Modeling and Simulations Exercises

Ime Rasenberg

Pieter Haas

Course: Modeling and Simulations

February 27, 2026

Contents

4 Monte Carlo simulation of Hard spheres in the NVT ensemble (correct)

4.1

Here we had to make code that tiled the space whit spheres in a cubic lattice formation. The code for the generation of this lattice is found in Appendix A.1.
On the web app for plotting this lattice, Figure 1 was made using the file that was generated.

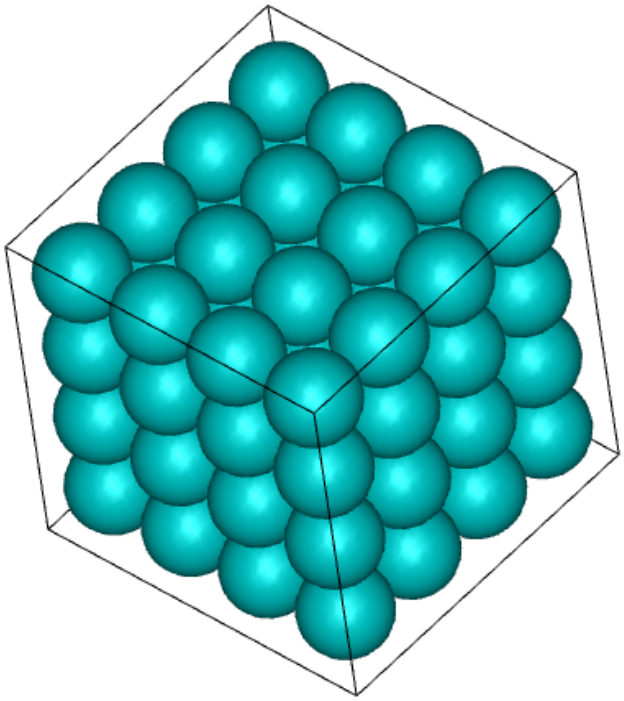


Figure 1: Here the cubic lattice generated by the code is graphed

4.2

We want to know the maximum packing density for spheres in a cubic lattice.
To get this first we need the lattice vector equation

$$\vec{R} = N_1\vec{a_1} + N_2\vec{a_2} + N_3\vec{a_3}. \tag{1}$$

Here the $N_i \in \mathcal{Z}$ is the counting number and $\vec{a_i}$ are the primitive translation vectors.
For the cubic case the vectors are unit vectors times the radius of the atoms. Dividing this up into unit cells gives us that only one atom may exist in the unit cell. meaning that the occupied fraction

$$f_o = \frac{V_p}{V_{uc}} = \frac{\frac{4}{3}\pi(\frac{a}{2})^3}{a^3} = \frac{\pi}{6} \tag{2}$$

Here V_p is the volume of particles occupying the unit cell, V_{uc} is the volume of the unit cell and a is the diameter of the particle.

4.3

Here we had to make code that tiled a space with spheres in a face-centered cubic (FCC) lattice. The code for the generation of this lattice is found in Appendix A.2.
On the web app for plotting this lattice, Figure 2 was made using the file that was generated.

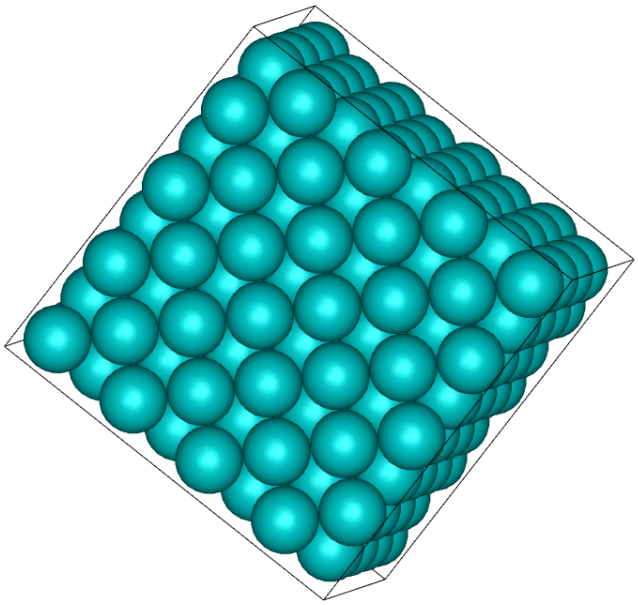


Figure 2: Here the cubic lattice generated by the code is graphed

4.4

We want to know the maximum packing density for spheres in a FCC lattice.
to get this we will use the same process as in 4.2. We know that our primitive translation vectors are

