NB: The graded, first version of the report must be returned if you hand in a second time!

H1b: Classical scattering by a central potential

Gustav Hjelmare and Albin Karlsson

November 21, 2014

Task №	Points	Avail. points
Σ		

Introduction

How the world works is a question that many strive to answer, and in order to do that we look at the world both at large and small scales. Here we try to describe and evaluate the properties and behavior of aluminium over short times and distances.

Task 1

In implementing our model we have used a lattice spacing of 4.05 Å (aluminium at room temperature and atmospheric pressure[1]). Before deciding on a suitable timestep, we also implemented the equilibration (Task 2). Thus, the energy data used for determining a suitable timestep is taken from the equilibration process. Plots of energy as a function of time for a few different timesteps can be seen in figure 1.

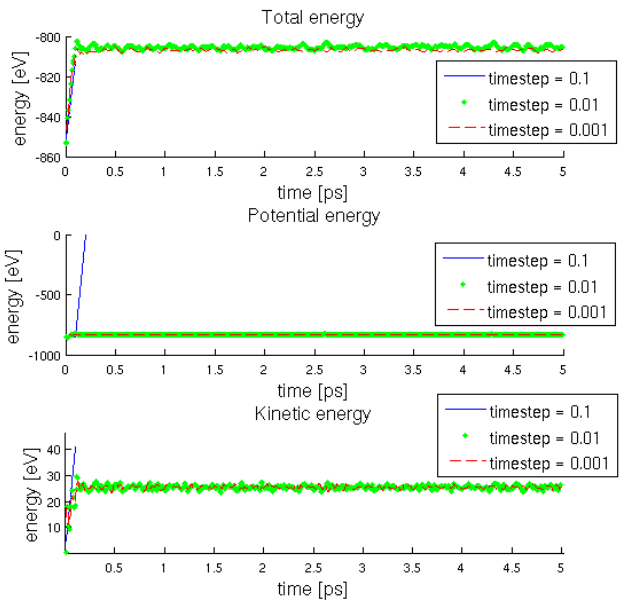

Figure 1: The figure displays how the total, kinetic and potential energy in the system behaves over time, for three different timesteps in the computational model. A timestep that is too large (0.1 ps) makes the model diverge, while a timestep that is too short increases computation time while giving no further information.

The figure shows that 0.01 ps is a suitable timestep. This is what we used for the remainder of the tasks.

Task 2

During equilibration the energy of the system changes, in order to bring the temperature and pressure to their desired values. We found that an equilibration time of 5 ps (corresponding to 500 timesteps at 0.01 ps each) is sufficient to bring the system to the desired state, using time constants $\tau_P = 0.05$ and $\tau_T = 0.02$. The change in total energy is visible in the early timesteps of figure 1.

Task 3 and 4

Figures 2 and 3 show the time evolution of the pressure and temperature. The desired pressure in these runs was $6.32420934 \cdot 10^{-7}$ eV/Å^3 (corresponding to 1 atm) and the desired temperature was 500+273=773 K in fig. 2 and 700+273=973 K in fig. 3. As can be seen in the figures, the pressure and temperature of the system stabilize around

the desired values. the temperature in figure 2 (after equilibration) has a mean value of 772.95 K and the pressure $4.62 \cdot 10^{-7}$ eV/Å^3 while the mean temperature and pressure in figure 3 are 972.8 K and $1.6447 \cdot 10^{-6}$ eV/Å^3.

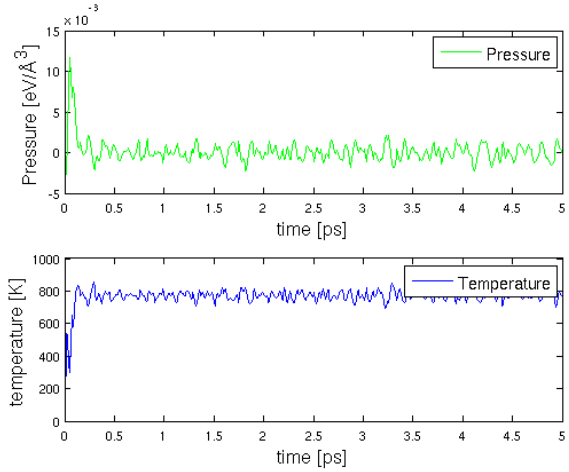

Figure 2: Pressure and temperature during equilibration, showing that the system is stabilized around the temperature 773K and the pressure $6.32420934 \cdot 10^{-7}$ eV/Å^3.

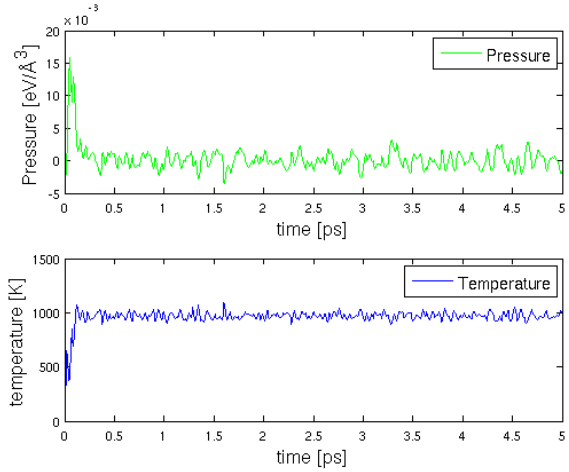

Figure 3: Pressure and temperature during equilibration, showing that the system is stabilized around the temperature 973K and the pressure $6.32420934 \cdot 10^{-7}$ eV/Å^3.

