# 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.

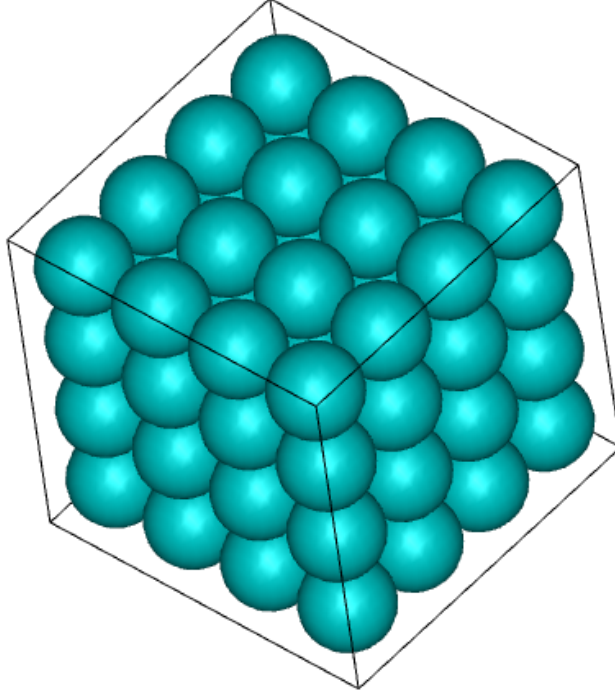On the web app for plotting this lattice, Figure 1 was made using the file that was generated.



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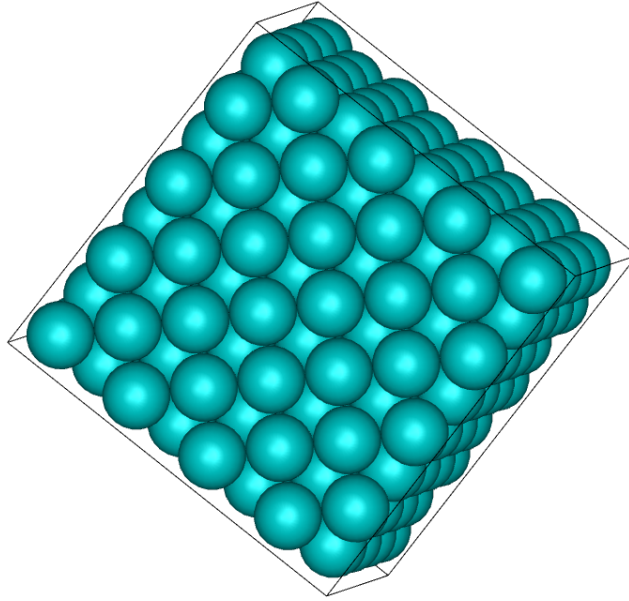
Figure 2: Here the cubic lattice generated by the code is graphed

## 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

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

## 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

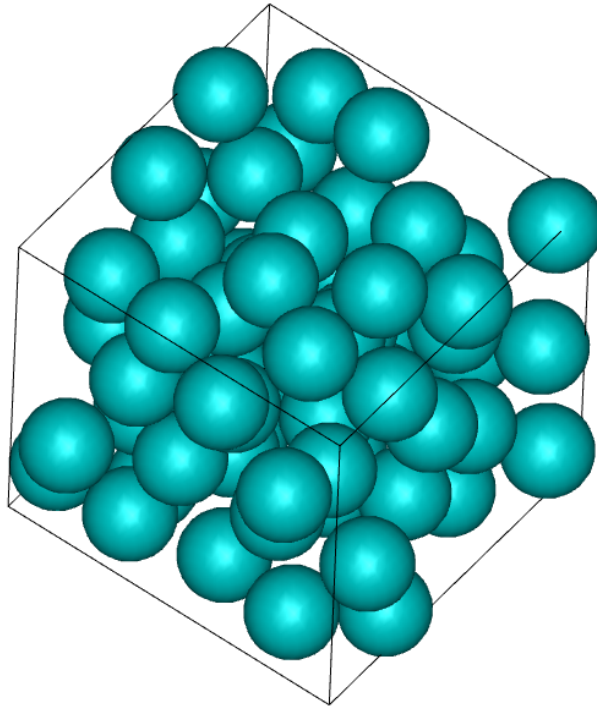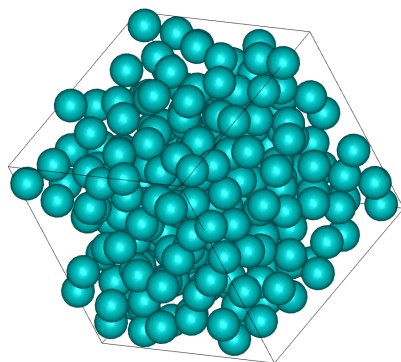The results of this NVT ensemble evolution of a cubic lattice are plotted in Figure 3.



