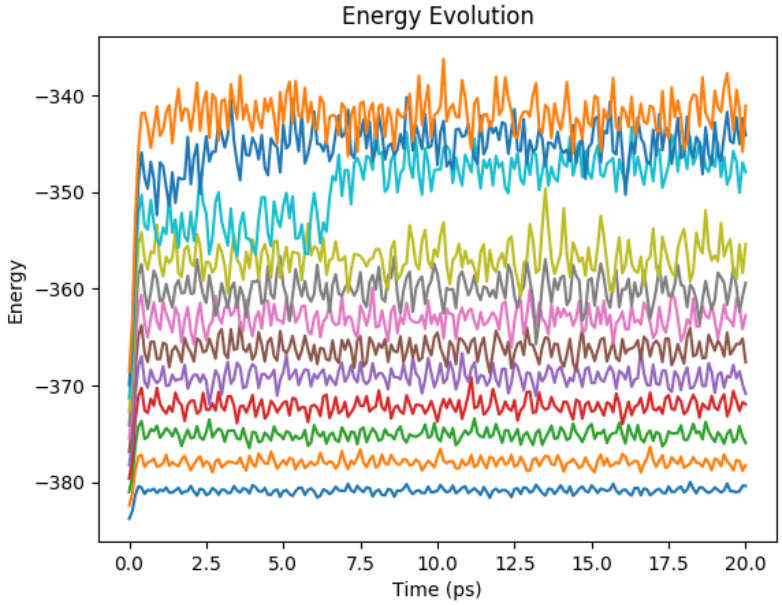
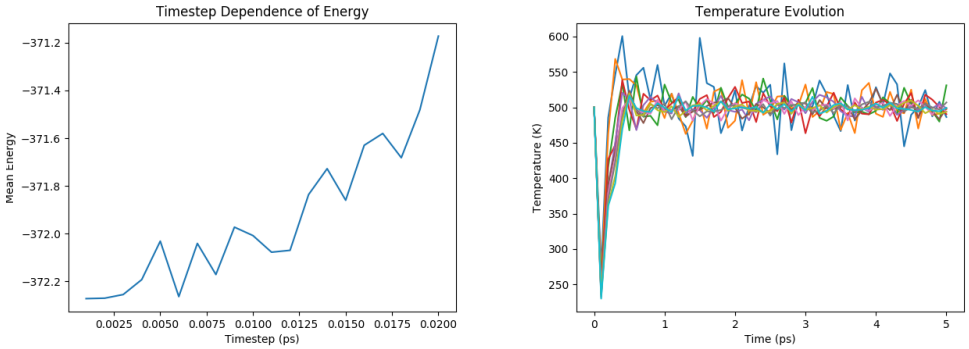


Figure 7: System energy evolution as a function of temperature.

to the latent heat of fusion, which is evident in the results from this simulation between 1000 and 1200 K. The state at 1100 K is at an intermediate stage of melting, hence the delayed increase in energy around 5 ps.



(a) Energy vs. Timestep.

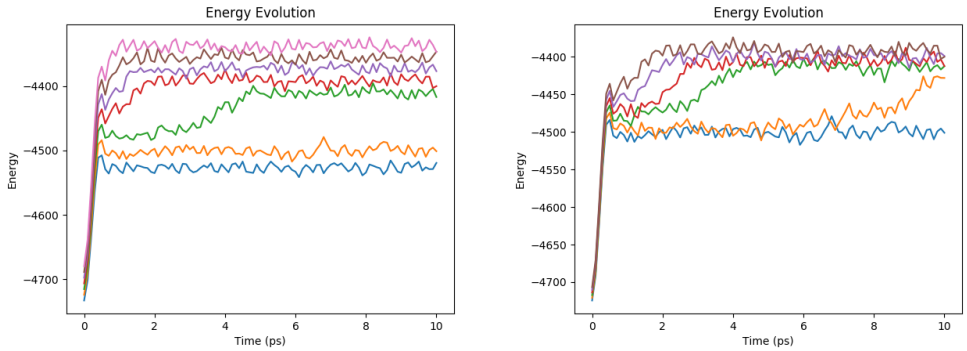
(b) Temperature vs. Size.

Figure 8: Parameter Optimization.