From the task description and other sources[1] it seems like our material is supposed to undergo phase transition into a liquid when taking it to 700 degrees Celcius, but looking at the trajectory of a particle over time (figure 4), this doesn't seem to be the case. If the system is a solid the trajectory is expected to remain in some bounded volume while the trajectory of a particle in a liquid is free to move as it pleases. Figure 4 shows the trajectories of one particle in the system at $T = 500$ degrees Celsius (773 K, blue line) and $T = 700$ degrees Celsius (973 K, green line) which obviously both remain bounded. In order to liquify the system, we raised the temperature to $T = 900$ degrees Celsius (1172 K, red line), which clearly shows a non-bounded behaviour in figure 4).

For the remaining tasks we will use the temperature 1173 K in addition to the specified 773 K in order to liquify our system.

The calculated statistical inefficiency constants are shown in table 1.

What can be seen in the table is that the error of the mean values in our result seem to get larger with larger temperature.

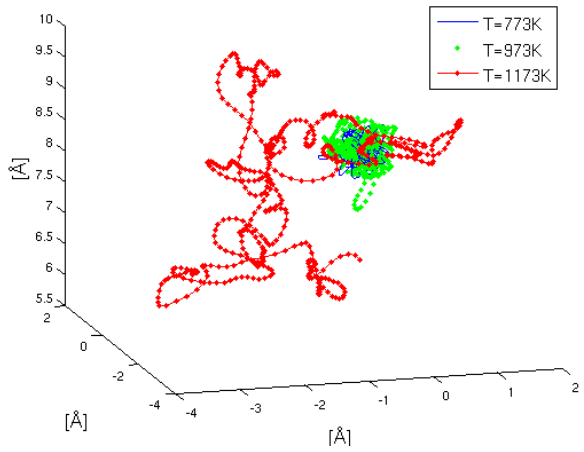

Figure 4: The figure shows the trajectories of one particle in the system at different temperatures. The displacement of the particle at 973 K is larger than the one at 773 K as can be expected, but remains bounded, which is not consistent with our expectations. The trajectory at 1173 K moves away from it's initial position, which is consistent with being in a liquid state.

Table 1: Statistical inefficiency parameters and the errors of our mean value for the temperature, at different temperatures.

	s_T	s_P	**Var**$[I_T]$	**Var**$[I_P]$
T=773 K	4.2	3.5	6.8	$4.5 \cdot 10^{-9}$
T=973K	3.7	6.1	10.0	$1.8 \cdot 10^{-8}$
T=1173	4.5	20	16.7	$6.5 \cdot 10^{-8}$

Task 5

The mean-squared displacement is shown in figure 5. It is clear that the two lower temperatures result in a bounded mean-squared displacement which confirms that they both are solids, while the highest temperature results in a diverging MSD and therefore should be a liquid. It is also reasonable that the mean-squared displacement is higher at T = 973 K than at T = 773 K because of more intense vibrations, which can also be seen to some degree in figure 4

The self-diffusion coefficient in the case of the liquid was calculated (using the mean-squared displacement) to be $8.006689 \cdot 10^{-1}$ which is consistent with the self-diffusion coefficient that is calculated using the velocity correlation function in Task 6.

Task 6

The velocity correlation function $\Phi(\Delta t)$ describes, on average, how similar the velocity at time t is to velocities at related times $t + \Delta t$ as t takes all possible values. Since we have identical particles, we use an average over all of them, shown in figure 6. It is clear that for small Δt, velocities are highly correlated, i.e. a particle is unlikely to change velocity completely in a short time span. As Δt grows, the correlation becomes negative, and then positive again after a similar additional time. This is likely due to lattice vibrations, causing many particles to periodically reverse direction after some characteristic time.

The cosine-transform of the velocity correlation function, shown in figure 7, describes the frequency content of the velocity correlation function. The small short-period perturbations are likely an artifact of the simulation, but the fairly clear peaks at 0.3 rad/ps 0.55 rad/ps correspond well with lattice vibration frequencies in other

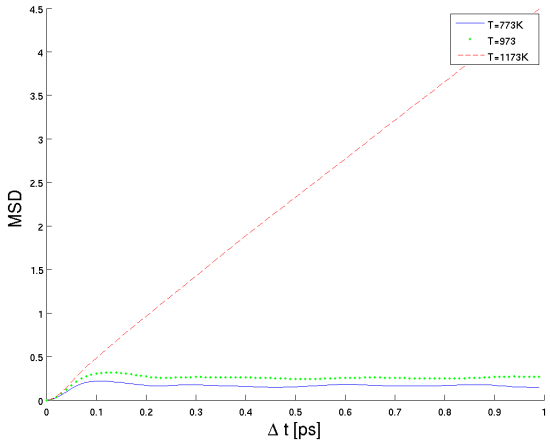

Figure 5: Mean-squared displacement at two different temperatures. At the highest temperature the system is a liquid, as the position appears to diverge over time, while the other two temperatures have bounded positions.

sources[2].