Figure 3: Here the cubic lattice generated is perturbed 100000 times to give this configuration
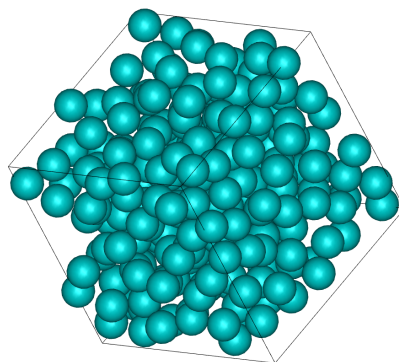
## 4.9

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].
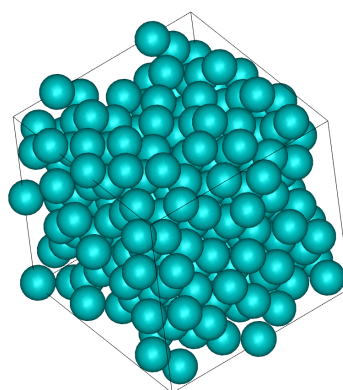
(a) the packing density is 0.40

(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

here we are asked to make a change volume to see if we have some overlap. The code for this part is found in Appendix B.1

This function first calculates the acceptance due to the change in volume and test to see if it is accepted. This is done because the random perturbations do not add any potential energy and

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

or if they overlap this would be 0.

Then the overlap between all particles is checked with periodic boundary conditions.

## 5.2

First n (# particles) moves are attempted in the NVT ensemble, then a volume move is attempted. These operations together is defined as one cycle. Every 200 cycles a state is saved to a file to keep track of what is happening over the number of steps.

Figure 8 shows how the volume changes with the number of cycles. Figure 9 shows the convergence to a steady state volume in the number of cycles. After 15000 cycles the system seems to be in equilibrium, from this point on the average volume is taken and plotted as a function of the dimensionless pressure.

changing the pressure gives a different steady state volume, which in turn gives a different packing density. This packing density as function of pressure is also plotted
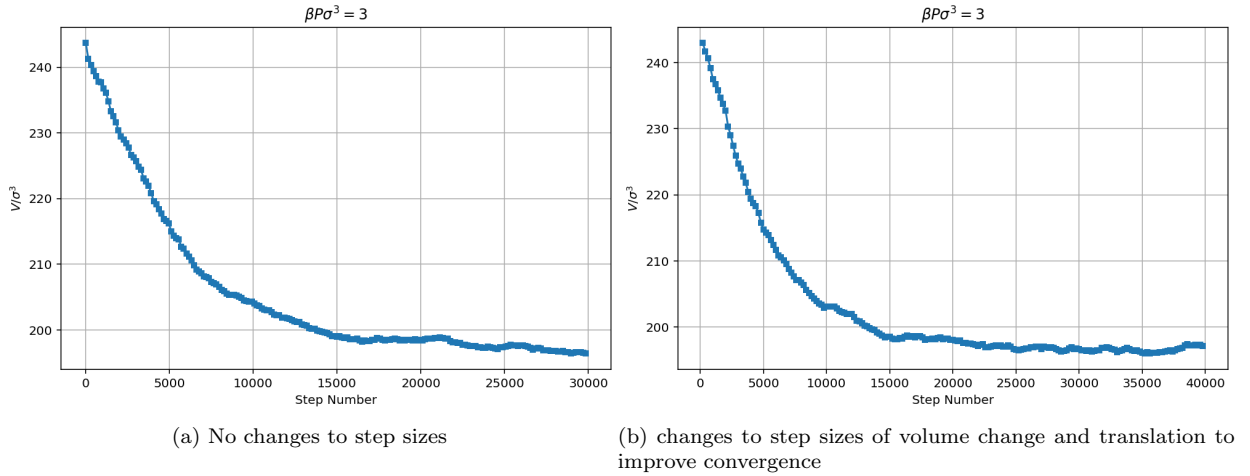