We next examine the trends in timestep and supercell size in a regime below our melting point but still in the NVT ensemble, i.e. at 500 K. We see similar trends to those found above, i.e. that the energy decreases as a function of timestep and levels off around 0.0025 ps and that temperature fluctuations are far more pronounced for smaller systems (the blue line corresponds to the smallest tested case at 3x3x3). The same timestep range (0.001 to 0.02 ps) was tested as earlier, and a system size range between 3 and 13 was examined. Average energies for timestep were calculated using the same method as earlier, i.e. by finding the first average energy which deviates by less than two standard deviations from the median. It is not surprising that the mean energy values show more variation than in the previous case, since energy is not a fixed quantity in our canonical ensemble. Even so, the trend appears qualitatively similar. As before,

simulations for timestep optimization are 10,000 steps in length and those for supercell size are 5,000. For the following calculations, we use a timestep of 0.001 ps and a supercell size of 7x7x7, though slightly less computationally intensive parameters are likely to be suitable as well (e.g. 0.0025 ps and a 5x5x5 supercell), and all simulations are 10 ps long.

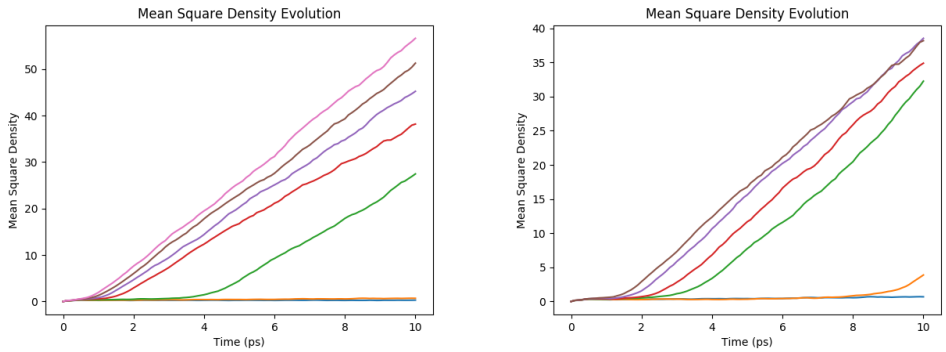
2.2 B: Determination of Melting



(a) 1000 to 1300 by 50 K. (b) 1050 to 1150 by 20 K.

Figure 9: Melting in Energy vs. Time Plots.

As mentioned above, one method of identifying the melting point is by plotting the instantaneous energy as a function of time and identifying the phase transition by the latent heat of fusion. With our optimized supercell and timestep parameters, we begin by plotting the region between 1000 and 1300 K in increments of 50 K. As we observed before, the phase transition is present within the range of 1050 to 1150 and is complete by the end of this range. For a final increase in our precision, we simulate between 1050 and 1150 K in increments of 20. From these results, we can see that the incomplete melting has begun by 1070 K and has essentially completed by 1130 K, or 857° C.

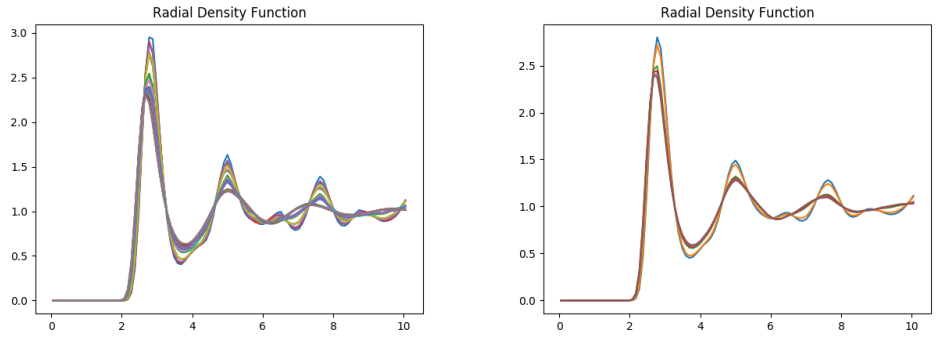


(a) 1000 to 1300 by 50 K. (b) 1050 to 1150 by 20 K.

Figure 10: Melting in Mean Squared Displacement vs. Time Plots.

