

4 Monte Carlo simulation of Hard spheres in the NVT ensemble

In this exercise we have a couple of assignment that have to be done. The web app used for plotting is found at <https://webspace.science.uu.nl/~herme107/viscol/>.

The main function in which the logic from the other functions is found is in A.7.

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.

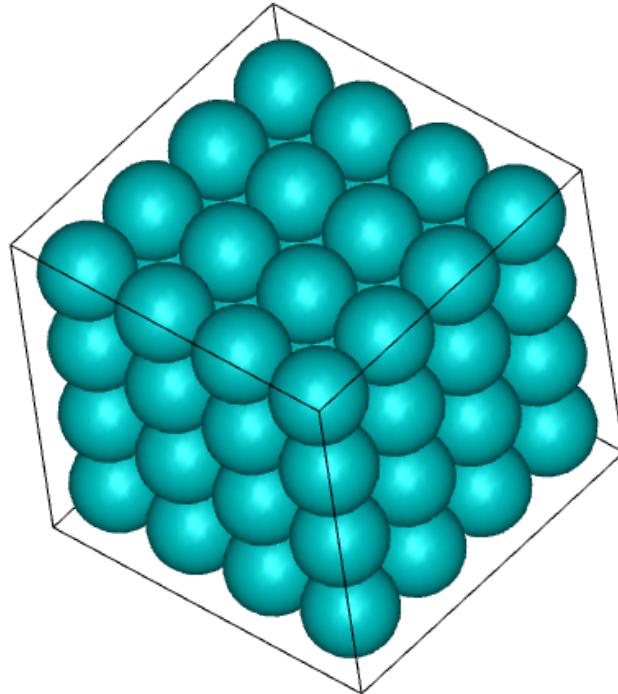


Figure 1: Here the cubic lattice generated by the code is graphed

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

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

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.

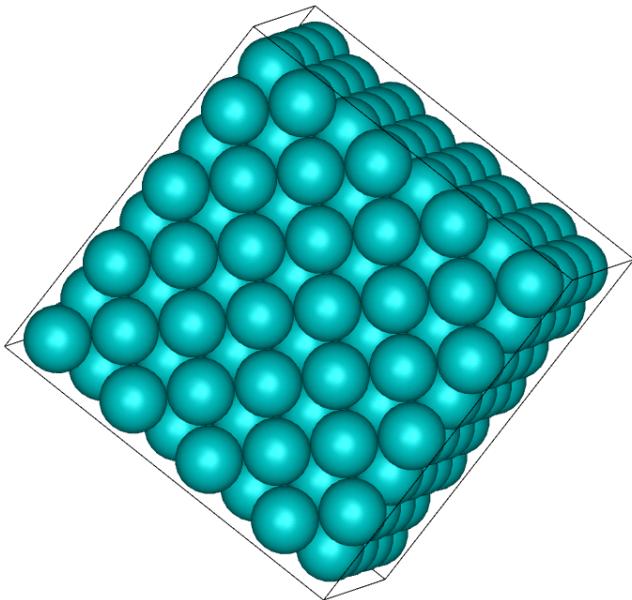


Figure 2: Here the cubic lattice generated by the code is graphed

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.3.

what we do in this part is, we first define a new data structure such that we can pass multiple things back from the reading. This structure is defined such that we can later define the number of points that the pointer should be able to count to with N found in the file. We also want to pass back the box dimensions and the size of each particle.

After this we load the file, it first reads the first line getting the number of particles. Now we can define the size of the position matrix and size vector. Then we read the dimensions of the box and place them in a matrix as well. Finally we read all the particle position and the size of each particle and close the file.

4.6

The code for this `move_particle()` subroutine is found in Appendix A.4.

Here we again define a new data structure where we can store the randomized x,y,z displacements, and the randomly selected particle.

then we just generate a four random numbers in the [-0.5,0.5] domain. Three of these are used to get a random pointing vector. This vector is normalized to have a length of one, then the last random number is scaled by $2*\Delta t$ to scale the vector to be in the [-delta, delta] domain.

One more random number is generated between [0,1] this is scaled by the number of particles and is floored to get a random particle choice.

using the function explained in the next section the validity of this translation is checked. If one is returned no overlap between particles is detected.

If the translation is valid, it is executed else it is skipped. The periodic boundary conditions are implemented here.

Keep in mind that if the packing is the most efficient it can be all the balls are touching each other and there can be no valid non overlapping translations, meaning that if we increase the spacing between the particles only then we can have valid translations.

A "good" choice for delta is expected distance between particles over 2, then we know that the chance of a non overlapping translation will be decently sized.