(a) No changes to step sizes

(b) changes to step sizes of volume change and translation to improve convergence
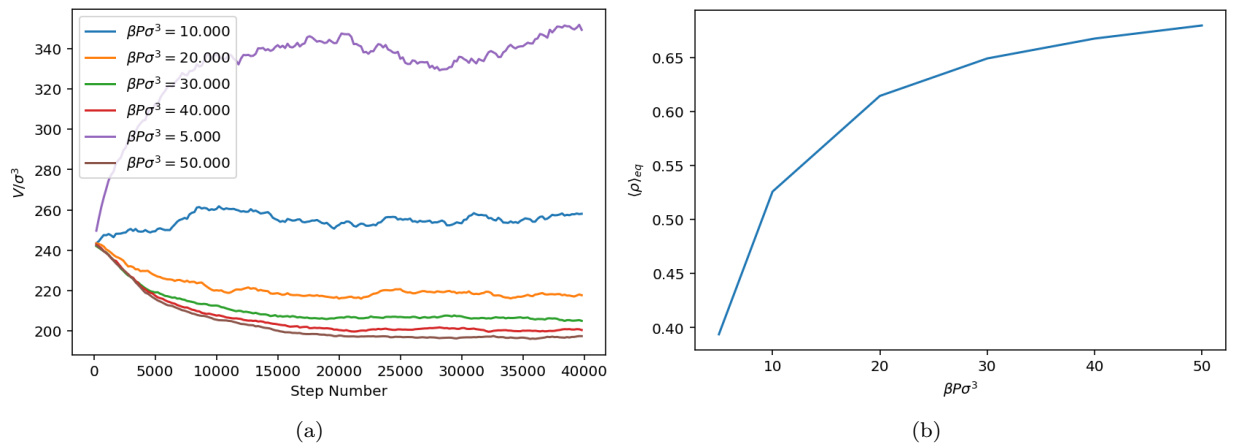
Figure 5:



(a)

(b)

Figure 6:

## 5.3

Running the simulation for more different pressures and getting the packing density gives Figure 10. In this plot you also see the carnahan-starling approximation

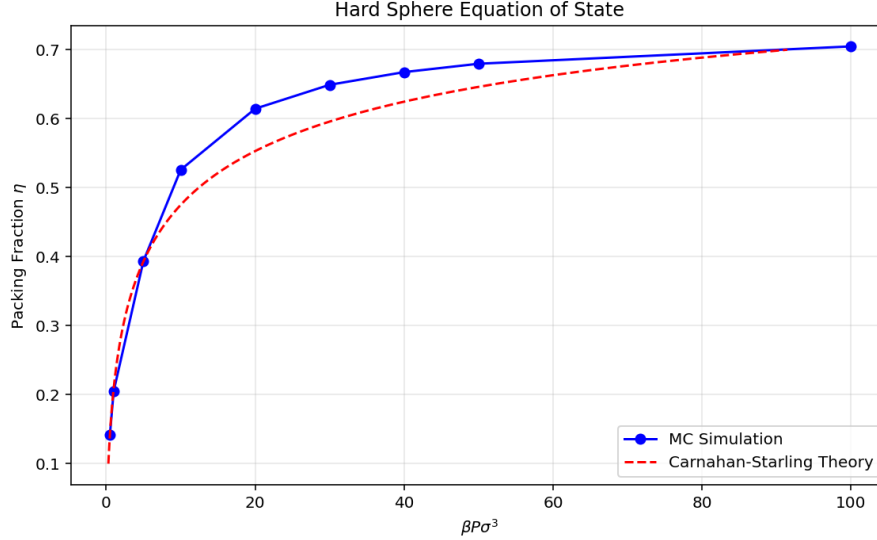$$\frac{P}{k_B T} = \frac{1 + \eta + \eta^2 - \eta^3}{(1 - \eta)^3} \tag{5}$$



Figure 7:

# 6 MC simulation of hard spheres in the NPT ensemble

## 6.1

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,

$$acc(o \to n) = \min\left(1, \exp\left[-\beta P \Delta V - N \ln\left(\frac{V'}{V}\right)\right]\right). \tag{6}$$

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.