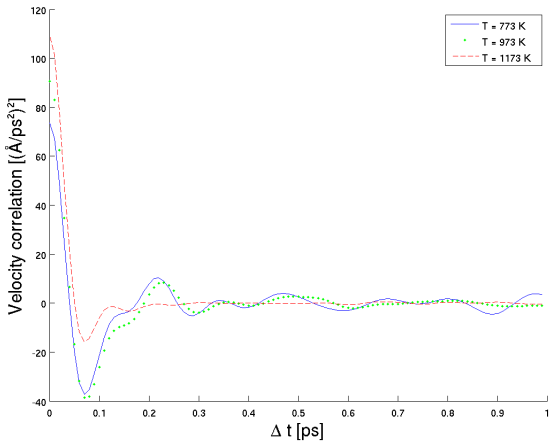

Figure 6: The velocity correlation function, averaged over all particles. For short times there is a significant contribution from the back-and-forth vibration of atoms in the lattice, but as time passes these vibrations get perturbed into other directions and the correlation decays.

When calculating the self-diffusion coefficient, using the velocity correlation we got the result $9.883325 \cdot 10^{-1}$ which is very close to the diffusion coefficient one got using the mean-squared displacement (as it should).

References

[1] C. Nordling, J. Österman (2006) *Physics Handbook*, 8 ed.

[2] P. A. Flinn, G. M. McManus (1963) *Lattice Vibrations and Debye Temperatures of Aluminum*. http://journals.aps.org/pr/pdf/10.1103/PhysRev.132.2458. (21-11-2014).

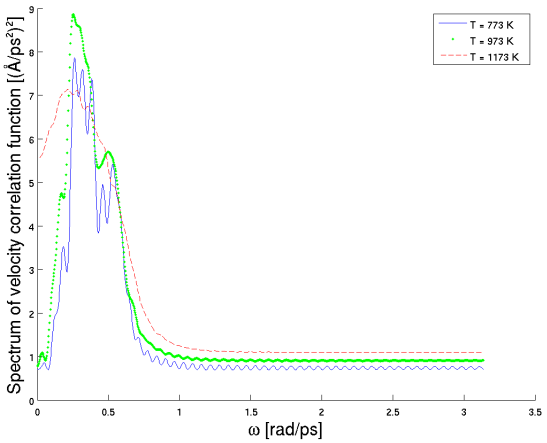

Figure 7: The frequency spectrum of the velocity correlation function. While the data has a significant short-period perturbation, there are also fairly clear peaks around 0.3 rad/ps and 0.55 rad/ps. As can be seen, the highest temperature starts out at rather high value at low frequencys, this means that the velocity is correlated for a long time, which can be expected of a liquid, which is consistent with figure 5 showing a steadily increasing MSD, while the two lower temperatures have no zero-frequency contribution and a limited MSD in figure 5.

A Source Code

A.1 Main file: `MD_main.c`

```c
// MD_main.c

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "initfcc.h"
#include "alpotential.h"
#include "MD_functions.h"

#define PI 3.14159265368979

/* Main program */
int main()
{
  // REMEMBER: \m/ METAL UNITS \m/
  // simulation settings
  double productionTime = 5;
  double equilibrationTime = 5;
  double timestep = 0.01;
//  timestep = 0.1;
//  timestep = 0.001;

  double msdAverageSteps = 50;

  int nSpectrumPoints = 1000;
  double spectrumInterval = PI;

  double maxCorrelationTime = 1;

  double timeConstantT = 0.02;
  double timeConstantP = 0.05;

  // physical parameters
  int dim = 3;    // do not change
  int nCells = 4;
  int nParticles = 4*pow(nCells,dim);
  double wantedTemperature = 500+273;   // Target temperature
//  wantedTemperature = 700+273;
//  wantedTemperature = 900+273;
  double wantedPressure =  6.32420934* 0.0000001; // 1 atm in metal units
  double mass = 0.00279636;  // 26.9815 u * 1.0364 * 0.0001
  double latticeParameter = 4.05;
  double maxDeviation = 0.05;