$$a_1 = \frac{a}{\sqrt{2}}\begin{pmatrix}0\\1\\1\end{pmatrix},\ a_2 = \frac{a}{\sqrt{2}}\begin{pmatrix}1\\0\\1\end{pmatrix},\ a_3 = \frac{a}{\sqrt{2}}\begin{pmatrix}1\\1\\0\end{pmatrix}$$

If we do this we have a problem, choosing the unit cell with length a in each direction leads to a non-translation symmetric unit cell. To get a translation symmetric unit cell we have to chose the length of the unit cell to be $\sqrt{2}a$.
Looking at the unit cell we can see that for each corner 1/8th of a sphere exists and we get 1/2 for each face of the cube, meaning the cube contains $8 * 1/8 + 6 * 1/2 = 4$ spheres
From this we know that if we look at how much particles would be in a unit cell we see that this would be

$$f_o = \frac{4V_p}{V_{uc}} = \frac{\frac{16}{3}\pi(\frac{a}{2})^3}{(\sqrt{2}a)^3} = \frac{\pi}{3\sqrt{2}} \tag{3}$$

4.5

The code for this *read_data*() subroutine is found in Appendix A.4.
we first initialize the file that we are interested in, we then open the file and scan the first line. The first line contains the code number of particles in the system using which we know how much of the file we need to read. Then we have a line of code to read out the 3 box dimensions that are defined. After this we know the file contains (x,y,z,r) where the x,y,z are the coordinates and the r is the radius of the box. We read these putting x,y,z into the "r" vector (1D pointer) and r into the "size" vector (1D pointer).

4.6

The code for this *move_particle*() subroutine is found in Appendix A.6.
First we generate a random particle index for our position pointer.
Then a random amplitude for our translation is generated in the domain [-delta,delta]. After a random translation direction in 3D is generated, which is then normalised and scaled by the amplitude of the translation.
This translation is validated using the move *check_particle_overlap*() if it is found to not overlap the translation is executed.
After the translation we look if the particle is still in the box. If it is not the periodic boundaries are imposed.

4.7

The code for this *check_particle_overlap*() subroutine is found in Appendix **??**.

It just tests that the random particle can do the translation without overlapping. This is done by checking its distance to all other particles and seeing if their radius's added are smaller then their distance.

4.8

The results of this NVT ensemble evolution of a cubic lattice are plotted in Figure 3.

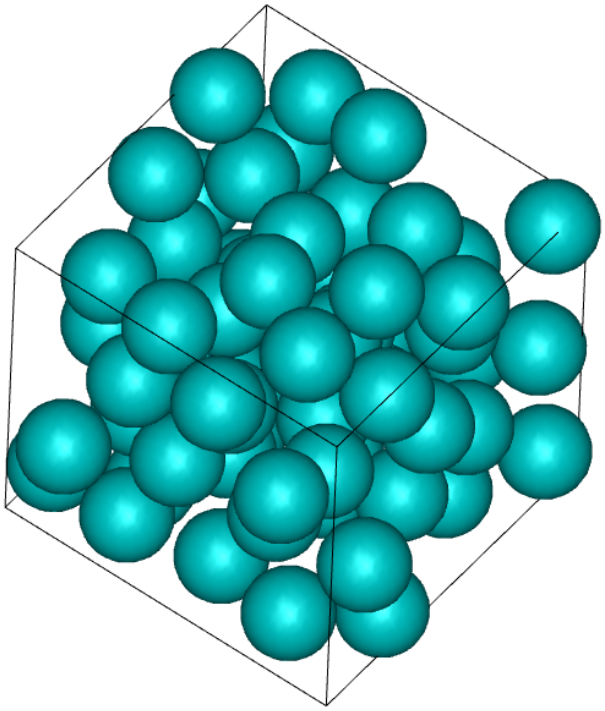
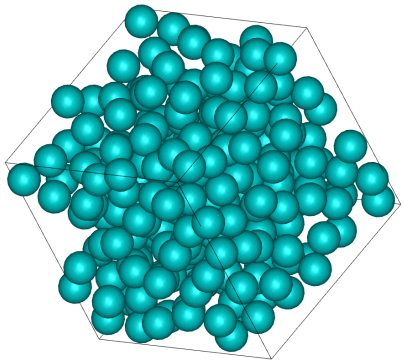