Rather than examine the change in total energy as a function of time, we may also examine the effect of temperature on the kinetics and atomic motion in the system. One such measure of kinetics is the mean square displacement of the atoms, *msd*, defined as the average squared displacement of all atoms relative to their resting position over time. If we examine this quantity as a function of time, we observe a similar temperature dependence between 1000 and 1300 K in increments of 50 K (**a)**), and between 1050 and 1150

K in increments of 20 K (**b)**), in which there is a clear difference in the behavior of the *msd* at the melting point. As before, we observe the beginning of melting around 1070 K, after which there is a large change in the displacement which stabilizes around 1130 K. Evolving mean squared density indicates that the atoms are moving away from their initial positions, and thus the slope is related to the rate at which these atoms move away over time, i.e. the diffusion coefficient. For unmelted conditions, the *msd* remains very close to zero, consistent with the atoms not diffusing within the crystal lattice. As the solid melts, the new liquid atoms are more free to move.



(a) 1000 to 1300 by 50 K.

(b) 1050 to 1150 by 20 K.

Figure 11: Melting in Radial Density Function Plots.

A final method we test to probe the melting transition relates to the structure of the system in the form of the radial density function (*rdf*), which captures the average density as a function of the distance r from a typical atom. For a periodic solid, the radial density function is expected to have sharp periodic peaks corresponding to interatomic distances. As the system evolves towards a liquid, however, peaks after the first (representing the average interatomic distance) decrease in sharpness and magnitude, as the periodicity of the lattice is disrupted, ultimately leading to small oscillations that quickly fade away as a function of distance. Over the 1000 to 1300 K range, we can clearly observe the presence of a phase transition by looking at the first peak in the radial density, shown as a gap between the solid and liquid states. These gaps are apparent at the other peaks and valleys in the curves as well, with the sharper peaks corresponding to the solid state. However, the exact temperatures of each curve are not readily identifiable from this plot, since there are four times as many curves as in the energy or mean square displacement. This is a consequence of the algorithm used to obtain the radial density function, which returns four 2 x 100 vectors, corresponding to four different *rdf* profiles during the evolution of the system. In each, the first column corresponds to normalized distance units, and the second to the values of the *rdf* for those units. In order to identify the specific temperatures associated with each curve, we average the *rdf* amongst all four calculated profiles for a given temperature, and plot the results over the finer range between 1050 and 1150 K in 20 K increments. While this removes some of the time evolution information, we can now clearly see that these results are consistent with the previous case, in which the first two curves (1050 and 1070 K) show a solid-like profile with sharp peaks and periodicity, after which melting begins and the *rdf* transitions to the damped periodic fluctuations characteristic of a liquid.

Ultimately, all three of these methods provide qualitatively similar results for the melting. By examining structural correlation, atomic kinetics, or total energy of the system, we observe the onset of melting by 1070 K and completion of the transition by 1130 K. The experimental value for the melting temperature of bulk FCC aluminum is substantially lower than this calculated value, around 660° C or 933 K. This is consistent with our expectations for the simulation due to the interfacial energy, which counteracts the free energy change associated with melting and will cause a nucleus of the energetically stable phase (based on free energy differences) to disappear until some critical nucleus size is reached. The requirement for the formation

of this nucleus necessitates some degree of superheating (or supercooling for solidification) to produce the phase transition, and so the observed melting temperature actually reflects some value $T + \Delta T_{superheat}$.

2.3 C: Other Methods to Determine Phase Transitions

There are a number of other quantitative methods that can be used to determine the melting temperature. The specific heat capacity, defined as the partial derivative of energy with respect to temperature at constant volume, is not constant but changes as a function of system temperature and phase. At the phase transition, the temperature remains constant but the energy changes, and so a specific heat vs. temperature plot should show a discontinuity at the transition point. Though not explicitly explored above, the code used in LAMMPS also returns the pressure and density of the system. The pressure dependence of the phase transition can be explored in an analogous manner to the temperature (by using any of the aforementioned methods), since the system N and V are held constant in the canonical ensemble. From the transition pressure, the temperature can be estimated using the ideal gas law (or other relations more applicable to the solid-liquid transition). Like other quantities, the system density should also show a change at the phase transformation point. Below melting, density will increase as a function of temperature due to larger atomic vibrations (thermal expansion), and this should similarly be true above the transition. However, the slope of the ρ vs. T plot should be discontinuous at the transition, since the liquid will likely expand faster than the solid. LAMMPS also has the capability of computing many other system quantities which are not included in the above code, but may be useful in identifying a phase transition, including the entropy per atom (which should increase above the transition), system enthalpy (which should evolve similarly to energy), or atomic coordination. Finally, changing the system itself may be useful in identifying the true melting point. By simulating an FCC crystal next to a region of liquid at various temperatures and observing which phase expands and which contracts, one can more accurately identify the melting temperature as the point at which both phases remain at their initial size since the hysteresis induced by nucleation would not be present due to the presence of the interface in the initial system.