4.7

The code for this `check_particle_overlap()` subroutine is found in Appendix A.5.

This routine takes in the particle positions, it checks if when the translation occurs there is no overlap by seeing if the minimum distance between any two particles is smaller than the two radius's summed.

This is done with periodic boundaries by checking if the distance greater than 1/2 the length of the box, if so then the distance is changed with the length of the box.

4.8

a new function is made depicted in appendix A.6 that saves the current location of each particle in a new file with the same structure as gotten from the saved file.

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

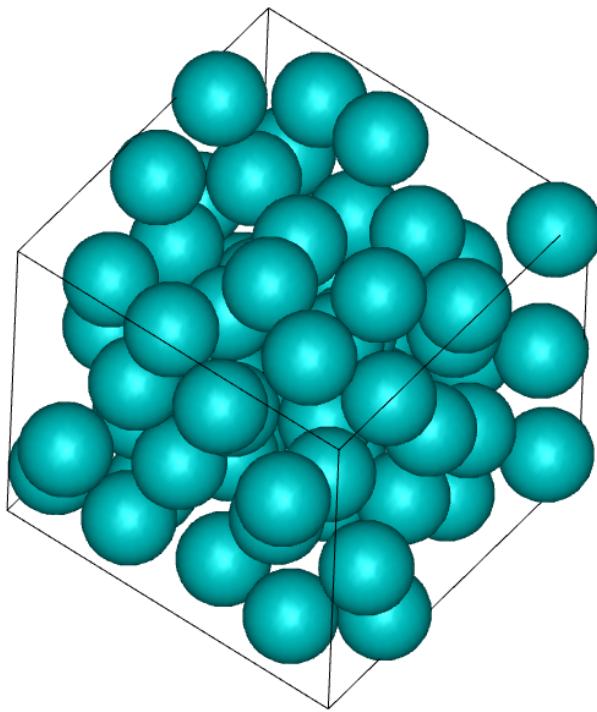


Figure 3: Here the cubic lattice generated is perturbed 100000 times to give this configuration

4.9

here we are asked to change the packing density, I will assume this is done by changing the particle size, perturbing and seeing if the structure is still fcc like.

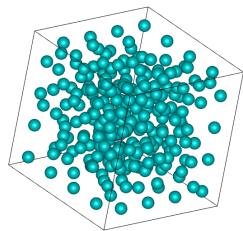
In Figure 4, configurations corresponding to different particle sizes are shown. For a particle size of 1, no translations are allowed due to the close packing, and the system therefore remains unchanged. Reducing the particle size to 0.5 leads to a clearly disordered configuration, meaning a melted phase. This means that the transition from an ordered to a disordered state occurs at an intermediate particle size.

At a particle size of 0.75, the system exhibits a perfectly ordered FCC structure. However, at 0.62 the configuration is disordered. Further refinement shows that the system remains ordered at 0.69, while at 0.65 slight disorder begins to appear.

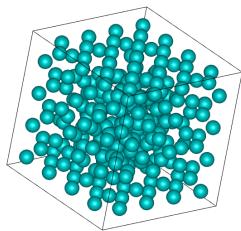
So the lattice melts somewhere between

$$\frac{\pi \cdot 0.62^3}{2\sqrt{2}} \quad \& \quad \frac{\pi \cdot 0.69^3}{2\sqrt{2}}$$

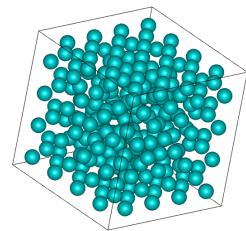
as a packing density.



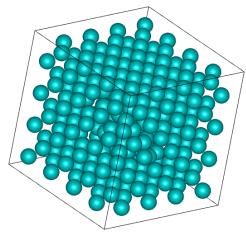
(a) particle size is 0.5



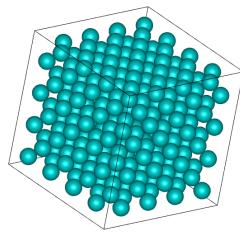
(b) particle size is 0.59



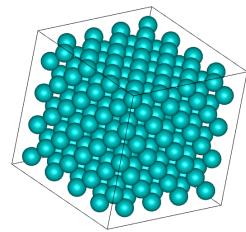
(c) particle size is 0.62



(d) particle size is 0.65



(e) particle size is 0.69



(f) particle size is 0.75

Figure 4: The packing density changed

5 MC simulation of hard spheres in the NPT ensemble

A Code Exercise 4

A.1 Code 4.a)

```
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++){
30            // 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 Code 4.c)

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

