

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)	2
4.1	2
4.2	2
4.3	2
4.4	3
4.5	3
4.6	3
4.7	3
4.8	4
4.9	4
5 MC simulation of hard spheres in the NPT ensemble	6
5.1	6
5.2	6
5.3	7
A Code Exercise 4	8
A.1	8
A.2	8
A.3	10
A.4	10
A.5	11
A.6	11
B Exercise 5	13
B.1	13
B.2	14
B.3	15

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

4.1

Here we had to make code that tiled the space with 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.

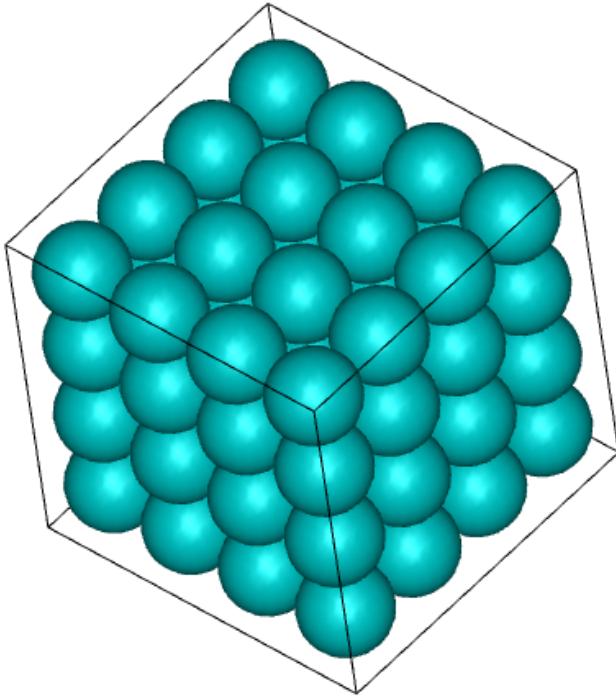


Figure 1: Here the cubic lattice generated by the code is graphed

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

$$\vec{R} = N_1 \vec{a}_1 + N_2 \vec{a}_2 + N_3 \vec{a}_3. \quad (1)$$

Here the $N_i \in \mathbb{Z}$ 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

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

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.

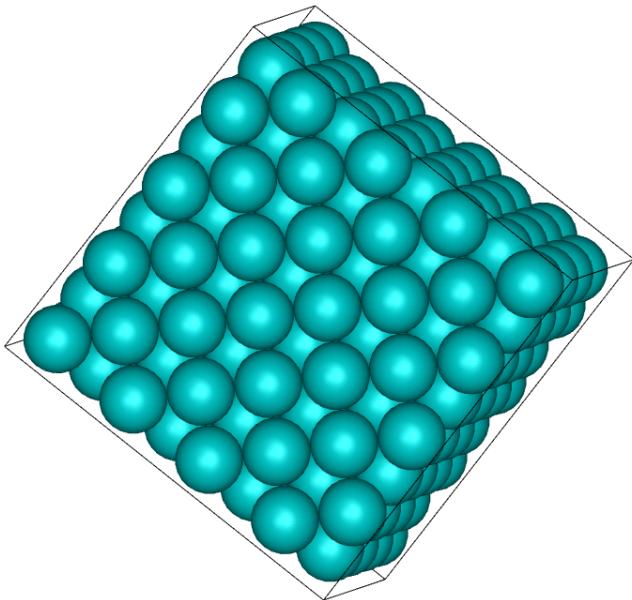


Figure 2: Here the cubic lattice generated by the code is graphed

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

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

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

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

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.

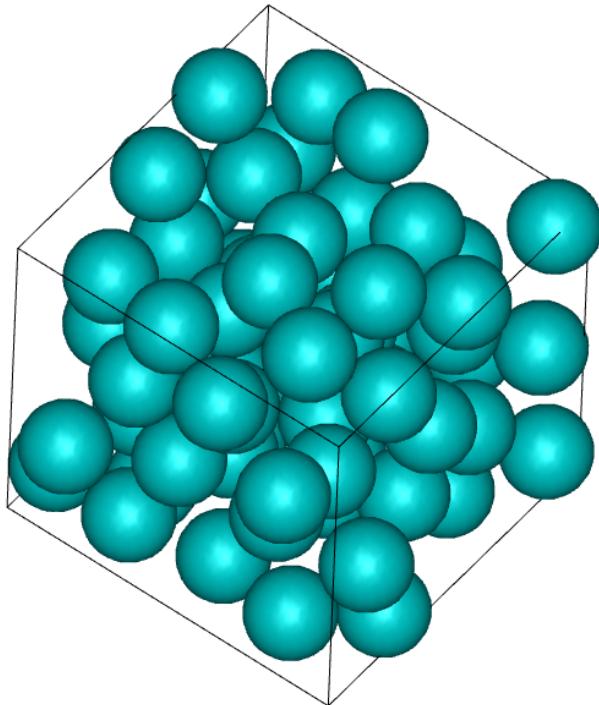
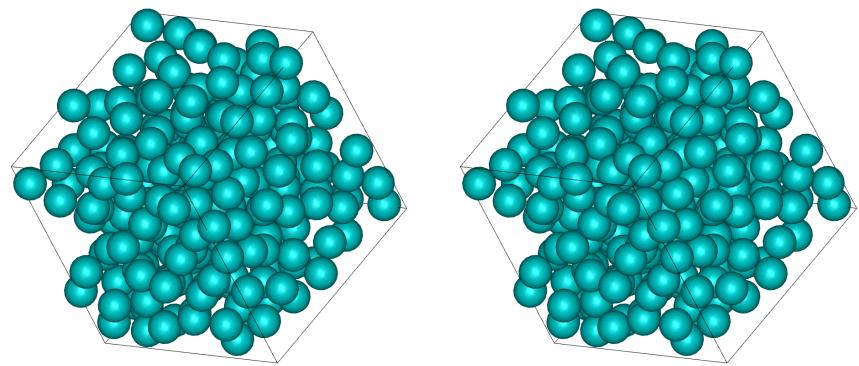


