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/`.

4.1

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

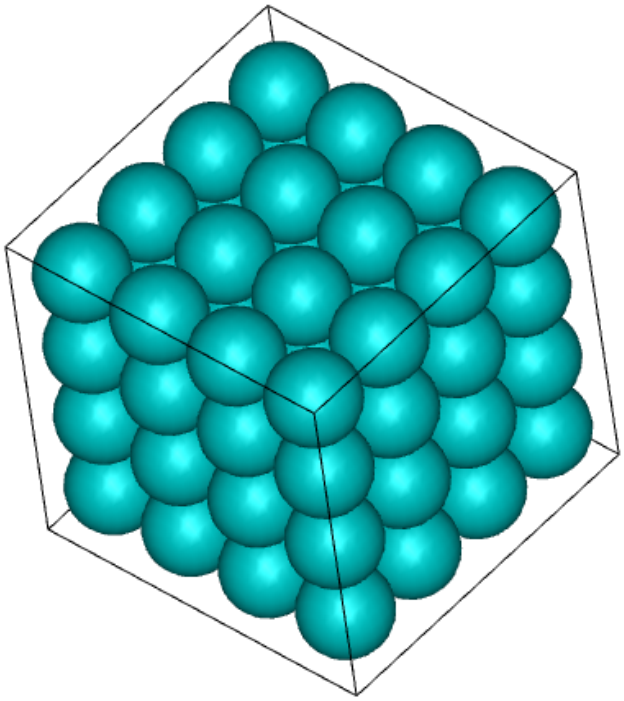


Figure 1: Here the cubic lattice generated by the code is graphed

4.2

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

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

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

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

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

4.3

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

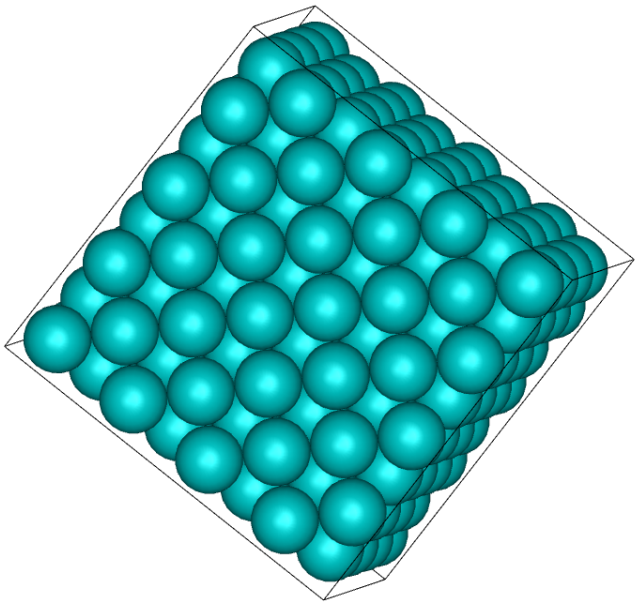


Figure 2: Here the cubic lattice generated by the code is graphed

4.4

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

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

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

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

4.5

The code for this *read_data*() subroutine is found in Appendix A.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 it is already defined with the number of dimensions (3D). 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 left 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 few random numbers and take the input delta and make sure that the total displacement is a random number between [-delta, delta].
using the function explained in the next section the validity of this translation is check. If one is returned no overlap between particles is detected.

If the translation is valid the translation is executed else it is skipped. The periodic boundary conditions are implemented here.
Keep in min 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 and also the translation of a single random particle. it translates this particle, then it checks wether this translation is valid or not by taking the distance from the particle to all the particles and seeing if the distance is at any point smaller then the radius's combined.
This is done with periodic boundaries by checking if the distance greater then 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 are plotted in Figure 3.

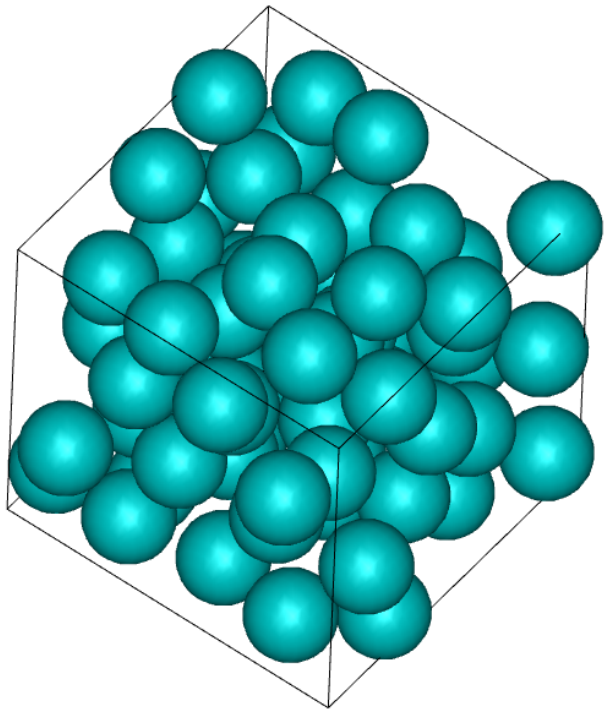
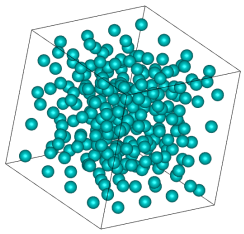