```

```
  // derived quantities
  double nEquilibrationSteps = equilibrationTime/timestep;
  double nProductionSteps = productionTime/timestep;
  int maxCorrelationSteps = maxCorrelationTime/timestep;


  // storage of physical quantities
  double pos[nParticles][dim];
  double vel[nParticles][dim];
  double force[nParticles][dim];
  double savedValuesT[maxCorrelationSteps];
  double meanValuesT[maxCorrelationSteps];
  double savedValuesP[maxCorrelationSteps];
  double meanValuesP[maxCorrelationSteps];
  double energy, potentialEnergy, kineticEnergy;
  double siquare_root_of_alpha;
  double alphaT = 1;
  double alphaP = 1;
  double currentTemperature;
  double currentPressure;
  double sqrtAlphaT;
  double curtAlphaP;
  double virial;
  double volume;
  double nAverageSteps;
  double meanTemperature, meanSquareTemperature;
  double meanTemperature_ik[maxCorrelationSteps-1];
  double meanPressure, meanSquarePressure;
  double meanPressure_ik[maxCorrelationSteps-1];
  double phiTemperature[maxCorrelationSteps];
  double phiPressure[maxCorrelationSteps];
  double meanTemperatureSquare, meanPressureSquare;
  double varTemperature, varPressure;
  double sTemperature, sPressure;
  double savedPos[maxCorrelationSteps][nParticles][dim];
  double savedVelocities[maxCorrelationSteps][nParticles][dim];
  double meanVelocityScalar[maxCorrelationSteps][nParticles];
  double meanVelocityAverage[maxCorrelationSteps];
  double spectrum[nSpectrumPoints];
  double diffusionCoefficient;
  double meanDistanceScalar[maxCorrelationSteps][nParticles];
  double meanDistanceAverage[maxCorrelationSteps];
  double msdDiffCoeff = 0;
  int MTemperature, MPressure;

  // other stuff
  int i,j,k,m;

  srand(time(NULL));
  double random_value;

// End of variable declarations, start of actual code

  printf("Initializing");
  init_fcc(pos, nCells, latticeParameter);

  // Applying random perturbations from strict fcc positions
  for(i = 0; i<nParticles; i++){
    for (j = 0; j<3; j++){
      random_value = (double) rand() / (double) RAND_MAX;
      pos[i][j] = pos[i][j] + (random_value-0.5) * maxDeviation * \
      latticeParameter;
    }
  }

  // Initialize velocities and forces to zero
  for (i = 0; i<nParticles; i++){
    for (j = 0; j<3; j++){
      vel[i][j] = 0;
      force[i][j] = 0;
    }
  }

  // Calculating initial energies
  potentialEnergy = get_energy_AL(pos, nCells*latticeParameter, nParticles);
  kineticEnergy = GetKineticEnergy(vel, mass, nParticles);
  energy = potentialEnergy + kineticEnergy;


  // Saving initial energies on the first line in the file
  FILE *energyFile;
  energyFile = fopen("energy.data","w");
  fprintf(energyFile, "%e \t %e \t %e \t %e \n", 0.0, energy, \
  potentialEnergy, kineticEnergy);

  FILE *ptFile;
  ptFile = fopen("pt.data","w");

  FILE *positionFile;
  positionFile = fopen("position.data","w");