Figure 3: Here the cubic lattice generated is perturbed 100000 times to give this configuration

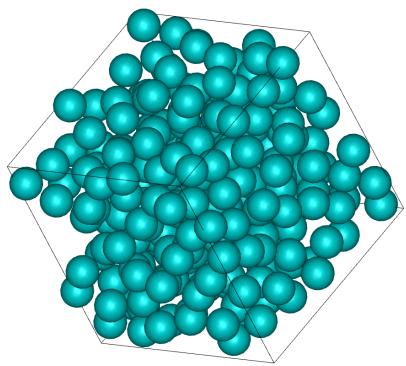
4.9

Figure 4 shows the NVT simulation at different packing densities.

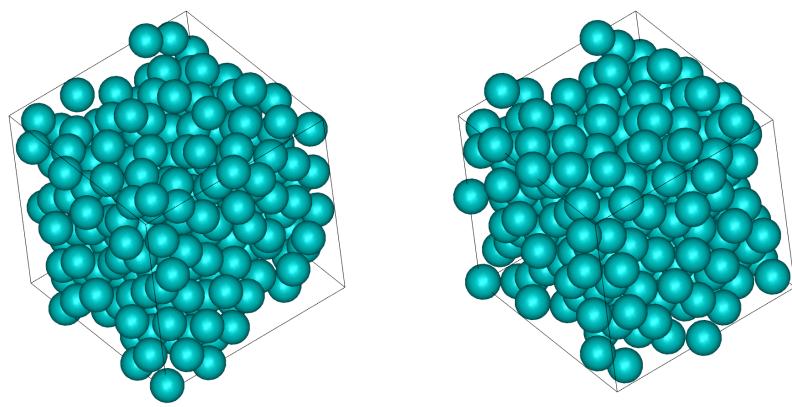
At a density .55 the particles still seem to have some underlying structure, while at a density of .4 it doesn't seem to have any underlying structure, with .5 and .45 its hard to tell, so somewhere in between the structure disappears meaning it melts in the range [.4 - .55].



(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,

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

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 at 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

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

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.

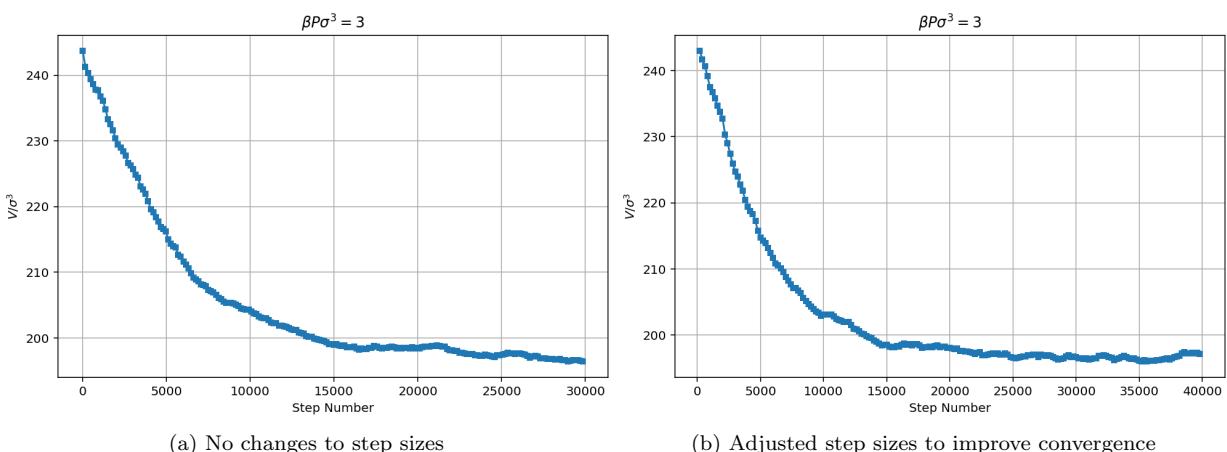


Figure 5: Evolution of the volume during equilibration.

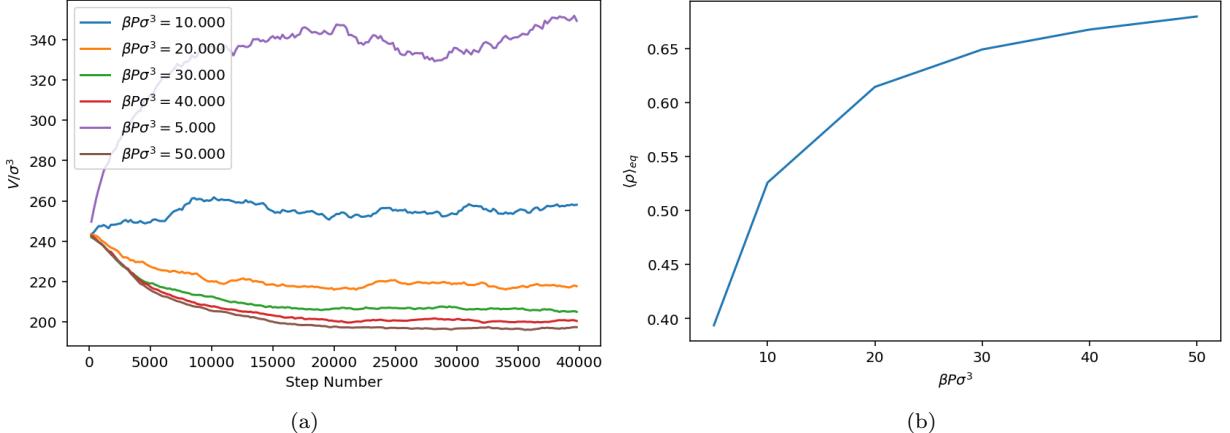


Figure 6: (a) Convergence of the volume towards equilibrium. (b) Resulting packing fraction obtained from the equilibrated volume.