## 6.2

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 8 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 9 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

$$\eta = \frac{n \langle V_p \rangle}{V_{sys}}. \tag{7}$$

7

Here $n$ is the number of particles

Changing the imposed pressure results in different equilibrium volumes, which correspond to different packing fractions. The packing fraction as a function of pressure is therefore also evaluated.
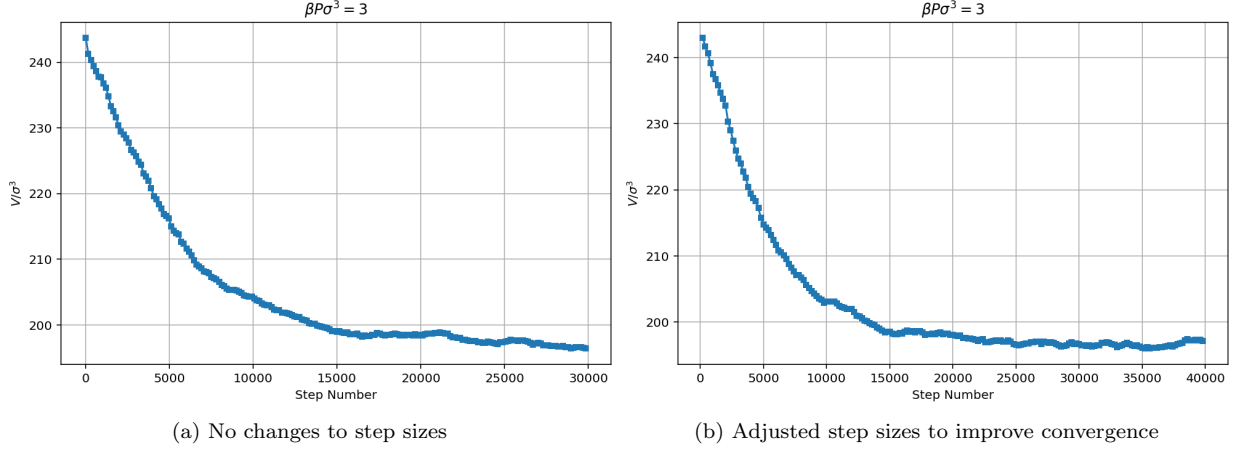


(a) No changes to step sizes

(b) Adjusted step sizes to improve convergence

Figure 8: Evolution of the volume during equilibration.



(a)

(b)

Figure 9: (a) Convergence of the volume towards equilibrium. (b) Resulting packing fraction obtained from the equilibrated volume.

## 6.3

The simulation is repeated for several pressures in order to obtain the packing fraction over a wider range of state points. The resulting equation of state is shown in Figure 10.

For comparison, the Carnahan–Starling approximation for hard spheres is included,

$$\frac{P}{k_B T} = \frac{1 + \eta + \eta^2 - \eta^3}{(1 - \eta)^3}, \tag{8}$$

where $\eta$ denotes the packing fraction.

The numerical results show good agreement with the theoretical prediction over the investigated pressure range.