```

```

80     r[n]= a;
81
82     // saving the x,y,z possition and radius of the particle
83     fprintf(print_coords, "%lf\t%lf\t%lf\t%lf\n", x[n], y[n],z[n],r[n]);
84
85     n++;
86
87     // here we will add the a3 vector and continue the same spacing
88     x[n]=(i+0.5)*l;
89     y[n]=(j+0.5)*l;
90     z[n]=(k)*l;
91     r[n]= a;
92
93     // saving the x,y,z possition and radius of the particle
94     fprintf(print_coords, "%lf\t%lf\t%lf\t%lf\n", x[n], y[n],z[n],r[n]);
95
96     n++;
97
98
99
100
101 }
102 }
103 }
104 }
105
106
107 fclose(print_coords);
108 return 0;
109 }
```

A.3

```

1 typedef struct
2 {
3     int N; // getting the number of particles from the file
4     float (*box)[2]; //the size of the box, one direction doesnt yet have a defined size
5     float (*r)[3]; // same idea here but with the possition of the particle
6     float *size; // the number of particles
7 } Loaded_Data;
8
9 Loaded_Data load_data( char *init_filename){
10     Loaded_Data Loaded_Data;
11     int NDIM=3;
12     // degining the file
13     FILE *read_cords;
14     read_cords = fopen(init_filename, "r");
15
16     // reading the first line to get the number of of particles that exist in the file (why is
17     // the exersise so weird??)
18     fscanf(read_cords, "%i\n", &Loaded_Data.N); // the total number of particles
19     // printf("%i\n", Loaded_Data.N);
20
21     // making sure that the size will be correctly degined instead of having to asign it before
22     // hand
23     // malloc is the memory allocation commman which is wat we need to have exact size matrixes
24     // , only this satisfies me
25     Loaded_Data.box = malloc(NDIM * sizeof * Loaded_Data.box); // the size of the box (to make
26     // the particles fit inside the box poroperly 2 points are defined for me)
27     Loaded_Data.r = malloc(Loaded_Data.N * sizeof * Loaded_Data.r); // all the position vectors
28     // of all the particles
29     Loaded_Data.size = malloc(Loaded_Data.N * sizeof * Loaded_Data.size); //The size of all
30     // particles
31
32     // lets turn the above into a loop because i want to
33     for(int i = 0; i<NDIM; i++){
34         // This reads the Min into box[0][i] and Max into box[1][i]
35         fscanf(read_cords, "%f %f", &Loaded_Data.box[i][0], &Loaded_Data.box[i][1]);
36         // printf("%f %f\n", Loaded_Data.box[0][i], Loaded_Data.box[1][i]);
37     }
38
39     // now that we have arived at the paricles lets be happy
40     for(int i = 0; i<Loaded_Data.N; i++){
41         fscanf(read_cords, "%f %f %f %f", &Loaded_Data.r[i][0], &Loaded_Data.r[i][1],
42             &Loaded_Data.r[i][2], &Loaded_Data.size[i]);
43     }
44 }
```

```

37     fclose(read_cords);
38     return Loaded_Data;
39 }

```

A.4

```

1
2 typedef struct
3 {
4     int index;
5     float d_r[3];
6     float l;
7     int disp;
8 } displacement;
9
10 displacement move_particle(float Delta,Loaded_Data Loaded_Data){
11     displacement d;
12
13     d.index = floor(dsfmt_genrand()*Loaded_Data.N);
14     d.d_r[0] = (dsfmt_genrand()-0.5);
15     d.d_r[1] = (dsfmt_genrand()-0.5);
16     d.d_r[2] = (dsfmt_genrand()-0.5);
17
18     d.l = (dsfmt_genrand()-0.5)*Delta+0.0001; //no devision by 0
19
20     float length = sqrt(d.d_r[0]*d.d_r[0]+d.d_r[1]*d.d_r[1]+d.d_r[2]*d.d_r[2]);
21
22     d.d_r[0] = d.d_r[0]/length*d.l;
23     d.d_r[1] = d.d_r[1]/length*d.l;
24     d.d_r[2] = d.d_r[2]/length*d.l;
25
26
27     d.disp = check_particle_overlap(Loaded_Data, d);
28
29
30     // printf("%i\n",displacement.disp);
31     if(d.disp == 1){
32         // printf("overlap found no displacement\n");
33         return d;
34     }
35     else if (d.disp == 0)
36     {
37         // printf("from: \t%f\t%f\t%f\n",Loaded_Data.r[displacement.index][0],
38         //       Loaded_Data.r[displacement.index][1],Loaded_Data.r[displacement.index][2]);
39         Loaded_Data.r[d.index][0] += d.d_r[0];
40         Loaded_Data.r[d.index][1] += d.d_r[1];
41         Loaded_Data.r[d.index][2] += d.d_r[2];
42
43         for(int l=0; l<3;l++){
44             float box_len = abs(Loaded_Data.box[l][0]-Loaded_Data.box[l][1]) ;
45
46             if(Loaded_Data.r[d.index][l]<Loaded_Data.box[l][0]){
47                 Loaded_Data.r[d.index][l]+= box_len;
48             }
49             if(Loaded_Data.r[d.index][l]>Loaded_Data.box[l][1]){
50                 Loaded_Data.r[d.index][l] -= box_len;
51             }
52         }
53         // printf("to: \t%f\t%f\t%f\n",Loaded_Data.r[displacement.index][0],
54         //       Loaded_Data.r[displacement.index][1],Loaded_Data.r[displacement.index][2]);
55         // printf("done a displacement\n");
56
57     }
58
59
60     return d;
61 }
62
63 }