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,

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

where η denotes the packing fraction.

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

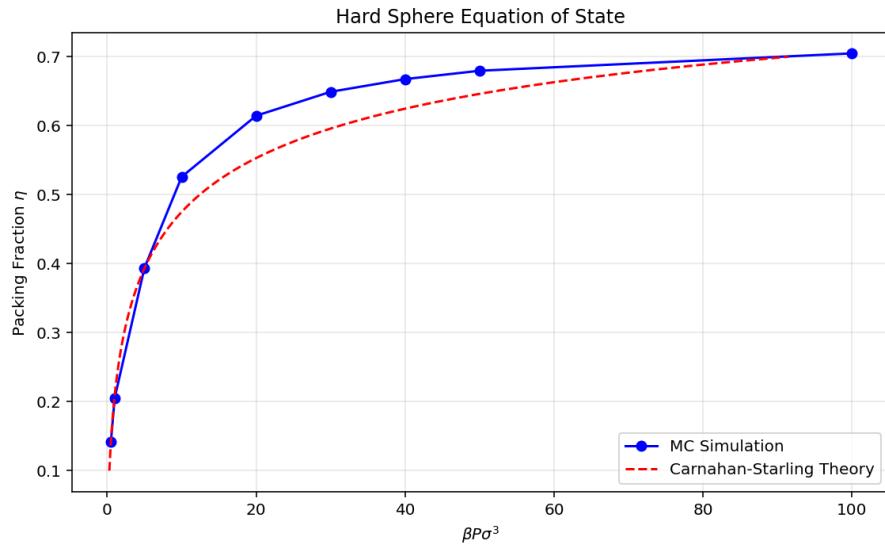


Figure 7: Packing fraction as a function of pressure compared with the Carnahan–Starling equation of state.

A Code Exercise 4

A.1

```
1 #include <stdio.h>
2 #include <math.h>
3 // in this file we will make the cubic lattice
4
5 int main(){
6     int N = 4; // The number of particles in each direction
7     float d = 1.0; // the distance between two spheres
8     float a = 1.0; // the radius of an sphere
9
10    // Make a file where we can save the position data
11    FILE *print_coords; // initialises a file variable
12    print_coords = fopen("cubic_xyz.dat","w"); // defining the file variable to be the opening
13        of some file cubic.xyz
14
15    // Let us print some initial coordinates
16    fprintf(print_coords, "%i\n", N*N*N); // the total number of particles
17    fprintf(print_coords, "%lf\t%lf\n", -0.0, 1.0*d*N); // The occupied space in the x direction
18    fprintf(print_coords, "%lf\t%lf\n", -0.0, 1.0*d*N); // The occupied space in the y direction
19    fprintf(print_coords, "%lf\t%lf\n", -0.0, 1.0*d*N); // The occupied space in the z direction
20
21    // we first initialise the particle possision saving arrays
22    float x[N*N*N], y[N*N*N], z[N*N*N], r[N*N*N];
23
24
25    // now we start generating particle possisions and radiuses
26    int n = 0; // this is our counting variable, it wil index which particle we will consider
27
28    /*
29    The lattice points are described by
30    R= a_x n_x + a_y n_y + a_z n_z
31    a_x = i a
32    a_y = j a
33    a_z = k a
34    */
35
36    // sweeping over the N_x particles
37    for(int i=0; i<N; i++){
38        // sweeping over the N_y particles
39        for(int j=0; j<N; j++){
40            // sweeping over the N_z particles
41            for(int k=0; k<N; k++){
42                // generating the possition for i,j,k lattice cite, also the radius of the particle
43                x[n] = (i+0.5)*d;
44                y[n] = (j+0.5)*d;
45                z[n] = (k+0.5)*d;
46                r[n] = a;
47
48                // saving the x,y,z possition and radius of the particle
49                fprintf(print_coords, "%lf\t%lf\t%lf\t%lf\n", x[n], y[n], z[n], r[n]);
50
51                n++;
52            }
53        }
54    }
55
56
57    fclose(print_coords);
58    return 0;
59 }
```

A.2

```
1 #include <stdio.h>
2 #include <math.h>
3 // in this file we will make the cubic lattice
4
5 int main(){
6     int N = 4; // The number of particles in each double
7     double d = 1.0; // the distance between two spheres
8     double a = 1.0; // the radius of an sphere
9
```

```

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