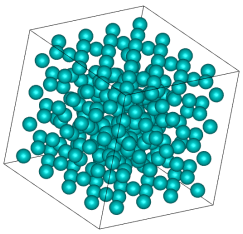
Figure 3: Here the cubic lattice generated is perturbed 100000 times to give this configuration

4.9

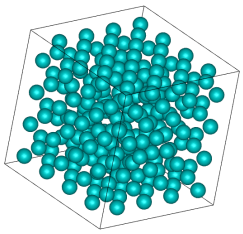
here we are asked to change the packing density, I will assume this is done by changing the particle size, perturbing and seeing if the structure is still fcc like.
In figure **??**, the figures of different particle sizes can be seen. To get to see the transition we know that if the particle size is 1 we will have no allowed translations thus no change. Now let us check a particle size of 0.5, we see that this one is chaotic, so somewhere in between there is order. Checking 0.75 gives perfect order again. Checking in between at 0.62, gives something disordered. 0.69 is still ordered while 0.65 gives a tiny bit of disorder. so the boundary between a melting FFC and an stable FFC lattice is somewhere between a packing density of $pi/(2\sqrt{2}0.69)$ and $pi/(2\sqrt{2}0.62)$



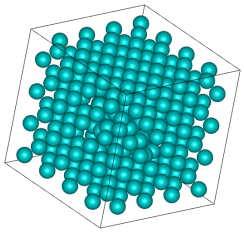
(a) particle size is 0.5



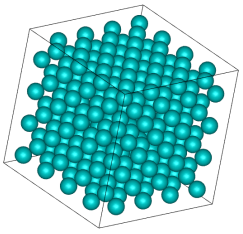
(b) particle size is 0.59



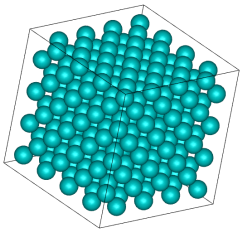
(c) particle size is 0.62



(d) particle size is 0.65



(e) particle size is 0.69



(f) particle size is 0.75

A Code Exercise 4

A.1 Code 4.a)

```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 Code 4.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

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

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

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

  // Make a file where we can save the position data
  FILE *print_coords; // inititialises a file variable
  print_coords = fopen("FCC_xyz.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", 4*N*N*N); // the total number of particles
  fprintf(print_coords, "%lf\t%lf\n", -s, x_max-sqrt(2)*s+0.5*d); // The ocupied space in the
    x direction
  fprintf(print_coords, "%lf\t%lf\n", -s, x_max-sqrt(2)*s+0.5*d); // The ocupied space in the
    y direction
  fprintf(print_coords, "%lf\t%lf\n", -s, x_max-sqrt(2)*s+0.5*d); // The ocupied space in the
    z direction

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

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

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

```
43
44
45
46    // sweeping over the N_x particles
47    for(int i=0; i<N; i++){
48      // sweeping over the N_y particles
49      for(int j=0; j<N; j++){
50        // sweeping over the N_z particles
51        for(int k=0; k<N; k++){
52          // generating the possition for i,j,k latice cite, also the radius of the particle
53
54          // first we start on the base vector because we know this patern reapeats every 2*unit
      vector in each direction
55          x[n]= (i)*l;
56          y[n]= (j)*l;
57          z[n]= (k)*l;
58          r[n]= a;
59
60          // saving the x,y,z possition and radius of the particle
61          fprintf(print_coords, "%lf\t%lf\t%lf\t%lf\n", x[n], y[n],z[n],r[n]);
62
63          n++;
64
65          // here we will add the a_1 vector and make the same spacing
66          x[n]= (i)*l;
67          y[n]= (j+0.5)*l;
68          z[n]= (k+0.5)*l;
69          r[n]= a;
70
71          // saving the x,y,z possition and radius of the particle
72          fprintf(print_coords, "%lf\t%lf\t%lf\t%lf\n", x[n], y[n],z[n],r[n]);
73
74          n++;
75
76          // here we will add the a_2 vector and make the same spacing
77          x[n]= (i+0.5)*l;
78          y[n]= (j)*l;
79          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
```

```
  float *size; // the number of particles
} Loaded_Data;

Loaded_Data load_data( char *init_filename){
  Loaded_Data Loaded_Data;
  int NDIM=3;
  // degining the file
  FILE *read_cords;
  read_cords = fopen(init_filename, "r");

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

  // making sure that the size will be correctly degined instead of having to asign it before
    hand
  // malloc is the memmory allocation commman which is wat we need to have exact size matrixes
    , only this satisfies me
  Loaded_Data.box = malloc(NDIM * sizeof * Loaded_Data.box); // the size of the box (to make
    the particles fit inside the box poroperly 2 points are defined for me)
  Loaded_Data.r = malloc(Loaded_Data.N * sizeof * Loaded_Data.r); // all the position vectors
    of all the particles
  Loaded_Data.size = malloc(Loaded_Data.N * sizeof * Loaded_Data.size); //The size of all
    particles