3 Problem 3: Scalability of MD Systems

3.1 A

An FCC unit cell contains 4 atoms, and therefore an NxNxN supercell will contain $4N^3$ atoms.

3.2 B

If all atoms are considered and no force cutoff is implemented such that each atom interacts with all other atoms, there are $4N^3(4N^3 - 1)$ total pairs, but as each pairwise force need only be determined once per timestep, we can divide this value by 2, yielding $8N^6 - 2N^3$ calculations. Introducing a potential cutoff should significantly reduce the number of required calculations, dependent on the magnitude of the cutoff. In crystalline solids, with approximately constant interatomic distances, this cutoff can be introduced as only including n nearest neighbors for each atom. Therefore, for each of the $4N^3$ atoms, we compute n interactions, and once again divide by 2 to prevent double-counting, yielding $2nN^3$ calculations. If interatomic distances vary significantly, however, we may not be able to strictly include only n atoms, requiring instead that the interatomic distances be computed first, and the number of atoms within some cutoff radius determined before the force calculations. In this case, we cannot determine the exact number of calculations beforehand, as it may vary for each step. The interatomic distances must be computed once per each timestep, in order to calculate the forces between atoms, using pairwise potentials, DFT, or another method.

3.3 C

If we consider the case without any potential cutoffs, we can perform $35x10^{12}$ operations in a second, including both distance calculations and force calculations. For each pair of atoms, the interatomic distance and the resultant force must be calculated, requiring 2 floating point operations per atomic pair. Therefore,

we have $8N^6 - 2N^3 = \frac{35}{2}x10^{12}$, or $8N^6 - 2N^3 - 17.5x10^{12}$. Solving this polynomial yields one positive real root at 113.9, so 113 particles can be computed using pairwise calculations without cutoffs in one second.

3.4 D

Depending on the nature of the system, a potentially beneficial means to reduce the time spent calculating distances is to use neighbor lists, i.e. lists of atoms that are within some "cutoff" distance from a given atom. These can be updated periodically by keeping track of how much atoms move, and if this movement places another atom within the cutoff distance, it can be added to the relevant neighbor list. Periodic boundary conditions may also be useful in reducing the number of distance calculations, but also reduce the number of unique atoms in the system, which may be undesirable if long-range effects are present.

3.5 E

Lennard-Jones potentials are simple pair potentials that can be calculated quite rapidly between each pair of atoms (or some reduced set if a neighbor list is used instead). Use of LJ potentials also does not consider any many-body interactions. For these reasons, LJ pair potentials are the least computationally expensive energy model. Due to their failure in metals, the embedded atom method (EAM) is commonly used to consider bond energy nonlinearity and local many-body effects. However, this (slightly) increases the computational cost above LJ potentials. The most computationally expensive force calculation is the Hartree-Fock (HF) method, as it requires variational minimization of the set of basis orbitals using matrix diagonalization, which is computationally challenging. Introduction of excitations, for example in CI, further increases the complexity and cost. Depending on the nature of the system, HF methods often exhibit scaling of N^3 - N^4 for number of electrons N. Given that DFT is similarly a quantum mechanical method that requires iteration until self-consistency involving matrix diagonalization, it is similarly expensive computationally. Depending on the nature of the system and the choice of exchange-correlation functional, the difference in computational cost between HF and DFT may vary. LDA or GGA methods often provide favorable scaling over HF (N^2 - N^3), but more complex functionals or hybrid methods will become more expensive. Therefore, in general, these four methods can be ranked from least expensive to most expensive as: LJ, EAM, DFT, HF, but the order of these latter two is subject to change depending on system and algorithm parameters.