```

```

83     // saving the x,y,z position 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 particles that exist in the file (why is
8     // the exersise so weird??)
9     fscanf(read_cords, "%i\n", &N); // the total number of particles
// printf("%i\n", Loaded_Data.N);
```

```

10 // making sure that the size will be correctly defined instead of having to assign it
11 // before hand
12 // malloc is the memory allocation command which is what we need to have exact size
13 // matrixes, only this satisfies me
14 r = malloc(N * sizeof * r); // all the position vectors of all the particles
15 size = malloc(N * sizeof * size); //The size of all particles
16
17 // lets turn the above into a loop because i want to
18 for(int i = 0; i<3; i++){
19     fscanf(read_cords, "%f %f", &dummy, &box[i]);
20 }
21
22 // now that we have arrived at the particles 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 int check_particle_overlap(int n){
2     double *p = r[n];
3
4     for(int i=0;i<n_particles;i++){
5         if(i==n){
6             continue;
7         }
8
9         double *p_c = r[i];
10        double distance_squared = 0;
11
12        for(int j=0;j<3;j++){
13            double difference = p[j] + dr[j] - (p_c[j]);
14            if(difference>0.5*box[j]){
15                difference -= box[j];
16            }
17            else if(difference<-0.5*box[j]){
18                difference += box[j];
19            }
20            distance_squared += difference*difference;
21        }
22
23        double sum_raduss = 0.5 * (size[i] + size[i]);
24
25
26
27        if (distance_squared < sum_raduss*sum_raduss){
28            // printf("%lf\t < \t %lf\n",distance_squared,sum_raduss*sum_raduss);
29            return 1;
30        }
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    }
```

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

B Exercise 5

B.1

```
1 int change_volume(){
2
3     dV = (dsfmt_genrand() - 0.5)*2*dV_m;
4
5     double V = box[0]*box[0]*box[0];
6
7     double mult_fac = cbrt(V+dV)/box[0];
8
9     double V_new=1;
10
11    for(int i=0; i<3; i++){
12        V_new *=box[i]*mult_fac;
13    }
14
15    double acc = fmin(1, exp(-betaP*dV + n_particles*log(V_new/V)));
16    if (dsfmt_genrand()>acc){
17        return 0;
18    }
19
20    double r_c[n_particles][3];
21
22    for(int i=0; i<n_particles; i++){
23
24        for(int j=0; j<3; j++){
25
26            r_c[i][j] = r[i][j] *mult_fac;
27        }
28    }
29
30    for(int h=0; h<n_particles; h++){
31
32        for(int i=0; i < h; i++){
33
34            double distance = 0;
35
36            for(int j=0; j<3; j++){
37
38                double dist = (r_c[h][j] - r_c[i][j]);
39
40                if(dist>0.5*box[j]*mult_fac){
41                    dist -= box[j]*mult_fac;
42                }
43                else if(dist<-0.5*box[j]*mult_fac){
44                    dist += box[j]*mult_fac;
45                }
46
47                distance += dist*dist;
48
49            }
50
51        }
52
53
54        if (distance<(0.5*(size[h]+size[i]))*(0.5*(size[h]+size[i]))){
55            // printf("volume cannot change there is overlap\n");
56            return 0;
57        }
58    }
59
60 }
61
62
63
64
65    for(int i=0;i<3;i++){
66        box[i]*=mult_fac;
67    }
68    for(int i=0; i<n_particles; i++){
69        for(int j=0; j<3; j++){
70            r[i][j] = r_c[i][j];
71        }
72    }
```

```

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

B.2