Figure 10: Packing fraction as a function of pressure compared with the Carnahan–Starling equation of state.
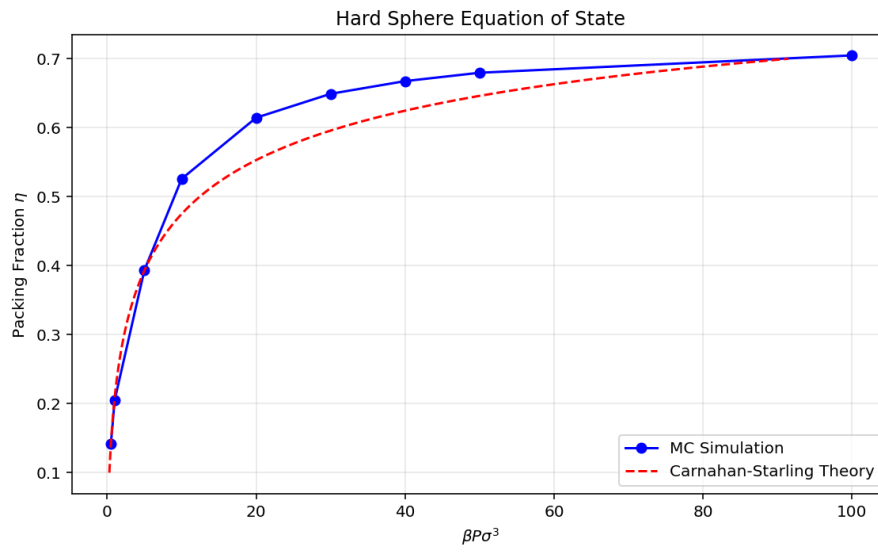
# A Code Exercise 4

## 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

```

```
10    // creating an distance variable that makes les typing
11    double l = sqrt(2.0)*d;
12
13    // defining the size of th  e box that will be spanned
14    double x_max = N*l;
15
16    // definging a variable such that the outline of the box aligns with the border of the
        particles
17    double s = 0.5*d;
18
19    // Make a file where we can save the position data
20    FILE *print_coords; // inititialises a file variable
21    print_coords = fopen("fcc.xyz","w"); // defining the file variable to be the opening of some
        file cubic.xyz
22
23    // Let us print some initial coordinates
24     // Let us print some initial coordinates
25    fprintf(print_coords, "%i\n", 4*N*N*N); // the total number of particles
26    fprintf(print_coords, "%lf\t%lf\n", 0, x_max); // The ocupied space in the x direction
27    fprintf(print_coords, "%lf\t%lf\n", 0, x_max); // The ocupied space in the y direction
28    fprintf(print_coords, "%lf\t%lf\n", 0, x_max); // The ocupied space in the z direction
29
30    // we first initialise the particle possesion saving arrays
31    double x[4*N*N*N], y[4*N*N*N], z[4*N*N*N], r[4*N*N*N];
32
33    // now we start generating particle possitions and radiuses
34    int n = 0; // this is our counting variable, it wil index which particle we will consider
35
36    /*
37    The latice points are described by
38    R= a_1 n_x + a_2 n_x + a_3 n_z
39    a_1 = a/2 (j + k)
40    a_2 = a/2 (i + k)
41    a_3 = a/2 (i + j)
42    i, j, k are the unit vectors in x, y and z directions respectively (not the counts)
43    */
44
45
46
47    // sweeping over the N_x particles
48    for(int i=0; i<N; i++){
49      // sweeping over the N_y particles
50      for(int j=0; j<N; j++){
51        // sweeping over the N_z particles
52        for(int k=0; k<N; k++){
53          // generating the possition for i,j,k latice cite, also the radius of the particle
54
55          // first we start on the base vector because we know this patern reapeats every 2*unit
        vector in each direction
56          x[n]= (i)*l;
57          y[n]= (j)*l;
58          z[n]= (k)*l;
59          r[n]= a;
60
61          // saving the x,y,z possition and radius of the particle
62          fprintf(print_coords, "%lf\t%lf\t%lf\t%lf\n", x[n], y[n],z[n],r[n]);
63
64          n++;
65
66          // here we will add the a_1 vector and make the same spacing
67          x[n]= (i)*l;
68          y[n]= (j+0.5)*l;
69          z[n]= (k+0.5)*l;
70          r[n]= a;
71
72          // saving the x,y,z possition and radius of the particle
73          fprintf(print_coords, "%lf\t%lf\t%lf\t%lf\n", x[n], y[n],z[n],r[n]);
74
75          n++;
76
77          // here we will add the a_2 vector and make the same spacing
78          x[n]= (i+0.5)*l;
79          y[n]= (j)*l;
80          z[n]= (k+0.5)*l;
81          r[n]= a;
82
```