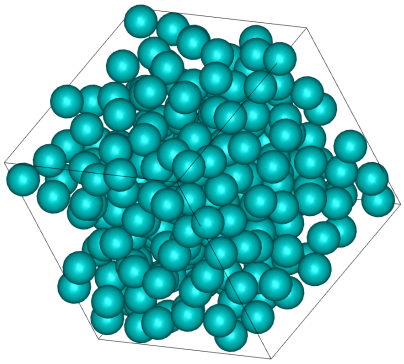
Figure 3: Here the cubic lattice generated is perturbed 100000 times to give this configuration

4.9

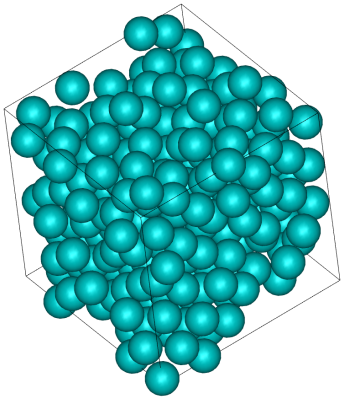
Figure 4 shows the NVT simulation at different packing densities.
At a density .55 the particles still seem to have some underlying structure, while at a density of .4 it doesn't seem to have any underlying structure, with .5 and .45 its hard to tell, so somewhere in between the structure disappears meaning it melts in the range [.4 - .55].



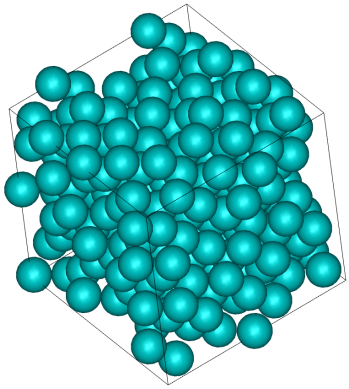
(a) the packing density is 0.40



(b) the packing density is 0.45



(c) the packing density is 0.50



(d) the packing density is 0.55

Figure 4: The packing density changed

5 MC simulation of hard spheres in the NPT ensemble

5.1

In this part a volume–change move is implemented in order to sample the NPT ensemble while ensuring that no particle overlaps occur. The code for this is given in Appendix B.1.

The function proposes a random change in volume and evaluates the acceptance probability. Since hard spheres do not contribute potential energy unless particles overlap, the acceptance probability depends only on the pressure work and the configurational entropy term,

$$acc(o \to n) = \min\left(1, \exp\left[-\beta P \Delta V - N \ln\left(\frac{V'}{V}\right)\right]\right). \tag{4}$$

This makes it so we can first check if the configuration would be accepted before checking if there is any overlap, saving on the computing time.

After a volume change is temporarily accepted, all particle positions are rescaled according to the new volume and overlaps between all particle pairs are checked using periodic boundary conditions.

5.2

First, n (the number of particles) trial displacement moves are performed in the NVT ensemble. After these particle moves, a single volume move is attempted. Together these operations define one cycle. The implementation of this procedure is shown in Appendix B.2.

Every 200 cycles the system state is written to file in order to monitor the evolution of the simulation and analyze convergence behavior.