```

1 int main(int argc, char* argv[]){
2     read_data();
3
4     assert(packing_fraction > 0.0 && packing_fraction < 1.0);
5     assert(diameter > 0.0);
6     assert(delta > 0.0);
7
8     radius = 0.5 * diameter;
9
10    if(NDIM == 3) particle_volume = M_PI * pow(diameter, 3.0) / 6.0;
11    else if(NDIM == 2) particle_volume = M_PI * pow(radius, 2.0);
12    else{
13        printf("Number of dimensions NDIM = %d, not supported.", NDIM);
14        return 0;
15    }
16
17
18
19
20    if(n_particles == 0){
21        printf("Error: Number of particles, n_particles = 0.\n");
22        return 0;
23    }
24
25    set_packing_fraction();
26
27    dprintf_seed(time(NULL));
28
29    int accepted = 0;
30    int step, n;
31    int ind =0;
32    int accepted_dv = 0;
33
34    for(step = 1; step < mc_steps; ++step){
35        for(n = 0; n < n_particles; ++n){
36            accepted += move_particle();
37        }
38        accepted_dv += change_volume();
39
40        if(step % output_steps == 0){
41
42            double acceptance_move = (double)accepted / (n_particles * output_steps);
43            double acceptance_vol = (double)accepted_dv / (output_steps);
44
45            printf("Step %d. Move acceptance: %lf.\n", step, acceptance_move);
46            printf("Step %d. Volume change acceptance: %lf.\n", step, acceptance_vol);
47
48            write_data(step);
49
50            if(converged_vol<4){
51                if (acceptance_vol>0.55){
52                    dV_m *= 1.1;
53                }
54                else if (acceptance_vol<0.45){
55                    dV_m *= 0.9;
56                }
57                else{
58                    converged_move =0;
59                    converged_vol++;
60                }
61            }
62        }
63
64        if(converged_move<4){
65            if (acceptance_move>0.55){
66                delta *= 1.1;
67            }
68        }
69    }
70
71    if(delta < 0.001)
72        break;
73
74    write_data(step);
75
76    if(acceptance_move > 0.55)
77        delta *= 1.1;
78
79    if(delta < 0.001)
80        break;
81
82    write_data(step);
83
84    if(acceptance_move > 0.55)
85        delta *= 1.1;
86
87    if(delta < 0.001)
88        break;
89
90    write_data(step);
91
92    if(acceptance_move > 0.55)
93        delta *= 1.1;
94
95    if(delta < 0.001)
96        break;
97
98    write_data(step);
99
100   if(acceptance_move > 0.55)
101      delta *= 1.1;
102
103   if(delta < 0.001)
104      break;
105
106   write_data(step);
107
108   if(acceptance_move > 0.55)
109      delta *= 1.1;
110
111   if(delta < 0.001)
112      break;
113
114   write_data(step);
115
116   if(acceptance_move > 0.55)
117      delta *= 1.1;
118
119   if(delta < 0.001)
120      break;
121
122   write_data(step);
123
124   if(acceptance_move > 0.55)
125      delta *= 1.1;
126
127   if(delta < 0.001)
128      break;
129
130   write_data(step);
131
132   if(acceptance_move > 0.55)
133      delta *= 1.1;
134
135   if(delta < 0.001)
136      break;
137
138   write_data(step);
139
140   if(acceptance_move > 0.55)
141      delta *= 1.1;
142
143   if(delta < 0.001)
144      break;
145
146   write_data(step);
147
148   if(acceptance_move > 0.55)
149      delta *= 1.1;
150
151   if(delta < 0.001)
152      break;
153
154   write_data(step);
155
156   if(acceptance_move > 0.55)
157      delta *= 1.1;
158
159   if(delta < 0.001)
160      break;
161
162   write_data(step);
163
164   if(acceptance_move > 0.55)
165      delta *= 1.1;
166
167   if(delta < 0.001)
168      break;
169
170   write_data(step);
171
172   if(acceptance_move > 0.55)
173      delta *= 1.1;
174
175   if(delta < 0.001)
176      break;
177
178   write_data(step);
179
180   if(acceptance_move > 0.55)
181      delta *= 1.1;
182
183   if(delta < 0.001)
184      break;
185
186   write_data(step);
187
188   if(acceptance_move > 0.55)
189      delta *= 1.1;
190
191   if(delta < 0.001)
192      break;
193
194   write_data(step);
195
196   if(acceptance_move > 0.55)
197      delta *= 1.1;
198
199   if(delta < 0.001)
200      break;
201
202   write_data(step);
203
204   if(acceptance_move > 0.55)
205      delta *= 1.1;
206
207   if(delta < 0.001)
208      break;
209
210   write_data(step);
211
212   if(acceptance_move > 0.55)
213      delta *= 1.1;
214
215   if(delta < 0.001)
216      break;
217
218   write_data(step);
219
220   if(acceptance_move > 0.55)
221      delta *= 1.1;
222
223   if(delta < 0.001)
224      break;
225
226   write_data(step);
227
228   if(acceptance_move > 0.55)
229      delta *= 1.1;
230
231   if(delta < 0.001)
232      break;
233
234   write_data(step);
235
236   if(acceptance_move > 0.55)
237      delta *= 1.1;
238
239   if(delta < 0.001)
240      break;
241
242   write_data(step);
243
244   if(acceptance_move > 0.55)
245      delta *= 1.1;
246
247   if(delta < 0.001)
248      break;
249
250   write_data(step);
251
252   if(acceptance_move > 0.55)
253      delta *= 1.1;
254
255   if(delta < 0.001)
256      break;
257
258   write_data(step);
259
260   if(acceptance_move > 0.55)
261      delta *= 1.1;
262
263   if(delta < 0.001)
264      break;
265
266   write_data(step);
267
268   if(acceptance_move > 0.55)
269      delta *= 1.1;
270
271   if(delta < 0.001)
272      break;
273
274   write_data(step);
275
276   if(acceptance_move > 0.55)