4 Appendix

The Python code used to run LAMMPS calculations is included below. The main body is divided into seven subsections, run by setting the Q parameter and corresponding to different portions of the solution (e.g. varying timestep or temperature or supercell size, or using NVE vs. NVT ensembles).

```
from my_labutil.src.plugins.lammps import *
from ase.spacegroup import crystal
from ase.build import make_supercell
import matplotlib.pyplot as plt
import numpy as np


def make_struc(size):
    """
    Creates the crystal structure using ASE.
    :param size: supercell multiplier
    :return: structure object converted from ase
    """
    alat = 4.10
    unitcell = crystal('Al', [(0, 0, 0)], spacegroup=225, cellpar=[alat, alat, alat, 90, 90, 90])
    multiplier = numpy.identity(3) * size
    supercell = make_supercell(unitcell, multiplier)
    structure = Struc(ase2struc(supercell))
    return structure


def compute_dynamics(size, timestep, nsteps, temperature, ensemble):
    """
    Make an input template and select potential and structure, and input parameters.
    Return a pair of output file and RDF file written to the runpath directory.
    """
    if ensemble == 'nve':
        intemplate = """
        # ---------- Initialize simulation ---------------------
        units metal
        atom_style atomic
        dimension  3
        boundary   p p p
        read_data $DATAINPUT

        pair_style eam/alloy
        pair_coeff * * $POTENTIAL  Al

        velocity  all create $TEMPERATURE 87287 dist gaussian

        # ---------- Describe computed properties---------------
        compute msdall all msd
        thermo_style custom step pe ke etotal temp press density c_msdall[4]
        thermo $TOUTPUT

        # ---------- Specify ensemble ---------------------
        fix  1 all nve
        #fix  1 all nvt temp $TEMPERATURE $TEMPERATURE $TDAMP

        # ---------- Compute RDF ---------------
        compute rdfall all rdf 100 1 1
        fix 2 all ave/time 1 $RDFFRAME $RDFFRAME c_rdfall[*] file $RDFFILE mode vector

        # ---------- Run ---------------
        timestep $TIMESTEP
        run $NSTEPS
        """

    elif ensemble == 'nvt':
        intemplate = """
        # ---------- Initialize simulation ---------------------
```

```
        units metal
        atom_style atomic
        dimension  3
        boundary   p p p
        read_data $DATAINPUT

        pair_style eam/alloy
        pair_coeff * * $POTENTIAL  Al

        velocity  all create $TEMPERATURE 87287 dist gaussian

        # ---------- Describe computed properties ----------
        compute msdall all msd
        thermo_style custom step pe ke etotal temp press density c_msdall[4]
        thermo $TOUTPUT

        # ---------- Specify ensemble ----------
        #fix  1 all nve
        fix  1 all nvt temp $TEMPERATURE $TEMPERATURE $TDAMP

        # ---------- Compute RDF ----------
        compute rdfall all rdf 100 1 1
        fix 2 all ave/time 1 $RDFFRAME $RDFFRAME c_rdfall[*] file $RDFFILE mode vector

        # ---------- Run ----------
        timestep $TIMESTEP
        run $NSTEPS
        """

    potential = ClassicalPotential(ptype='eam', element='Al', name='Al_zhou.eam.alloy')
    runpath = Dir(path=os.path.join(os.environ['WORKDIR'], "Lab4/Problem1/",str(size) + "_"
    + str(timestep) + "_" + str(temperature)))
    struc = make_struc(size=size)
    inparam = {
        'TEMPERATURE': temperature,
        'NSTEPS': nsteps,
        'TIMESTEP': timestep,
        'TOUTPUT': 100,                 # how often to write thermo output
        'TDAMP': 50 * timestep,        # thermostat damping time scale
        'RDFFRAME': int(nsteps / 4),   # frames for radial distribution function
    }
    outfile = lammps_run(struc=struc, runpath=runpath, potential=potential,
                                intemplate=intemplate, inparam=inparam)
    output = parse_lammps_thermo(outfile=outfile)
    rdffile = get_rdf(runpath=runpath)
    rdfs = parse_lammps_rdf(rdffile=rdffile)
    return output, rdfs


def md_run(size,timestep,nsteps,temperature,ensemble):
    output, rdfs = compute_dynamics(size=size, timestep=timestep, nsteps=nsteps, temperature
    =temperature, ensemble=ensemble)
    [simtime, pe, ke, energy, temp, press, dens, msd] = output
    ## ------- plot output properties
    return energy,simtime,temp,pe,ke,press,dens,msd,rdfs