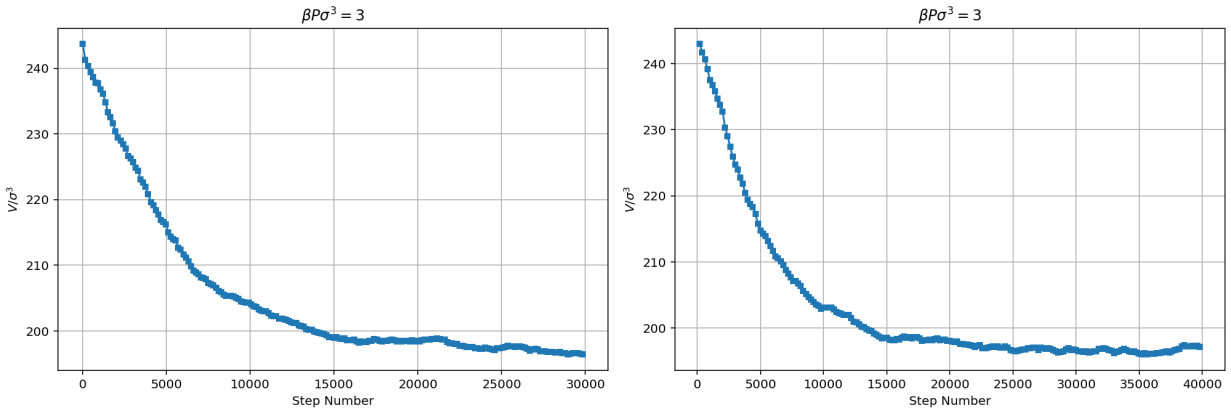
Figure 5 shows the evolution of the system volume as a function of the number of cycles. Adjusting the translation and volume step sizes improves convergence, as shown by comparing both panels. After about 15000 cycles the system seems equilibrated. illustrates the relaxation towards a steady-state volume. After approximately 15000 cycles the system appears equilibrated. From this point onward, the average volume is computed and used for further analysis.

Figure 6 shows the equalization process for systems ate different pressures in the left hand panel. From the equilibrated state the average volume is gotten from the other cycles, this is then converted in the packing fraction by

$$\eta = \frac{n\langle V_p \rangle}{\langle V_{sys,eq} \rangle}. \tag{5}$$

Here n is the number of particles, $\langle V_p \rangle$ is the average volume of these particles and $\langle V_{sys,eq} \rangle$ is the average volume of the system in equilibrium.

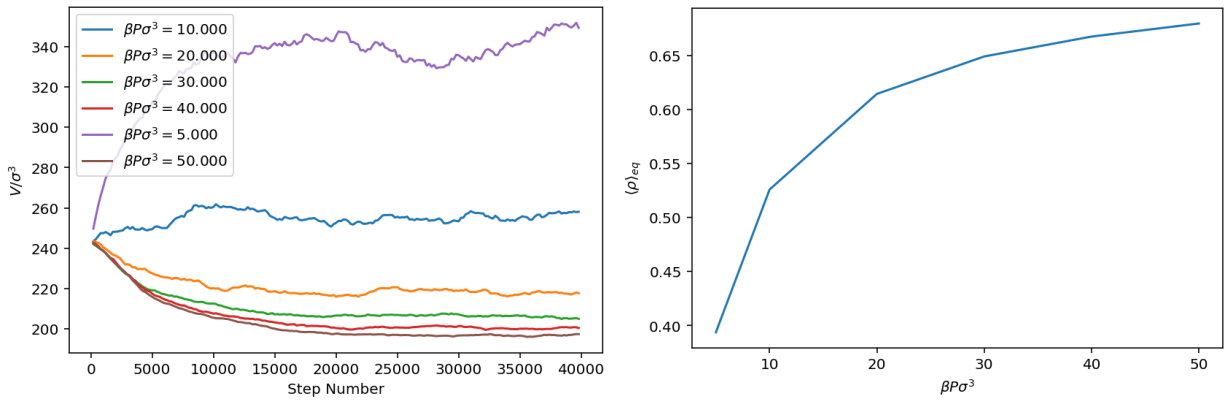
It also shows how the packing fraction changes with volume.



(a) No changes to step sizes

(b) Adjusted step sizes to improve convergence

Figure 5: Evolution of the volume during equilibration.



(a) (b)

Figure 6: (a) Convergence of the volume towards equilibrium. (b) Resulting packing fraction obtained from the equilibrated volume.