277      delta *= 1.1;
278
279   if(delta < 0.001)
280      break;
281
282   write_data(step);
283
284   if(acceptance_move > 0.55)
285      delta *= 1.1;
286
287   if(delta < 0.001)
288      break;
289
290   write_data(step);
291
292   if(acceptance_move > 0.55)
293      delta *= 1.1;
294
295   if(delta < 0.001)
296      break;
297
298   write_data(step);
299
300   if(acceptance_move > 0.55)
301      delta *= 1.1;
302
303   if(delta < 0.001)
304      break;
305
306   write_data(step);
307
308   if(acceptance_move > 0.55)
309      delta *= 1.1;
310
311   if(delta < 0.001)
312      break;
313
314   write_data(step);
315
316   if(acceptance_move > 0.55)
317      delta *= 1.1;
318
319   if(delta < 0.001)
320      break;
321
322   write_data(step);
323
324   if(acceptance_move > 0.55)
325      delta *= 1.1;
326
327   if(delta < 0.001)
328      break;
329
330   write_data(step);
331
332   if(acceptance_move > 0.55)
333      delta *= 1.1;
334
335   if(delta < 0.001)
336      break;
337
338   write_data(step);
339
340   if(acceptance_move > 0.55)
341      delta *= 1.1;
342
343   if(delta < 0.001)
344      break;
345
346   write_data(step);
347
348   if(acceptance_move > 0.55)
349      delta *= 1.1;
350
351   if(delta < 0.001)
352      break;
353
354   write_data(step);
355
356   if(acceptance_move > 0.55)
357      delta *= 1.1;
358
359   if(delta < 0.001)
360      break;
361
362   write_data(step);
363
364   if(acceptance_move > 0.55)
365      delta *= 1.1;
366
367   if(delta < 0.001)
368      break;
369
370   write_data(step);
371
372   if(acceptance_move > 0.55)
373      delta *= 1.1;
374
375   if(delta < 0.001)
376      break;
377
378   write_data(step);
379
380   if(acceptance_move > 0.55)
381      delta *= 1.1;
382
383   if(delta < 0.001)
384      break;
385
386   write_data(step);
387
388   if(acceptance_move > 0.55)
389      delta *= 1.1;
390
391   if(delta < 0.001)
392      break;
393
394   write_data(step);
395
396   if(acceptance_move > 0.55)
397      delta *= 1.1;
398
399   if(delta < 0.001)
400      break;
401
402   write_data(step);
403
404   if(acceptance_move > 0.55)
405      delta *= 1.1;
406
407   if(delta < 0.001)
408      break;
409
410   write_data(step);
411
412   if(acceptance_move > 0.55)
413      delta *= 1.1;
414
415   if(delta < 0.001)
416      break;
417
418   write_data(step);
419
420   if(acceptance_move > 0.55)
421      delta *= 1.1;
422
423   if(delta < 0.001)
424      break;
425
426   write_data(step);
427
428   if(acceptance_move > 0.55)
429      delta *= 1.1;
430
431   if(delta < 0.001)
432      break;
433
434   write_data(step);
435
436   if(acceptance_move > 0.55)
437      delta *= 1.1;
438
439   if(delta < 0.001)
440      break;
441
442   write_data(step);
443
444   if(acceptance_move > 0.55)
445      delta *= 1.1;
446
447   if(delta < 0.001)
448      break;
449
450   write_data(step);
451
452   if(acceptance_move > 0.55)
453      delta *= 1.1;
454
455   if(delta < 0.001)
456      break;
457
458   write_data(step);
459
460   if(acceptance_move > 0.55)
461      delta *= 1.1;
462
463   if(delta < 0.001)
464      break;
465
466   write_data(step);
467
468   if(acceptance_move > 0.55)
469      delta *= 1.1;
470
471   if(delta < 0.001)
472      break;
473
474   write_data(step);
475
476   if(acceptance_move > 0.55)
477      delta *= 1.1;
478
479   if(delta < 0.001)
480      break;
481
482   write_data(step);
483
484   if(acceptance_move > 0.55)
485      delta *= 1.1;
486
487   if(delta < 0.001)
488      break;
489
490   write_data(step);
491
492   if(acceptance_move > 0.55)
493      delta *= 1.1;
494
495   if(delta < 0.001)
496      break;
497
498   write_data(step);
499
500   if(acceptance_move > 0.55)
501      delta *= 1.1;
502
503   if(delta < 0.001)
504      break;
505
506   write_data(step);
507
508   if(acceptance_move > 0.55)
509      delta *= 1.1;
510
511   if(delta < 0.001)
512      break;
513
514   write_data(step);
515
516   if(acceptance_move > 0.55)
517      delta *= 1.1;
518
519   if(delta < 0.001)
520      break;
521
522   write_data(step);
523
524   if(acceptance_move > 0.55)
525      delta *= 1.1;
526
527   if(delta < 0.001)
528      break;
529
530   write_data(step);
531
532   if(acceptance_move > 0.55)
533      delta *= 1.1;
534
535   if(delta < 0.001)
536      break;
537
538   write_data(step);
539
540   if(acceptance_move > 0.55)
541      delta *= 1.1;
542
543   if(delta < 0.001)
544      break;
545
546   write_data(step);
547
548   if(acceptance_move > 0.55)
549      delta *= 1.1;
550
551   if(delta < 0.001)
552      break;
553
554   write_data(step);
555
556   if(acceptance_move > 0.55)
557      delta *= 1.1;
558
559   if(delta < 0.001)
560      break;
561
562   write_data(step);
563
564   if(acceptance_move > 0.55)
565      delta *= 1.1;
566
567   if(delta < 0.001)
568      break;
569
570   write_data(step);
571
572   if(acceptance_move > 0.55)
573      delta *= 1.1;
574
575   if(delta < 0.001)
576      break;
577
578   write_data(step);
579
580   if(acceptance_move > 0.55)
581      delta *= 1.1;
582
583   if(delta < 0.001)
584      break;
585
586   write_data(step);
587
588   if(acceptance_move > 0.55)
589      delta *= 1.1;
590
591   if(delta < 0.001)
592      break;
593
594   write_data(step);
595
596   if(acceptance_move > 0.55)
597      delta *= 1.1;
598
599   if(delta < 0.001)