```
83        // saving the x,y,z possition and radius of the particle
84        fprintf(print_coords, "%lf\t%lf\t%lf\t%lf\n", x[n], y[n],z[n],r[n]);
85
86        n++;
87
88        // here we will add the a3 vector and continue the same spacing
89        x[n]= (i+0.5)*l;
90        y[n]= (j+0.5)*l;
91        z[n]= (k)*l;
92        r[n]= a;
93
94        // saving the x,y,z possition and radius of the particle
95        fprintf(print_coords, "%lf\t%lf\t%lf\t%lf\n", x[n], y[n],z[n],r[n]);
96
97        n++;
98
99
100
101
102
103      }
104    }
105  }
106
107
108  fclose(print_coords);
109  return 0;
110 }
```

## A.3

```
1   #include <stdio.h>
2   #include <time.h>
3   #include <assert.h>
4   #include <math.h>
5   #include "../downloads/mt19937.h"
6
7   #ifndef M_PI
8   #define M_PI 3.14159265358979323846
9   #endif
10
11  #define NDIM 3
12
13  /* Initialization variables */
14  const int mc_steps = 10000;
15  const int output_steps = 100;
16  const double packing_fraction = 0.6;
17  const double diameter = 1.0;
18  const double delta = 0.1;
19  const char* init_filename = "FCC_xyz.dat";
20
21  /* Simulation variables */
22  int N;
23  int n_particles = 0;
24  double radius;
25  double particle_volume;
26  double (*r)[3];
27  double *size;
28  double box[NDIM];
29
30
31  double dummy;
```

## A.4

```
1   void read_data(void){
2     /*-------- Your code goes here ----------*/
3     // degining the file
4     FILE *read_cords;
5     read_cords = fopen(init_filename, "r");
6
7     // reading the first line to get the number of of particles that exist in the file (why is
        the exersise so weird???)
8     fscanf(read_cords, "%i\n", &N); // the total number of particles
9     // printf("%i\n", Loaded_Data.N);
```

```
10
11      // making sure that the size will be correctly degined instead of having to asign it
        before hand
12      // malloc is the memmory allocation commman which is wat we need to have exact size
        matrixes , only this satisfies me
13      r = malloc(N * sizeof * r); // all the position vectors of all the particles
14      size = malloc(N * sizeof * size); //The size of all particles
15
16      // lets turn the above into a loop because i want to
17      for(int i = 0; i<3; i++){
18        fscanf(read_cords, "%f %f", &dummy, &box[i]);
19
20      }
21
22      // now that we have arived at the paricles lets be happy
23      for(int i = 0; i<N; i++){
24        fscanf(read_cords, "%f %f %f %f", &r[i][0], &r[i][1], &r[i][2], &size[i]);
25      }
26
27      fclose(read_cords);
28    }
```

## A.5

```
1
2    int check_particle_overlap(int n){
3      double *p = r[n];
4
5      for(int i=0;i<n_particles;i++){
6        if(i==n){
7          continue;
8        }
9
10        double *p_c = r[i];
11        double distance_squared = 0;
12
13        for(int j=0;j<3;j++){
14          double difference = p[j] + dr[j] - (p_c[j]);
15          if(difference>0.5*box[j]){
16            difference -= box[j];
17          }
18          else if(difference<-0.5*box[j]){
19            difference += box[j];
20          }
21          distance_squared += difference*difference;
22        }
23
24        double sum_raduss = 0.5 * (size[i] + size[i]);
25
26
27
28        if (distance_squared < sum_raduss*sum_raduss){
29          // printf("%lf\t < \t %lf\n",distance_squared,sum_raduss*sum_raduss);
30          return 1;
31        }
32      }
33      // printf("accept\n");
34      return 0;
35    }
36
```

## A.6

```
1  int move_particle(void){
2    n = floor(dsfmt_genrand()*n_particles);
3
4    for(int i=0;i<3;i++){
5      dr[i] = (dsfmt_genrand()-0.5) + 0.00001;
6    }
7    double delta_l=(dsfmt_genrand()-0.5)*2*delta + 0.00001;
8
9    double length = sqrt(dr[0]*dr[0] + dr[1]*dr[1] + dr[2]*dr[2]);
10    for(int i=0;i<3;i++){
11      dr[i] *= delta_l/length;
12    }
```