5.3

The simulation is repeated for several pressures in order to obtain the packing fraction over a wider range of pressure points. The resulting equation of state is shown in Figure 7.
For comparison, the Carnahan–Starling approximation for hard spheres is included,

$$\frac{P}{k_B T} = \frac{1 + \eta + \eta^2 - \eta^3}{(1 - \eta)^3}, \tag{6}$$

where η denotes the packing fraction.
The numerical results show good agreement with the theoretical prediction over the investigated pressure range.

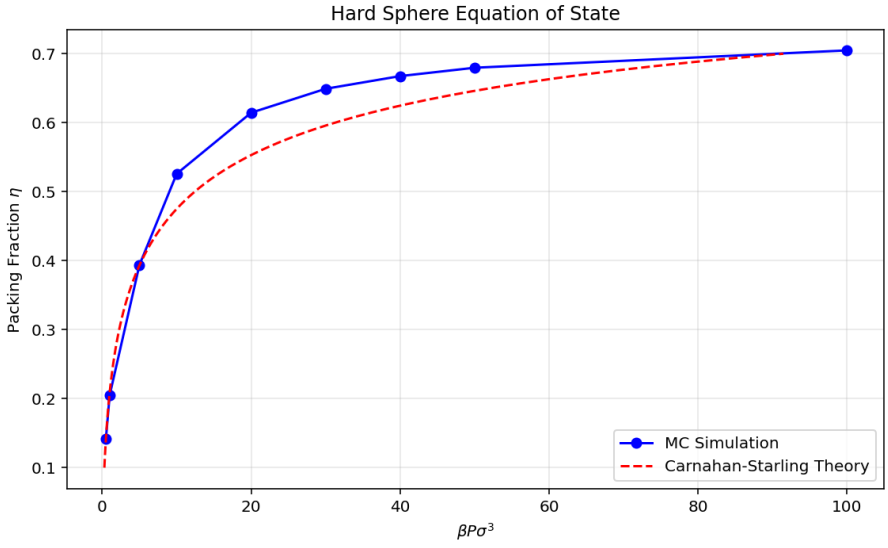


Figure 7: Packing fraction as a function of pressure compared with the Carnahan–Starling equation of state.

A Code Exercise 4

A.1

```c
#include <stdio.h>
#include <math.h>
// in this file we will make the cubic latice

int main(){
  int N = 4; // The number of particles in each dirrection
  float d = 1.0; // the distance between two spheres
  float a = 1.0; // the radius of an sphere

  // Make a file where we can save the position data
  FILE *print_coords; // inititialises a file variable
  print_coords = fopen("cubic_xyz.dat","w"); // defining the file variable to be the opening
    of some file cubic.xyz

  // Let us print some initial coordinates
  fprintf(print_coords, "%i\n", N*N*N); // the total number of particles
  fprintf(print_coords, "%lf\t%lf\n", -0.0, 1.0*d*N); // The ocupied space in the x direction
  fprintf(print_coords, "%lf\t%lf\n", -0.0, 1.0*d*N); // The ocupied space in the y direction
  fprintf(print_coords, "%lf\t%lf\n", -0.0, 1.0*d*N); // The ocupied space in the z direction

  // we first initialise the particle possision saving arrays
  float x[N*N*N], y[N*N*N], z[N*N*N], r[N*N*N];


  // now we start generating particle possitions and radiuses
  int n = 0; // this is our counting variable, it wil index which particle we will consider

  /*
  The latice points are described by
  R= a_x n_x + a_y n_x + a_z n_z
  a_x = i a
  a_y = j a
  a_z = k a
  */


  // sweeping over the N_x particles
  for(int i=0; i<N; i++){
    // sweeping over the N_y particles
    for(int j=0; j<N; j++){
      // sweeping over the N_z particles
      for(int k=0; k<N; k++){
        // generating the possition for i,j,k latice cite, also the radius of the particle
        x[n]= (i+0.5)*d;
        y[n]= (j+0.5)*d;
        z[n]= (k+0.5)*d;
        r[n]= a;

        // saving the x,y,z possition and radius of the particle
        fprintf(print_coords, "%lf\t%lf\t%lf\t%lf\n", x[n], y[n],z[n],r[n]);

        n++;
      }
    }
  }


  fclose(print_coords);
  return 0;
}
```