600      break;
601
602   write_data(step);
603
604   if(acceptance_move > 0.55)
605      delta *= 1.1;
606
607   if(delta < 0.001)
608      break;
609
610   write_data(step);
611
612   if(acceptance_move > 0.55)
613      delta *= 1.1;
614
615   if(delta < 0.001)
616      break;
617
618   write_data(step);
619
620   if(acceptance_move > 0.55)
621      delta *= 1.1;
622
623   if(delta < 0.001)
624      break;
625
626   write_data(step);
627
628   if(acceptance_move > 0.55)
629      delta *= 1.1;
630
631   if(delta < 0.001)
632      break;
633
634   write_data(step);
635
636   if(acceptance_move > 0.55)
637      delta *= 1.1;
638
639   if(delta < 0.001)
640      break;
641
642   write_data(step);
643
644   if(acceptance_move > 0.55)
645      delta *= 1.1;
646
647   if(delta < 0.001)
648      break;
649
650   write_data(step);
651
652   if(acceptance_move > 0.55)
653      delta *= 1.1;
654
655   if(delta < 0.001)
656      break;
657
658   write_data(step);
659
660   if(acceptance_move > 0.55)
661      delta *= 1.1;
662
663   if(delta < 0.001)
664      break;
665
666   write_data(step);
667
668   if(acceptance_move > 0.55)
669      delta *= 1.1;
670
671   if(delta < 0.001)
672      break;
673
674   write_data(step);
675
676   if(acceptance_move > 0.55)
677      delta *= 1.1;
678
679   if(delta < 0.001)
680      break;
681
682   write_data(step);
683
684   if(acceptance_move > 0.55)
685      delta *= 1.1;
686
687   if(delta < 0.001)
688      break;
689
690   write_data(step);
691
692   if(acceptance_move > 0.55)
693      delta *= 1.1;
694
695   if(delta < 0.001)
696      break;
697
698   write_data(step);
699
700   if(acceptance_move > 0.55)
701      delta *= 1.1;
702
703   if(delta < 0.001)
704      break;
705
706   write_data(step);
707
708   if(acceptance_move > 0.55)
709      delta *= 1.1;
710
711   if(delta < 0.001)
712      break;
713
714   write_data(step);
715
716   if(acceptance_move > 0.55)
717      delta *= 1.1;
718
719   if(delta < 0.001)
720      break;
721
722   write_data(step);
723
724   if(acceptance_move > 0.55)
725      delta *= 1.1;
726
727   if(delta < 0.001)
728      break;
729
730   write_data(step);
731
732   if(acceptance_move > 0.55)
733      delta *= 1.1;
734
735   if(delta < 0.001)
736      break;
737
738   write_data(step);
739
740   if(acceptance_move > 0.55)
741      delta *= 1.1;
742
743   if(delta < 0.001)
744      break;
745
746   write_data(step);
747
748   if(acceptance_move > 0.55)
749      delta *= 1.1;
750
751   if(delta < 0.001)
752      break;
753
754   write_data(step);
755
756   if(acceptance_move > 0.55)
757      delta *= 1.1;
758
759   if(delta < 0.001)
760      break;
761
762   write_data(step);
763
764   if(acceptance_move > 0.55)
765      delta *= 1.1;
766
767   if(delta < 0.001)
768      break;
769
770   write_data(step);
771
772   if(acceptance_move > 0.55)
773      delta *= 1.1;
774
775   if(delta < 0.001)
776      break;
777
778   write_data(step);
779
780   if(acceptance_move > 0.55)
781      delta *= 1.1;
782
783   if(delta < 0.001)
784      break;
785
786   write_data(step);
787
788   if(acceptance_move > 0.55)
789      delta *= 1.1;
790
791   if(delta < 0.001)
792      break;
793
794   write_data(step);
795
796   if(acceptance_move > 0.55)
797      delta *= 1.1;
798
799   if(delta < 0.001)
800      break;
801
802   write_data(step);
803
804   if(acceptance_move > 0.55)
805      delta *= 1.1;
806
807   if(delta < 0.001)
808      break;
809
810   write_data(step);
811
812   if(acceptance_move > 0.55)
813      delta *= 1.1;
814
815   if(delta < 0.001)
816      break;
817
818   write_data(step);
819
820   if(acceptance_move > 0.55)
821      delta *= 1.1;
822
823   if(delta < 0.001)
824      break;
825
826   write_data(step);
827
828   if(acceptance_move > 0.55)
829      delta *= 1.1;
830
831   if(delta < 0.001)
832      break;
833
834   write_data(step);
835
836   if(acceptance_move > 0.55)
837      delta *= 1.1;
838
839   if(delta < 0.001)
840      break;
841
842   write_data(step);
843
844   if(acceptance_move > 0.55)
845      delta *= 1.1;
846
847   if(delta < 0.001)
848      break;
849
850   write_data(step);
851
852   if(acceptance_move > 0.55)
853      delta *= 1.1;
854
855   if(delta < 0.001)
856      break;
857
858   write_data(step);
859
860   if(acceptance_move > 0.55)
861      delta *= 1.1;
862
863   if(delta < 0.001)
864      break;
865
866   write_data(step);
867
868   if(acceptance_move > 0.55)
869      delta *= 1.1;
870
871   if(delta < 0.001)
872      break;
873
874   write_data(step);
875
876   if(acceptance_move > 0.55)
877      delta *= 1.1;
878
879   if(delta < 0.001)
880      break;
881
882   write_data(step);
883
884   if(acceptance_move > 0.55)
885      delta *= 1.1;
886
887   if(delta < 0.001)
888      break;
889
890   write_data(step);
891
892   if(acceptance_move > 0.55)
893      delta *= 1.1;
894
895   if(delta < 0.001)
896      break;
897
898   write_data(step);
899
900   if(acceptance_move > 0.55)
901      delta *= 1.1;
902
903   if(delta < 0.001)
904      break;
905
906   write_data(step);
907
908   if(acceptance_move > 0.55)
909      delta *= 1.1;
910
911   if(delta < 0.001)
912      break;
913
914   write_data(step);
915
916   if(acceptance_move > 0.55)
917      delta *= 1.1;
918
919   if(delta < 0.001)
920      break;
921
922   write_data(step);