```

```
// Initialization is done, moving on to equilibration

  printf("\t"); // Progress indicator stuff
  for (i = 0; i < ( nEquilibrationSteps > nProductionSteps  ? \
  nEquilibrationSteps:nProductionSteps )/100; i++) {
      printf(".");
  }
  printf("\tDone!\nEquilibration\t");

  for (i=0;i<nEquilibrationSteps;i++) {   // Start of equilibration loop
    // Update velocities and positions
    for (j=0; j<nParticles; j++) {
      for (k = 0; k<dim; k++) {
        vel[j][k] = vel[j][k] + 0.5*timestep*force[j][k] / mass;
        pos[j][k] = pos[j][k] + timestep*vel[j][k];
      }
    }

    // Calculate forces so that we can take the next half step
    get_forces_AL(force, pos, nCells*latticeParameter, nParticles);

    // Update velocities again
    for (j=0; j<nParticles; j++) {
      for (k=0; k<dim; k++) {
        vel[j][k] = vel[j][k] + 0.5*timestep*force[j][k] / mass;
      }
    }

    // Calculating energies
    potentialEnergy = get_energy_AL(pos, nCells*latticeParameter, \
    nParticles);
    kineticEnergy = GetKineticEnergy(vel, mass, nParticles);
    energy = potentialEnergy + kineticEnergy;
    fprintf(energyFile,"%e \t %e \t %e \t %e \n", (i+1)*timestep, energy, \
    potentialEnergy, kineticEnergy);

    // Get temperature and pressure
    currentTemperature = GetInstantTemperature(vel, nParticles, mass, dim);
    volume = pow(nCells*latticeParameter, 3);
    virial = get_virial_AL(pos, nCells*latticeParameter, nParticles);
    currentPressure = GetPressure(currentTemperature, volume, virial, \
    nParticles);
    fprintf(ptFile, "%e \t %e \t %e \n", (i+1)*timestep, currentTemperature, \
    currentPressure);

    // Calculate scaling parameters for equilibration
    alphaT = GetAlphaT(wantedTemperature, currentTemperature, timestep, \
    timeConstantT);
    alphaP = GetAlphaP(wantedPressure, currentPressure, timestep, \
    timeConstantP);
    sqrtAlphaT = sqrt(alphaT);
    curtAlphaP = pow(alphaP, 1.0 / 3);

    // Rescale velocity and position
    latticeParameter = latticeParameter * curtAlphaP;
    for (j=0; j<nParticles; j++) {
      for (k=0; k<dim; k++) {
        vel[j][k] = vel[j][k] * sqrtAlphaT;
        pos[j][k] = pos[j][k] * curtAlphaP;
      }
    }
    if( i % 100 == 0) {   // Progress indicator
      printf("I");
      fflush(stdout);
    }
  } // End of equilibration loop

// End of equilibration, start of production

  printf("\tDone!\nProduction\t");

  for (i=0;i<nProductionSteps;i++) {
    // Save positions of one particle to check whether the
    // aluminium is solid or liquid
    fprintf(positionFile, "%d \t %e \t %e \t %e \n", i, \
    pos[0][0], pos[0][1], pos[0][2]);

    // Update velocities and positions
    for (j=0; j<nParticles; j++) {
      for (k = 0; k<dim; k++) {
        vel[j][k] = vel[j][k] + 0.5*timestep*force[j][k] / mass;
        pos[j][k] = pos[j][k] + timestep*vel[j][k];
      }
    }

    // Calculate forces so that we can take the next half step
    get_forces_AL(force, pos, nCells*latticeParameter, nParticles);

    // Update velocities again
    for (j=0; j<nParticles; j++) {
      for (k=0; k<dim; k++) {
```

```
        vel[j][k] = vel[j][k] + 0.5*timestep*force[j][k] / mass;
      }
    }

    potentialEnergy = get_energy_AL(pos, nCells*latticeParameter, \
    nParticles);
    kineticEnergy = GetKineticEnergy(vel, mass, nParticles);
    energy = potentialEnergy + kineticEnergy;

    currentTemperature = GetInstantTemperature(vel, nParticles, mass, dim);
    volume = pow(nCells*latticeParameter, 3);
    virial = get_virial_AL(pos, nCells*latticeParameter, nParticles);
    currentPressure = GetPressure(currentTemperature, volume, virial, \
    nParticles);

    //Saves temp and pressure values in order to calculate s.
    if (i < maxCorrelationSteps) {
      savedValuesT[i] = currentTemperature;
      savedValuesP[i] = currentPressure;
    } else {
      // updates the saved values.
      for (j = 0; j < maxCorrelationSteps - 1; j++){
        savedValuesT[j] = savedValuesT[j+1];
        savedValuesP[j] = savedValuesP[j+1];
      }
      savedValuesT[maxCorrelationSteps - 1] = currentTemperature;
      savedValuesP[maxCorrelationSteps - 1] = currentPressure;

      // calculates and saves f_i*f_k
      for (j = 0; j<maxCorrelationSteps; j++){
        meanTemperature_ik[j] += savedValuesT[maxCorrelationSteps - 1] \
        * savedValuesT[maxCorrelationSteps - 1 - j];
        meanPressure_ik[j] += savedValuesP[maxCorrelationSteps - 1] \
        * savedValuesP[maxCorrelationSteps - 1 - j];
      }
      meanTemperature += currentTemperature;
      meanPressure += currentPressure;
      meanSquareTemperature += currentTemperature * currentTemperature;
      meanSquarePressure += currentPressure * currentPressure;
    }

    // Saving data for the velocity correlation function
    if (i < maxCorrelationSteps) {
      for (m = 0; m < nParticles ; m++) {
        savedVelocities[i][m][0] = vel[m][0];
        savedVelocities[i][m][1] = vel[m][1];
        savedVelocities[i][m][2] = vel[m][2];
      }
    } else {
      for (j = 0; j< maxCorrelationSteps- 1; j++) {
        for (m = 0; m<nParticles;  m++) {
          savedVelocities[j][m][0] = savedVelocities[j+1][m][0];
          savedVelocities[j][m][1] = savedVelocities[j+1][m][1];
          savedVelocities[j][m][2] = savedVelocities[j+1][m][2];
        }
      }
      for (m = 0; m<nParticles; m++) {
        savedVelocities[maxCorrelationSteps - 1][m][0] = vel[m][0];
        savedVelocities[maxCorrelationSteps - 1][m][1] = vel[m][1];
        savedVelocities[maxCorrelationSteps - 1][m][2] = vel[m][2];
      }

      for (j = 0; j<maxCorrelationSteps; j++) {
        for (m = 0; m < nParticles; m++) {
          meanVelocityScalar[j][m] += ScalarProduct( \
          savedVelocities[0][m], savedVelocities[j][m]);
        }
      }
    }

    if(i <  maxCorrelationSteps) {
      for(j = 0; j<nParticles; j++) {
        for (k = 0; k<dim; k++) {
          savedPos[i][j][k] = pos[j][k];
        }
      }
    }else {