```c
   int disp = check_particle_overlap(n);

   if(disp ==1){
     return 0;
   }
   else if (disp==0){
     for(int i=0;i<3;i++){
       r[n][i] += dr[i];

       if(r[n][i]<0){
         r[n][i] +=box[i];
       }
       if(r[n][i]>box[i]){
         r[n][i] -=box[i];
       }
     }
     return 1;
   }

   }
```

```c
   int disp = check_particle_overlap(n);

   if(disp ==1){
     return 0;
   }
   else if (disp==0){
     for(int i=0;i<3;i++){
       r[n][i] += dr[i];

       if(r[n][i]<0){
```

# B  Exercise 5

## B.1

```c
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];
    }
  }
```

```
73
74    // printf("volume changed\n");
75    return 1;
76
77  }
```

## B.2

```
1
2   int main(int argc, char* argv[]){
3     read_data();
4
5     assert(packing_fraction > 0.0 && packing_fraction < 1.0);
6     assert(diameter > 0.0);
7     assert(delta > 0.0);
8
9     radius = 0.5 * diameter;
10
11    if(NDIM == 3) particle_volume = M_PI * pow(diameter, 3.0) / 6.0;
12    else if(NDIM == 2) particle_volume = M_PI * pow(radius, 2.0);
13    else{
14      printf("Number of dimensions NDIM = %d, not supported.", NDIM);
15      return 0;
16    }
17
18
19
20
21    if(n_particles == 0){
22      printf("Error: Number of particles, n_particles = 0.\n");
23      return 0;
24    }
25
26    set_packing_fraction();
27
28    dsfmt_seed(time(NULL));
29
30    int accepted = 0;
31    int step, n;
32    int ind =0;
33    int accepted_dv = 0;
34
35    for(step = 1; step < mc_steps; ++step){
36      for(n = 0; n < n_particles; ++n){
37        accepted += move_particle();
38      }
39      accepted_dv += change_volume();
40
41      if(step % output_steps == 0){
42
43        double acceptance_move =  (double)accepted / (n_particles * output_steps);
44        double acceptance_vol = (double)accepted_dv / (output_steps);
45
46        printf("Step %d. Move acceptance: %lf.\n", step,  acceptance_move);
47
48        printf("Step %d. Volume change acceptance: %lf.\n", step, acceptance_vol);
49
50        write_data(step);
51
52        if(converged_vol<4){
53          if (acceptance_vol>0.55){
54            dV_m *= 1.1;
55          }
56          else if (acceptance_vol<0.45){
57            dV_m *= 0.9;
58          }
59          else{
60            converged_move =0;
61            converged_vol ++;
62          }
63        }
64
65        if(converged_move<4){
66          if (acceptance_move>0.55){
67            delta *= 1.1;
68          }
```

```
69        else if (acceptance_move <0.45){
70          delta *= 0.9;
71        }
72        else{
73          converged_move ++;
74
75        }
76      }
77
78      inf[ind][2]=(double)acceptance_vol;
79      inf[ind][4]=(double)acceptance_move;
80      inf[ind][3]=(double)converged_vol;
81      inf[ind][5]=(double)converged_move;
82      inf[ind][0]=(double)step;
83      inf[ind][1]=(double)box[0]*box[0]*box[0];
84      ind++;
85
86
87
88
89      accepted = 0;
90      accepted_dv = 0;
91    }
92
93
94
95  }
96
97
98
99  info_2_file();
100
101  printf("done");
102  return 0;
103 }
```

## B.3

```
1
2 void info_2_file(){
3   char new_name[128];
4   sprintf(new_name, "./data/data %i/info.dat",(int)floor(betaP));
5
6   FILE *print_coords; // inititialises a file variable
7   // char *new_name = "info.dat";
8   print_coords = fopen(new_name,"w");
9
10  double size_average = 0;
11  for(int i = 0; i<n_particles;i++){
12    size_average += size[i];
13  }
14  size_average/=n_particles;
15
16  fprintf(print_coords, "%lf\t%lf\t%lf\t%i\t%lf\n", delta, dV_m, betaP, n_particles,
     size_average);
17
18  for(int i=0;i<output_steps;i++){
19
20    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]);
21  }
22
23  fclose(print_coords);
24
25
26 }
```