    # ----- plot radial distribution functions
    #for rdf in rdfs:
        #plt.plot(rdf[0], rdf[1])
    #plt.show()


if __name__ == '__main__':
    # put here the function that you actually want to run
    Q=1
    if Q <4:
        ensemble = 'nve'
    else:
```

```
128         ensemble = 'nvt'
129
130     Econv = []
131     if Q == 1:
132         # 1 is used for timestep optimization (1A)
133         Etrend=[]
134         for timestep in np.linspace(0.001,0.02,20):
135             E = []
136             nsteps=int(10/timestep)
137             [E, simtime, T, PE, KE, p, d, msd,rdfs] = md_run(size=3,timestep=timestep,nsteps
    =10000,temperature=300,ensemble=ensemble)
138
139             Eavg = []
140             E=np.array(E).astype(np.float)
141             for i in range(1,len(E)):
142                 print(E[(i-1):(len(E)-1)])
143                 Eavg.append(np.mean(E[(i-1):(len(E)-1)]))
144             Eavg.append(E[len(E)-1])
145
146             m = np.median(Eavg)
147             print(m)
148             s = np.std(Eavg)
149             print(s)
150             ind = next(i for i in Eavg if abs(i) < (abs(m)+2*s))
151             print(ind)
152             Etrend.append(ind)
153
154             time = np.array(simtime,dtype=float)*timestep
155
156             # plt.plot(time, T)
157             # plt.show()
158
159
160             plt.plot(time, Eavg)
161             plt.xlabel('Time (ps)')
162             plt.ylabel('Energy')
163             plt.title("Energy vs. Time")
164             plt.show()
165
166             plt.plot(time, Eavg)
167             plt.xlabel('Time of First Included Value in Average (ps)')
168             plt.ylabel('Mean Energy')
169             plt.title("Mean Energy vs. Time")
170             plt.show()
171
172
173         plt.plot(np.linspace(0.001,0.02,20), Etrend)
174         plt.xlabel('Timestep (ps)')
175         plt.ylabel('Mean Energy')
176         plt.title("Timestep Dependence of Energy")
177         plt.show()
178
179     elif Q == 2:
180         # 2 is used for temperature vs. time plots at multiple T (1B)
181         E=[]
182         T=[]
183         for temp in np.linspace(200,500,7):
184             [E, simtime, T, PE, KE, p, d, msd] = md_run(size=3, timestep=0.001, nsteps
    =10000, temperature=temp, ensemble=ensemble)
185
186             time = np.array(simtime, dtype=float) * 0.001
187
188             plt.figure(1)
189             plt.plot(time, T)
190             plt.xlabel('Time (ps)')
191             plt.ylabel('Temperature (K)')
192             plt.title("Temperature Evolution")
193
```

```
194             plt.figure(2)
195             plt.plot(time, PE)
196             plt.xlabel('Time (ps)')
197             plt.ylabel('Potential Energy')
198             plt.title("Potential Energy Evolution")
199 
200             plt.figure(3)
201             plt.plot(time, KE)
202             plt.xlabel('Time (ps)')
203             plt.ylabel('Kinetic Energy')
204             plt.title("Kinetic Energy Evolution")
205 
206         plt.figure(1)
207         plt.show()
208 
209         plt.figure(2)
210         plt.show()
211 
212         plt.figure(3)
213         plt.show()
214 
215     elif Q == 3:
216         # 3 is used for supercell dependence (1C)
217         E=[]
218         T=[]
219         Tfluc=[]
220         siz=[]
221         for size in range(3,15,1):
222             Tdiff=[]
223             #for temp in np.linspace(200, 500, 20):
224             [E, simtime, T, PE, KE, p, d, msd] = md_run(size=size, timestep=0.001, nsteps
    =5000, temperature=300, ensemble=ensemble)
225 
226             siz.append(1/(4*size**3))
227             time = np.array(simtime, dtype=float) * 0.001
228             plt.plot(time, T)
229             T = np.array(T).astype(np.float)
230 
231             m = np.median(T)
232             s = np.std(T)
233             conv = next(i for i in T if abs(i) < (abs(m) + 2 * s))
234 
235             ind = np.nonzero(T==conv)[0][0]
236             Tm=np.mean(T[ind:(len(T) - 1)])
237 
238             for i in range(ind,len(T)-1):
239                 Tdiff.append((T[i]-Tm)**2)
240             Tfluc.append(np.mean(Tdiff))
241 
242         plt.xlabel('Time (ps)')
243         plt.ylabel('Temperature (K)')
244         plt.title("Temperature Evolution")
245         plt.show()
246 
247         plt.plot(siz,Tfluc)
248         plt.xlabel('Reciprocal Size')