      for (j = 0; j<maxCorrelationSteps-1; j++) {
        for (k = 0; k<nParticles; k++) {
          for (m = 0; m<dim; m++) {
            savedPos[j][k][m] = savedPos[j+1][k][m];
          }
        }
      }
      for (j = 0; j<nParticles; j++) {
        for (k=0; k<dim; k++) {
          savedPos[maxCorrelationSteps-1][j][k] = pos[j][k];
        }
      }
      for(j = 0; j<maxCorrelationSteps; j++){
        for(k = 0; k<nParticles; k++){
```

```
          meanDistanceScalar[j][k] += getDistanceSquared( \
          savedPos[0][k], savedPos[j][k]);
        }
      }
    }


    if(i%100 == 0) {  // Progress indicator
      printf("I");
      fflush(stdout);
    }

  } // End of production loop

  printf("\tDone!\nCleaning up\n");

// The production part of the simulation is now done.
// The code below forms some averages and other stuff that
// is best done with all data on hand

  FILE *valuesFile;   // For saving various values (non-array data)
  valuesFile = fopen("values.data","w");

  nAverageSteps = nProductionSteps-maxCorrelationSteps;
      // take averages over this many steps

  FILE *msdFile;
  msdFile = fopen("msd3.data","w");

  for( i = 0; i < maxCorrelationSteps; i++) {
    meanDistanceAverage[i] = 0;
    for (j = 0; j<nParticles; j++) {
      meanDistanceScalar[i][j] = meanDistanceScalar[i][j]/nAverageSteps;
      meanDistanceAverage[i] += meanDistanceScalar[i][j];
    }
    meanDistanceAverage[i] = meanDistanceAverage[i]/nParticles;
    fprintf(msdFile, "%e\t%e\n", i*timestep, meanDistanceAverage[i]);
  }

  // Diffusion coeff from MSD
  for (i = maxCorrelationSteps - msdAverageSteps; \
      i < maxCorrelationSteps; i++) {
    msdDiffCoeff += 1.0/(6*i*timestep)*meanDistanceAverage[i];
  }
  msdDiffCoeff = msdDiffCoeff / msdAverageSteps;
  fprintf(valuesFile,"%e\tDiffusion coeff (MSD)\n", msdDiffCoeff);

  // Final processing and saving of velocity correlation
  // function and diffusion coeff from time integral
  FILE *velcorFile;
  velcorFile = fopen("velcor.data","w");

  diffusionCoefficient = 0;

  for( i = 0; i < maxCorrelationSteps; i++) {
    meanVelocityAverage[i] = 0;
    for (j = 0; j<nParticles; j++) {
      meanVelocityScalar[i][j] = meanVelocityScalar[i][j]/nAverageSteps;
      meanVelocityAverage[i] += meanVelocityScalar[i][j];
    }
    meanVelocityAverage[i] = meanVelocityAverage[i]/nParticles;
    fprintf(velcorFile, "%e\t%e\n", i*timestep, meanVelocityAverage[i]);

    diffusionCoefficient += meanVelocityAverage[i];
  }

  diffusionCoefficient = (1.0/3.0) * \
  diffusionCoefficient / maxCorrelationSteps;
  fprintf(valuesFile,"%e\tDiffusion coeff (time integral)\n", \
  diffusionCoefficient);

  // Calculating and saving spectrum integral thingy
  FILE *spectrumFile;
  spectrumFile = fopen("spectrum.data","w");

  for( i = 0; i < nSpectrumPoints; i++) {
    spectrum[i] = 0;
    for( j = 0; j < maxCorrelationSteps; j++) {
      spectrum[i] += meanVelocityAverage[j] * cos(spectrumInterval * \
      i * j / nSpectrumPoints);
    }
    spectrum[i] = 2 * spectrum[i]/maxCorrelationSteps;
    fprintf(spectrumFile,"%e \t %e \n",i*spectrumInterval/nSpectrumPoints, \
    spectrum[i]);
  }

  // Setting up for calculating correlation data for temperature and pressure
  meanTemperature = meanTemperature/nAverageSteps;
  meanSquareTemperature = meanSquareTemperature/nAverageSteps;
  meanPressure = meanPressure/nAverageSteps;
  meanSquarePressure = meanSquarePressure/nAverageSteps;
```

```
  meanTemperatureSquare = meanTemperature*meanTemperature;
  meanPressureSquare = meanPressure*meanPressure;

  for(i = 0; i < maxCorrelationSteps+1; i++){
    meanTemperature_ik[i] = meanTemperature_ik[i]/nAverageSteps;
    meanPressure_ik[i] = meanPressure_ik[i]/nAverageSteps;
  }


  // Correlation data for temperature and pressure
  FILE *phiTFile;
  phiTFile = fopen("phiTemperature.data","w");

  FILE *phiPFile;
  phiPFile = fopen("phiPressure.data","w");

  for(i = 0; i<maxCorrelationSteps; i++){
    phiTemperature[i] = (meanTemperature_ik[i] - meanTemperatureSquare)/ \
    (meanSquareTemperature - meanTemperatureSquare);
    phiPressure[i] = (meanPressure_ik[i] - meanPressureSquare)/ \
    (meanSquarePressure - meanPressureSquare);
    fprintf(phiTFile, "%e \t %e \n", (i+1)*timestep, phiTemperature[i]);
    fprintf(phiPFile, "%e \t %e \n", (i+1)*timestep, phiPressure[i]);
  }

  // Finding M (the longest time for which correlation is above some limit)
  double limit = exp(-2.0);
  i = 0;
  while( (phiTemperature[i] > limit) && (i < maxCorrelationSteps) ) {
    i++;
  }
  MTemperature = i;

  i = 0;
  while( (phiPressure[i] > limit) && (i < maxCorrelationSteps) ) {
    i++;
  }
  MPressure = i;

  //Sum over phi to get statistical inefficiency for temperature
  for(i = 0; i<MTemperature; i++){
    sTemperature += phiTemperature[i];
  }
  sTemperature = sTemperature*2;
  // phi is symmetric and we sum from 0 to M instead of from -M to M

  //Sum over phi to get statistical inefficiency for pressure
  for(i = 0; i<MPressure; i++){
    sPressure += phiPressure[i];
  }
  sPressure = sPressure*2;
  // phi is symmetric and we sum from 0 to M instead of from -M to M

  // Calculating variances
  varTemperature = meanSquareTemperature - meanTemperatureSquare;
  varPressure = meanSquarePressure - meanPressureSquare;

  varTemperature = varTemperature*sTemperature/nProductionSteps;
  varPressure = varPressure*sPressure/nProductionSteps;
  fprintf(valuesFile,"%e\tsTemperature\n", sTemperature);
  fprintf(valuesFile,"%e\tsPressure\n", sPressure);
  fprintf(valuesFile,"%e\tvarTemperature\n", varTemperature);
  fprintf(valuesFile,"%e\tvarPressure\n",varPressure);

  FILE *donefile;   // Just an empty file to indicate completion
  donefile = fopen("done.data", "w"); // useful when using ssh+screen
  fprintf(donefile, "done");

  close(energyFile);
  close(ptFile);
  close(positionFile);
  close(valuesFile);
  close(msdFile);
  close(velcorFile);
  close(spectrumFile);
  close(phiTFile);
  close(phiPFile);
  close(donefile);

  printf("Done!\n");
  return 0;
}
```