923
924   if(acceptance_move > 0.55)
925      delta *= 1.1;
926
927   if(delta < 0.001)
928      break;
929
930   write_data(step);
931
932   if(acceptance_move > 0.55)
933      delta *= 1.1;
934
935   if(delta < 0.001)
936      break;
937
938   write_data(step);
939
940   if(acceptance_move > 0.55)
941      delta *= 1.1;
942
943   if(delta < 0.001)
944      break;
945
946   write_data(step);
947
948   if(acceptance_move > 0.55)
949      delta *= 1.1;
950
951   if(delta < 0.001)
952      break;
953
954   write_data(step);
955
956   if(acceptance_move > 0.55)
957      delta *= 1.1;
958
959   if(delta < 0.001)
960      break;
961
962   write_data(step);
963
964   if(acceptance_move > 0.55)
965      delta *= 1.1;
966
967   if(delta < 0.001)
968      break;
969
970   write_data(step);
971
972   if(acceptance_move > 0.55)
973      delta *= 1.1;
974
975   if(delta < 0.001)
976      break;
977
978   write_data(step);
979
980   if(acceptance_move > 0.55)
981      delta *= 1.1;
982
983   if(delta < 0.001)
984      break;
985
986   write_data(step);
987
988   if(acceptance_move > 0.55)
989      delta *= 1.1;
990
991   if(delta < 0.001)
992      break;
993
994   write_data(step);
995
996   if(acceptance_move > 0.55)
997      delta *= 1.1;
998
999   if(delta < 0.001)
1000      break;
1001
1002   write_data(step);
1003
1004   if(acceptance_move > 0.55)
1005      delta *= 1.1;
1006
1007   if(delta < 0.001)
1008      break;
1009
1010   write_data(step);
1011
1012   if(acceptance_move > 0.55)
1013      delta *= 1.1;
1014
1015   if(delta < 0.001)
1016      break;
1017
1018   write_data(step);
1019
1020   if(acceptance_move > 0.55)
1021      delta *= 1.1;
1022
1023   if(delta < 0.001)
1024      break;
1025
1026   write_data(step);
1027
1028   if(acceptance_move > 0.55)
1029      delta *= 1.1;
1030
1031   if(delta < 0.001)
1032      break;
1033
1034   write_data(step);
1035
1036   if(acceptance_move > 0.55)
1037      delta *= 1.1;
1038
1039   if(delta < 0.001)
1040      break;
1041
1042   write_data(step);
1043
1044   if(acceptance_move > 0.55)
1045      delta *= 1.1;
1046
1047   if(delta < 0.001)
1048      break;
1049
1050   write_data(step);
1051
1052   if(acceptance_move > 0.55)
1053      delta *= 1.1;
1054
1055   if(delta < 0.001)
1056      break;
1057
1058   write_data(step);
1059
1060   if(acceptance_move > 0.55)
1061      delta *= 1.1;
1062
1063   if(delta < 0.001)
1064      break;
1065
1066   write_data(step);
1067
1068   if(acceptance_move > 0.55)
1069      delta *= 1.1;
1070
1071   if(delta < 0.001)
1072      break;
1073
1074   write_data(step);
1075
1076   if(acceptance_move > 0.55)
1077      delta *= 1.1;
1078
1079   if(delta < 0.001)
1080      break;
1081
1082   write_data(step);
1083
1084   if(acceptance_move > 0.55)
1085      delta *= 1.1;
1086
1087   if(delta < 0.001)
1088      break;
1089
1090   write_data(step);
1091
1092   if(acceptance_move > 0.55)
1093      delta *= 1.1;
1094
1095   if(delta < 0.001)
1096      break;
1097
1098   write_data(step);
1099
1100   if(acceptance_move > 0.55)
1101      delta *= 1.1;
1102
1103   if(delta < 0.001)
1104      break;
1105
1106   write_data(step);
1107
1108   if(acceptance_move > 0.55)
1109      delta *= 1.1;
1110
1111   if(delta < 0.001)
1112      break;
1113
1114   write_data(step);
1115
1116   if(acceptance_move > 0.55)
1117      delta *= 1.1;
1118
1119   if(delta < 0.001)
1120      break;
1121
1122   write_data(step);
1123
1124   if(acceptance_move > 0.55)
1125      delta *= 1.1;
1126
1127   if(delta < 0.001)
1128      break;
1129
1130   write_data(step);
1131
1132   if(acceptance_move > 0.55)
1133      delta *= 1.1;
1134
1135   if(delta < 0.001)
1136      break;
1137
1138   write_data(step);
1139
1140   if(acceptance_move > 0.55)
1141      delta *= 1.1;
1142
1143   if(delta < 0.001)
1144      break;
1145
1146   write_data(step);
1147
1148   if(acceptance_move > 0.55)
1149      delta *= 1.1;
1150
1151   if(delta < 0.001)
1152      break;
1153
1154   write_data(step);
1155
1156   if(acceptance_move > 0.55)
1157      delta *= 1.1;
1158
1159   if(delta < 0.001)
1160      break;
1161
1162   write_data(step);
1163
1164   if(acceptance_move > 0.55)
1165      delta *= 1.1;
1166
1167   if(delta < 0.001)
1168      break;
1169
1170   write_data(step);
1171
1172   if(acceptance_move > 0.55)
1173      delta *= 1.1;
1174
1175   if(delta < 0.001)
1176      break;
1177
1178   write_data(step);
1179
1180   if(acceptance_move > 0.55)
1181      delta *= 1.1;
1182
1183   if(delta < 0.001)
1184      break;
1185
1186   write_data(step);
1187
1188   if(acceptance_move > 0.55)
1189      delta *= 1.1;
1190
1191   if(delta < 0.001)
1192      break;
1193
1194   write_data(step);
1195
1196   if(acceptance_move > 0.55)
1197      delta *= 1.1;
1198
1199   if(delta < 0.001)
1200      break;
1201
1202   write_data(step);
1203
1204   if(acceptance_move > 0.55)
1205      delta *= 1.1;
1206
1207   if(delta < 0.001)
1208      break;
1209
1210   write_data(step);
1211
1212   if(acceptance_move > 0.55)
1213      delta *= 1.1;
1214
1215   if(delta < 0.001)
1216      break;
1217
1218   write_data(step);
1219
1220   if(acceptance_move > 0.55)
1221      delta *= 1.1;
1222
1223   if(delta <
```

```

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

B.3

```

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