```

A.5

```

2 int check_particle_overlap(Loaded_Data l, displacement d) {
3     // getting the selected particle and adding the gotten displacement to it
4     float *p = l.r[d.index];
5     // p[0] += d.d_x;
6     // p[1] += d.d_y;
7     // p[2] += d.d_z;
8
9     // checking over all particles
10    for (int i = 0; i < l.N; i++) {
11        // if we consider the same particle the distance is always smaller then the combination of
12        // the two raduses
13        if (i == d.index) {
14            continue;
15        }
16        // getting the position of the particle
17        float *p_c = l.r[i];
18
19        // setting the distance between this particle and the changed particle to 0
20        float dist_sq = 0.0;
21        // updating this distance to be accurate
22        for (int j = 0; j < 3; j++) {
23            float diff = p[j] + d.d_r[j] - p_c[j];
24            float length = abs(l.box[j][1] - l.box[j][0]);
25            if (diff>0.5*length){
26                diff -= length;
27            }
28            else if (diff<-0.5*length){
29                diff += length;
30            }
31
32            dist_sq += diff * diff;
33        }
34
35        // making degining the minimum distance between two
36        float sum_raduses = 0.5 * (l.size[i] + l.size[d.index]);
37
38
39        if (dist_sq < sum_raduses) {
40            // printf("%f\t%f\t%f\n",d.d_r[0],d.d_r[1],d.d_r[2]);
41            return 1; // Overlap detected
42        }
43    }
44    // printf("there is no overlap\n");
45    return 0; // No overlaps found
46}

```

A.6

```

1 void write_to_file(Loaded_Data l){
2
3     FILE *print_coords; // initialises a file variable
4     char *new_name = "NVT_output.dat";
5     print_coords = fopen(new_name,"w");
6
7     fprintf(print_coords, "%i\n", l.N); // the total number of particles
8     fprintf(print_coords, "%lf\t%lf\n", l.box[0][0], l.box[0][1]); // The occupied space in the x
9         direction
10    fprintf(print_coords, "%lf\t%lf\n", l.box[1][0], l.box[1][1]); // The occupied space in the y
11        direction
12    fprintf(print_coords, "%lf\t%lf\n", l.box[2][0], l.box[2][1]); // The occupied space in the z
13        direction
14
15    for(int i=0;i<l.N;i++){
16
17        fprintf(print_coords, "%lf\t%lf\t%lf\t%lf\n", l.r[i][0], l.r[i][1],l.r[i][2],l.size[i]);
18    }
19
20    fclose(print_coords);
21
22}

```

A.7

```

1 int main(){
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19

```

```

2   dsfmt_seed(time(NULL)); //setting the seed for the random displacement
3
4   int NDIM = 3; //the number of dimmensions reading for reading out the files (you cannot
      change this to switch to 2D because you particles will overlap????)
5
6   int succes_count=0 ;
7   int mc_steps = 100000;
8   // the file that will be considerd
9   char *init_filename= "FCC_xyz.dat";
10
11 Loaded_Data Loaded_Data = load_data(init_filename);
12 printf("starting displacement loop\n");
13 for (int k=0; k< mc_steps; k++){
14
15   // printf(" run %i\n",k);
16   displacement displacement = move_particle(0.1, Loaded_Data);
17
18   if (displacement.disp == 0){
19     succes_count+=1;
20   }
21
22 }
23 printf("finnished displacement loop\n");
24
25 printf("fracction succes: %i/%i\n",succes_count,mc_steps);
26 write_to_file(Loaded_Data);
27 return 0;
28 }
```