A.2

```c
#include <stdio.h>
#include <math.h>
// in this file we will make the cubic latice

int main(){
  int N = 4; // The number of particles in each double
  double d = 1.0; // the distance between two spheres
  double a = 1.0; // the radius of an sphere

```

```
// creating an distance variable that makes les typing
double l = sqrt(2.0)*d;

// defining the size of th  e box that will be spanned
double x_max = N*l;

// definging a variable such that the outline of the box aligns with the border of the
  particles
double s = 0.5*d;

// Make a file where we can save the position data
FILE *print_coords; // inititialises a file variable
print_coords = fopen("fcc.xyz","w"); // defining the file variable to be the opening of some
   file cubic.xyz

// Let us print some initial coordinates
 // Let us print some initial coordinates
fprintf(print_coords, "%i\n", 4*N*N*N); // the total number of particles
fprintf(print_coords, "%lf\t%lf\n", 0, x_max); // The ocupied space in the x direction
fprintf(print_coords, "%lf\t%lf\n", 0, x_max); // The ocupied space in the y direction
fprintf(print_coords, "%lf\t%lf\n", 0, x_max); // The ocupied space in the z direction

// we first initialise the particle possision saving arrays
double x[4*N*N*N], y[4*N*N*N], z[4*N*N*N], r[4*N*N*N];

// now we start generating particle possitions and radiuses
int n = 0; // this is our counting variable, it wil index which particle we will consider

/*
The latice points are described by
R= a_1 n_x + a_2 n_x + a_3 n_z
a_1 = a/2 (j + k)
a_2 = a/2 (i + k)
a_3 = a/2 (i + j)
i, j, k are the unit vectors in x, y and z directions respectively (not the counts)
*/



// sweeping over the N_x particles
for(int i=0; i<N; i++){
  // sweeping over the N_y particles
  for(int j=0; j<N; j++){
    // sweeping over the N_z particles
    for(int k=0; k<N; k++){
      // generating the possition for i,j,k latice cite, also the radius of the particle

      // first we start on the base vector because we know this patern reapeats every 2*unit
   vector in each direction
      x[n]= (i)*l;
      y[n]= (j)*l;
      z[n]= (k)*l;
      r[n]= a;

      // saving the x,y,z possition and radius of the particle
      fprintf(print_coords, "%lf\t%lf\t%lf\t%lf\n", x[n], y[n],z[n],r[n]);

      n++;

      // here we will add the a_1 vector and make the same spacing
      x[n]= (i)*l;
      y[n]= (j+0.5)*l;
      z[n]= (k+0.5)*l;
      r[n]= a;

      // saving the x,y,z possition and radius of the particle
      fprintf(print_coords, "%lf\t%lf\t%lf\t%lf\n", x[n], y[n],z[n],r[n]);

      n++;

      // here we will add the a_2 vector and make the same spacing
      x[n]= (i+0.5)*l;
      y[n]= (j)*l;
      z[n]= (k+0.5)*l;
      r[n]= a;

```

```
        // saving the x,y,z possition and radius of the particle
        fprintf(print_coords, "%lf\t%lf\t%lf\t%lf\n", x[n], y[n],z[n],r[n]);

        n++;

        // here we will add the a3 vector and continue the same spacing
        x[n]= (i+0.5)*l;
        y[n]= (j+0.5)*l;
        z[n]= (k)*l;
        r[n]= a;

        // saving the x,y,z possition and radius of the particle
        fprintf(print_coords, "%lf\t%lf\t%lf\t%lf\n", x[n], y[n],z[n],r[n]);

        n++;







      }
    }
  }


  fclose(print_coords);
  return 0;
}
```