249         plt.ylabel('Average Temperature Fluctuation')
250         plt.title('Temperature Fluctuation vs. System Size')
251         plt.show()
252 
253     elif Q == 4:
254         # 4 is used for finding relevant T for melting
255         E = []
256         T = []
257         for temp in np.linspace(200, 1300, 12):
258             [E, simtime, T, PE, KE, p, d, msd] = md_run(size=3, timestep=0.001, nsteps
    =20000, temperature=temp, ensemble=ensemble)
259 
```

```
260             time = np.array(simtime, dtype=float) * 0.001
261
262             plt.plot(time, E)
263         plt.xlabel('Time (ps)')
264         plt.ylabel('Energy')
265         plt.title("Energy Evolution")
266         plt.show()
267
268     elif Q == 5:
269         # 5 is used to perform timestep optimization in a regime below melting
270         Etrend=[]
271         for timestep in np.linspace(0.001,0.02,20):
272             E = []
273
274             [E, simtime, T, PE, KE, p, d, msd] = md_run(size=3,timestep=timestep,nsteps
    =10000,temperature=500,ensemble=ensemble)
275
276             Eavg = []
277             E = np.array(E).astype(np.float)
278             for i in range(1, len(E)):
279                 print(E[(i - 1):(len(E) - 1)])
280                 Eavg.append(np.mean(E[(i - 1):(len(E) - 1)]))
281             Eavg.append(E[len(E) - 1])
282
283             m = np.median(Eavg)
284             print(m)
285             s = np.std(Eavg)
286             print(s)
287             ind = next(i for i in Eavg if abs(i) < (abs(m) + 2 * s))
288             print(ind)
289             Etrend.append(ind)
290
291             time = np.array(simtime, dtype=float) * timestep
292
293             # plt.plot(time, T)
294             # plt.show()
295
296             if timestep == 0.02:
297                 plt.plot(time, E)
298                 plt.xlabel('Time (ps)')
299                 plt.ylabel('Energy')
300                 plt.title("Energy vs. Time")
301                 plt.show()
302                 #
303                 # plt.plot(time, Eavg)
304                 # plt.xlabel('Time of First Included Value in Average (ps)')
305                 # plt.ylabel('Mean Energy')
306                 # plt.title("Mean Energy vs. Time")
307                 # plt.show()
308
309         plt.plot(np.linspace(0.001, 0.02, 20), Etrend)
310         plt.xlabel('Timestep (ps)')
311         plt.ylabel('Mean Energy')
312         plt.title("Timestep Dependence of Energy")
313         plt.show()
314
315     elif Q == 6:
316         # 6 is used to perform supercell optimization in a regime below melting
317         E = []
318         T = []
319         for size in range(3, 13, 1):
320             [E, simtime, T, PE, KE, p, d, msd] = md_run(size=size, timestep=0.001, nsteps
    =5000, temperature=500, ensemble=ensemble)
321
322             time = np.array(simtime, dtype=float) * 0.001
323             plt.plot(time, T)
324
325         plt.xlabel('Time (ps)')
```

```
        plt.ylabel('Temperature (K)')
        plt.title("Temperature Evolution")
        plt.show()

    elif Q == 7:
        # 7 is used for the NVT ensemble with the results from above
        E=[]
        for temp in np.linspace(1050,1150,6):
            [E, simtime, T, PE, KE, p, d, msd,rdfs] = md_run(size = 7, timestep = 0.001,
    nsteps = 10000, temperature = temp, ensemble = ensemble)
            print(simtime)
            time = np.array(simtime, dtype=float) * 0.001

            plt.figure(1)
            plt.plot(time, E)
            plt.xlabel('Time (ps)')
            plt.ylabel('Energy')
            plt.title("Energy Evolution")

            plt.figure(2)
            plt.plot(time, msd)
            plt.xlabel('Time (ps)')
            plt.ylabel('Mean Square Density')
            plt.title("Mean Square Density Evolution")

            plt.figure(3)
            rdfs = np.array(rdfs).astype(np.float)
            rdfm=[]
            print(rdfs.shape)
            s = rdfs.shape[0]
            for i in range(0,rdfs.shape[2]):
                sum=0
                for j in range(0,s):
                    sum = sum + rdfs[j][1][i]
                rdfm.append(sum/s)
            plt.plot(rdfs[0][0], rdfm)
            plt.title("Radial Density Function")

        plt.figure(1)
        plt.show()

        plt.figure(2)
        plt.show()

        plt.figure(3)
        plt.show()
```