A.2 Functions header: `MD_functions.h`

```
#ifndef _MD_functions_h
#define _MD_functions_h

extern double getDistanceSquared(double *, double *);
extern double ScalarProduct(double *, double *);
extern double GetPressure(double, double, double, int);
extern double GetInstantTemperature(double [][3], int, double, int);
extern double GetKineticEnergy(double [][3], double, int);
extern double GetAlphaT(double, double, double, double);
extern double GetAlphaP(double, double, double, double);


#endif
```

A.3 Functions code: `MD_functions.c`

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define BOLTZMANN 8.6173324*0.00001 // in eV/K


double getDistanceSquared(double vec1[3], double vec2[3])
{
  double sum = 0;
  double temporary;
  int i;


  for(i = 0; i<3; i++){
    temporary = vec1[i] - vec2[i];
    sum += pow(temporary,2);
  }
  return sum;
}

double ScalarProduct(double vec1[3], double vec2[3])
{
  double product = 0;
  int i;

  for(i = 0 ; i < 3 ; i++) {
    product += vec1[i]*vec2[i];
  }

  return product;
}


double GetPressure(double temperature, double volume, double virial, \
int nParticles)
{
  double pressure = 0;

  pressure = (nParticles * BOLTZMANN * temperature + virial)/volume;

  return pressure;
}

double GetInstantTemperature(double vel [][3], int nParticles, \
double mass, int dim)


{
  double temperature = 0;
  int i,j;

  double factor = 2/(3*nParticles*BOLTZMANN);

  for (i=0;i<nParticles;i++) {
    for (j=0;j<dim;j++) {
      temperature += factor * pow(vel[i][j],2)*mass/2;
    }
  }

  return temperature;
}

// Calculates the kinetic energy
double GetKineticEnergy(double vel[][3], double mass, int nParticles)
{
  int i,j;
  double sum;
  double energy = 0;

```

```
  for(i = 0; i<nParticles; i++) {
    sum = 0;
    for(j = 0; j<3; j++) {
      sum +=vel[i][j]*vel[i][j];
    }
    energy += sum*mass/2;
  }
  return energy;
}

// Calculate alphaT (the modifier to get the right temperature)
double GetAlphaT(double wantedTemp, double currentTemp, \
                         double timestep, double timeConstant)
{
  double alpha;
  alpha = 1 + timestep/timeConstant * (wantedTemp-currentTemp)/currentTemp;

  return alpha;
}



// Calculate alphaP (the modifier to get the right pressure)
double GetAlphaP(double wantedPressure, double currentPressure, \
                           double timestep, double timeConstant)
{
  double kappa = 2.21901454; // \AA^3/eV (at 300K)
  double alpha;

  alpha = 1 - kappa*timestep/timeConstant* \
        (wantedPressure - currentPressure);

  return alpha;
}
```

A.4 Plotting in matlab: `plot_results.m`

```
% in some places, data for several temperatures is required
% so you need to run the main program several times with
% different temperature settings and manually rename data files

clear all
clc

data = importdata('energyT1.data');

time1 = data(:,1);
energy1 = data(:,2);
potentialEnergy1 = data(:,3);
kineticEnergy1 = data(:,4);

data = importdata('energyT2.data');
time2 = data(:,1);
energy2 = data(:,2);
potentialEnergy2 = data(:,3);
kineticEnergy2 = data(:,4);

data = importdata('energyT3.data');
time3 = data(:,1);
energy3 = data(:,2);
potentialEnergy3 = data(:,3);
kineticEnergy3 = data(:,4);



%%
textStorlek = 14;
legendStorlek = 11;
subplot(3,1,1)
hold on
plot(time1, energy1, 'b');
plot(time2, energy2, 'g.');
plot(time3, energy3, 'r--');
text =legend('timestep = 0.1', 'timestep = 0.01', 'timestep = 0.001');
set(text, 'FontSize', legendStorlek);
title('Total energy', 'FontSize',textStorlek)
ylabel('energy [eV]', 'FontSize', textStorlek);
xlabel('time [ps]', 'FontSize',textStorlek);
subplot(3,1,2)
hold on
plot(time1, potentialEnergy1, 'b');
plot(time2, potentialEnergy2, 'g.');
plot(time3, potentialEnergy3, 'r--');
text = legend('timestep = 0.1', 'timestep = 0.01', 'timestep = 0.001');
set(text, 'FontSize', legendStorlek);
title('Potential energy', 'FontSize',textStorlek);
```