A.3

```
  #include <stdio.h>
  #include <time.h>
  #include <assert.h>
  #include <math.h>
  #include "../downloads/mt19937.h"

  #ifndef M_PI
  #define M_PI 3.14159265358979323846
  #endif

  #define NDIM 3

  /* Initialization variables */
  const int mc_steps = 10000;
  const int output_steps = 100;
  const double packing_fraction = 0.6;
  const double diameter = 1.0;
  const double delta = 0.1;
  const char* init_filename = "FCC_xyz.dat";

  /* Simulation variables */
  int N;
  int n_particles = 0;
  double radius;
  double particle_volume;
  double (*r)[3];
  double *size;
  double box[NDIM];


  double dummy;
```

A.4

```
  void read_data(void){
    /*--------- Your code goes here -----------*/
    // degining the file
    FILE *read_cords;
    read_cords = fopen(init_filename, "r");

    // reading the first line to get the number of of particles that exist in the file (why is
     the exersise so weird???)
    fscanf(read_cords, "%i\n", &N); // the total number of particles
    // printf("%i\n", Loaded_Data.N);
```

```
    // making sure that the size will be correctly degined instead of having to asign it
    before hand
    // malloc is the memmory allocation commman which is wat we need to have exact size
    matrixes, only this satisfies me
    r = malloc(N * sizeof * r); // all the position vectors of all the particles
    size = malloc(N * sizeof * size); //The size of all particles

    // lets turn the above into a loop because i want to
    for(int i = 0; i<3; i++){
      fscanf(read_cords, "%f %f", &dummy, &box[i]);

    }

    // now that we have arived at the paricles lets be happy
    for(int i = 0; i<N; i++){
      fscanf(read_cords, "%f %f %f %f", &r[i][0], &r[i][1], &r[i][2], &size[i]);
    }

    fclose(read_cords);
  }
```

A.5

```

  int check_particle_overlap(int n){
    double *p = r[n];

    for(int i=0;i<n_particles;i++){
      if(i==n){
        continue;
      }

      double *p_c = r[i];
      double distance_squared = 0;

      for(int j=0;j<3;j++){
        double difference = p[j] + dr[j] - (p_c[j]);
        if(difference>0.5*box[j]){
          difference -= box[j];
        }
        else if(difference<-0.5*box[j]){
          difference += box[j];
        }
        distance_squared += difference*difference;
      }

      double sum_raduss = 0.5 * (size[i] + size[i]);



      if (distance_squared < sum_raduss*sum_raduss){
        // printf("%lf\t < \t %lf\n",distance_squared,sum_raduss*sum_raduss);
        return 1;
      }
    }
    // printf("accept\n");
    return 0;
  }

```

A.6

```
int move_particle(void){
  n = floor(dsfmt_genrand()*n_particles);

  for(int i=0;i<3;i++){
    dr[i] = (dsfmt_genrand()-0.5) + 0.00001;
  }
  double delta_l=(dsfmt_genrand()-0.5)*2*delta + 0.00001;

  double length = sqrt(dr[0]*dr[0] + dr[1]*dr[1] + dr[2]*dr[2]);
  for(int i=0;i<3;i++){
    dr[i] *= delta_l/length;
  }
```

```
13
14  int disp = check_particle_overlap(n);
15
16  if(disp ==1){
17    return 0;
18  }
19  else if (disp==0){
20    for(int i=0;i<3;i++){
21      r[n][i] += dr[i];
22
23      if(r[n][i]<0){
24        r[n][i] +=box[i];
25      }
26      if(r[n][i]>box[i]){
27        r[n][i] -=box[i];
28      }
29    }
30    return 1;
31  }
32
33  }
```

B Exercise 5

B.1

```
int change_volume(){

  dV = (dsfmt_genrand()-0.5)*2*dV_m;

  double V = box[0]*box[0]*box[0];

  double mult_fac = cbrt(V+dV)/box[0];

  double V_new=1;

  for(int i=0;i<3;i++){
    V_new *=box[i]*mult_fac;
  }

  double acc = fmin(1, exp(-betaP*dV + n_particles*log(V_new/V)));
  if (dsfmt_genrand()>acc){
    return 0;
  }


  double r_c[n_particles][3];

  for(int i=0; i<n_particles; i++){

    for(int j=0; j<3; j++){

      r_c[i][j] = r[i][j] *mult_fac;
    }
  }

  for(int h=0; h<n_particles; h++){


    for(int i=0; i < h; i++){

      double distance = 0;

      for(int j=0; j<3; j++){

        double dist = (r_c[h][j] - r_c[i][j]);

        if(dist>0.5*box[j]*mult_fac){
          dist -= box[j]*mult_fac;
        }
        else if(dist<-0.5*box[j]*mult_fac){
          dist += box[j]*mult_fac;
        }

        distance += dist*dist;

      }


      if (distance<(0.5*(size[h]+size[i]))*(0.5*(size[h]+size[i]))){
        // printf("volume cannot change there is overlap\n");
        return 0;
      }
    }

  }




  for(int i=0;i<3;i++){
    box[i]*=mult_fac;
  }
  for(int i=0; i<n_particles; i++){
    for(int j=0; j<3; j++){
      r[i][j] = r_c[i][j];
    }
  }
```

```
  // printf("volume changed\n");
  return 1;

}
```

B.2

```

int main(int argc, char* argv[]){
  read_data();

  assert(packing_fraction > 0.0 && packing_fraction < 1.0);
  assert(diameter > 0.0);
  assert(delta > 0.0);

  radius = 0.5 * diameter;

  if(NDIM == 3) particle_volume = M_PI * pow(diameter, 3.0) / 6.0;
  else if(NDIM == 2) particle_volume = M_PI * pow(radius, 2.0);
  else{
    printf("Number of dimensions NDIM = %d, not supported.", NDIM);
    return 0;
  }




  if(n_particles == 0){
    printf("Error: Number of particles, n_particles = 0.\n");
    return 0;
  }

  set_packing_fraction();

  dsfmt_seed(time(NULL));

  int accepted = 0;
  int step, n;
  int ind =0;
  int accepted_dv = 0;

  for(step = 1; step < mc_steps; ++step){
    for(n = 0; n < n_particles; ++n){
      accepted += move_particle();
    }
    accepted_dv += change_volume();

    if(step % output_steps == 0){

      double acceptance_move =  (double)accepted / (n_particles * output_steps);
      double acceptance_vol = (double)accepted_dv / (output_steps);

      printf("Step %d. Move acceptance: %lf.\n", step,  acceptance_move);

      printf("Step %d. Volume change acceptance: %lf.\n", step, acceptance_vol);

      write_data(step);

      if(converged_vol<4){
        if (acceptance_vol>0.55){
          dV_m *= 1.1;
        }
        else if (acceptance_vol<0.45){
          dV_m *= 0.9;
        }
        else{
          converged_move =0;
          converged_vol ++;
        }
      }

      if(converged_move<4){
        if (acceptance_move>0.55){
          delta *= 1.1;
        }
```

```
        else if (acceptance_move<0.45){
          delta *= 0.9;
        }
        else{
          converged_move ++;

        }
      }

      inf[ind][2]=(double)acceptance_vol;
      inf[ind][4]=(double)acceptance_move;
      inf[ind][3]=(double)converged_vol;
      inf[ind][5]=(double)converged_move;
      inf[ind][0]=(double)step;
      inf[ind][1]=(double)box[0]*box[0]*box[0];
      ind++;




      accepted = 0;
      accepted_dv = 0;
    }



  }



  info_2_file();

  printf("done");
  return 0;
}
```

B.3

```

void info_2_file(){
  char new_name[128];
  sprintf(new_name, "./data/data %i/info.dat",(int)floor(betaP));

  FILE *print_coords; // inititialises a file variable
  // char *new_name = "info.dat";
  print_coords = fopen(new_name,"w");

  double size_average = 0;
  for(int i = 0; i<n_particles;i++){
    size_average += size[i];
  }
  size_average/=n_particles;

  fprintf(print_coords, "%lf\t%lf\t%lf\t%i\t%lf\n", delta, dV_m, betaP, n_particles,
    size_average);

  for(int i=0;i<output_steps;i++){

    fprintf(print_coords, "%lf\t%lf\t%lf\t%lf\t%lf\t%lf\n", inf[i][0], inf[i][1],inf[i][2],inf
    [i][3],inf[i][4],inf[i][5]);
  }

  fclose(print_coords);


}
```