```
ylabel('energy [eV]', 'FontSize',textStorlek);
xlabel('time [ps]', 'FontSize',textStorlek);
subplot(3,1,3)
hold on
plot(time1, kineticEnergy1, 'b');
plot(time2, kineticEnergy2, 'g.');
plot(time3, kineticEnergy3, 'r--');
text = legend('timestep = 0.1', 'timestep = 0.01', 'timestep = 0.001');
set(text, 'FontSize', legendStorlek);
title('Kinetic energy', 'FontSize',textStorlek);
ylabel('energy [eV]', 'FontSize',textStorlek);
xlabel('time [ps]', 'FontSize',textStorlek);
%% Equilibration - T & P
data = importdata('pt500.data');
temp500 = data(:,2);
pressure500 = data(:,3);

data = importdata('pt700.data');
temp700 = data(:,2);
pressure700 = data(:,3);

textStorlek = 14;
legendStorlek = 11;


hold on
subplot(2,1,1);
plot(data(:,1), pressure500, 'g');
text  = legend('Pressure');
set(text, 'FontSize', legendStorlek);
ylabel('Pressure [eV/  ^{3}]', 'FontSize',textStorlek);
xlabel('time [ps]', 'FontSize',textStorlek);

hold on

subplot(2,1,2);
plot(data(:,1), temp500, 'b');
text = legend('Temperature');
set(text, 'FontSize', legendStorlek);
ylabel('temperature [K]', 'FontSize',textStorlek);
xlabel('time [ps]', 'FontSize',textStorlek);

figure
hold on
subplot(2,1,1);
plot(data(:,1), pressure700, 'g');
text  = legend('Pressure');
set(text, 'FontSize', legendStorlek);
ylabel('Pressure [eV/  ^{3}]', 'FontSize',textStorlek);
xlabel('time [ps]', 'FontSize',textStorlek);

hold on

subplot(2,1,2);
plot(data(:,1), temp700, 'b');
text = legend('Temperature');
set(text, 'FontSize', legendStorlek);
ylabel('temperature [K]', 'FontSize',textStorlek);
xlabel('time [ps]', 'FontSize',textStorlek);

useFrom = 0.6;
start = fix(length(energy)*useFrom);

meanTemp = mean(temp(start:end))
meanPress = mean(pressure(start:end))

%% Velocity correlation function

clear all
clc
clf

textStorlek = 14;
legendStorlek = 11;

data_500 = importdata('velcor_500.data');
data_700 = importdata('velcor_700.data');
data_900 = importdata('velcor_900.data');

hold on
plot(data_500(:,1),data_500(:,2),'b')
plot(data_700(:,1),data_700(:,2),'g.')
plot(data_900(:,1),data_900(:,2),'r--')

ylabel('Velocity correlation [(  /ps^2)^2]','FontSize',textStorlek)
xlabel('\Delta t [ps]','FontSize',textStorlek)

text = legend('T = 773 K', 'T = 973 K', 'T = 1173 K');
set(text, 'FontSize', legendStorlek);

saveas(gcf,'velcor.png','png')
```

```
%% Spectrum analysis

clear all
clc
clf

textStorlek = 14;
legendStorlek = 11;

data_500 = importdata('spectrum_500.data');
data_700 = importdata('spectrum_700.data');
data_900 = importdata('spectrum_900.data');

hold on
plot(data_500(:,1),data_500(:,2),'b')
plot(data_700(:,1),data_700(:,2),'g.')
plot(data_900(:,1),data_900(:,2),'r--')

ylabel('Spectrum of velocity correlation function [(  /ps^2)^2]', ...
    'FontSize',textStorlek)
xlabel('\omega [rad/ps]','FontSize',textStorlek)

text = legend('T = 773 K', 'T = 973 K', 'T = 1173 K');
set(text, 'FontSize', legendStorlek);

saveas(gcf,'spectrum.png','png')

%% MSD function

clear all
clc
clf

data = importdata('msd.data');
data2 = importdata('msd2.data');
data3 = importdata('msd3.data');

textStorlek = 14;
legendStorlek = 11;

hold on
plot(data(:,1),data(:,2), 'b');
plot(data3(:,1),data3(:,2),'g.');
plot(data2(:,1),data2(:,2),'r--');



ylabel('MSD ', 'FontSize',textStorlek)
xlabel('\Delta t [ps]', 'FontSize',textStorlek)
text = legend('T=773K', 'T=973', 'T=1173K');
set(text, 'FontSize', legendStorlek);

saveas(gcf,'MSD.png','png')

%% Plot positions (3D-plot) of one particle
clear all
clc
pos1 = importdata('position1.data');
pos2 = importdata('position2.data');
pos3 = importdata('position3.data');

textStorlek = 14;
legendStorlek = 11;

hold on
plot3(pos1(:,2), pos1(:,3),pos1(:,4),'b');
plot3(pos2(:,2), pos2(:,3),pos2(:,4),'g.');
plot3(pos3(:,2), pos3(:,3),pos3(:,4),'r.-');
text = legend('T=773K', 'T=973K', 'T=1173K');
set(text, 'FontSize', legendStorlek);

xlabel('[  ]', 'FontSize',textStorlek);
ylabel('[  ]', 'FontSize',textStorlek);
zlabel('[  ]', 'FontSize',textStorlek);
```