  // lets turn the above into a loop because i want to
  for(int i = 0; i<NDIM; i++){
    // This reads the Min into box[0][i] and Max into box[1][i]
    fscanf(read_cords, "%f %f", &Loaded_Data.box[i][0], &Loaded_Data.box[i][1]);
    // printf("%f %f\n", Loaded_Data.box[0][i], Loaded_Data.box[1][i]);
  }

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

  fclose(read_cords);
  return Loaded_Data;
}
```

A.4

```

typedef struct
{
  int index;
  float d_r[3];
  float l;
  int disp;
} displacement;

displacement move_particle(float Delta,Loaded_Data Loaded_Data){
  displacement d;

  d.index =floor(dsfmt_genrand()*Loaded_Data.N);
  d.d_r[0] = (dsfmt_genrand()-0.5);
  d.d_r[1] = (dsfmt_genrand()-0.5);
  d.d_r[2] = (dsfmt_genrand()-0.5);

  d.l = (dsfmt_genrand()-0.5)*Delta+0.0001; //no devision by 0

  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]);

  d.d_r[0] = d.d_r[0]/length*d.l;
  d.d_r[1] = d.d_r[1]/length*d.l;
  d.d_r[2] = d.d_r[2]/length*d.l;


  d.disp =  check_particle_overlap(Loaded_Data, d);


  // printf("%i\n",displacement.disp);
  if(d.disp == 1){
```

```
    // printf("overlap found no displacement\n");
    return d;
  }
  else if (d.disp == 0)
  {
    // printf("from: \t%f\t%f\t%f\n",Loaded_Data.r[displacement.index][0],
    //     Loaded_Data.r[displacement.index][1],Loaded_Data.r[displacement.index][2]);
    Loaded_Data.r[d.index][0] += d.d_r[0];
    Loaded_Data.r[d.index][1] += d.d_r[1];
    Loaded_Data.r[d.index][2] += d.d_r[2];

    for(int l=0; l<3;l++){
      float box_len = abs(Loaded_Data.box[l][0]-Loaded_Data.box[l][1]) ;

      if(Loaded_Data.r[d.index][l]<Loaded_Data.box[l][0]){
        Loaded_Data.r[d.index][l]+= box_len;
      }
      if(Loaded_Data.r[d.index][l]>Loaded_Data.box[l][1]){
        Loaded_Data.r[d.index][l] -= box_len;
      }
    }
    // printf("to: \t%f\t%f\t%f\n",Loaded_Data.r[displacement.index][0],
    // Loaded_Data.r[displacement.index][1],Loaded_Data.r[displacement.index][2]);
    // printf("done a displacement\n");


  }


  return d;
}

}
```

A.5

```

int check_particle_overlap(Loaded_Data l, displacement d) {
  // getting the selected particle and adding the gotten displacement to it
  float *p = l.r[d.index];
  // p[0] += d.d_x;
  // p[1] += d.d_y;
  // p[2] += d.d_z;

  // checking over all particles
  for (int i = 0; i < l.N; i++) {
    // if we consider the same particle the distance is always smaller then the combination of
     the two raduses
    if (i == d.index) {
      continue;
    }
    // getting the postion of the particle
    float *p_c = l.r[i];

    // setting the distance between this particle and the changed particle to 0
    float dist_sq = 0.0;
    // updating this dastance to be acurate
    for (int j = 0; j < 3; j++) {
      float diff = p[j] + d.d_r[j] - p_c[j];
      float length = abs(l.box[j][1] - l.box[j][0]);
      if (diff>0.5*length){
        diff -= length;

      }
      else if (diff<-0.5*length){
        diff += length;
      }

      dist_sq += diff * diff;
    }

    // making degining the minimum distance between two
    float sum_raduses = 0.5 * (l.size[i] + l.size[d.index]);


    if (dist_sq < sum_raduses) {
```

```
      // printf("%f\t%f\t%f\nf",d.d_r[0],d.d_r[1],d.d_r[2]);
      return 1; // Overlap detected
    }
  }
  // printf("there is no overlap\n");
  return 0; // No overlaps found
}
```

A.6

```

void write_to_file(Loaded_Data l){

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

  fprintf(print_coords, "%i\n", l.N); // the total number of particles
  fprintf(print_coords, "%lf\t%lf\n", l.box[0][0], l.box[0][1]); // The ocupied space in the x direction
  fprintf(print_coords, "%lf\t%lf\n", l.box[1][0], l.box[1][1]); // The ocupied space in the y direction
  fprintf(print_coords, "%lf\t%lf\n", l.box[2][0], l.box[2][1]); // The ocupied space in the z direction

  for(int i=0;i<l.N;i++){

    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]);
  }

  fclose(print_coords);
}
```

A.7

```
int main(){
  dsfmt_seed(time(NULL)); //setting the seed for the random displacement

  int NDIM = 3; //the number of dimmentions reading for reading out the files (you cannot
    change this to switch to 2D because you particles will overlap????)

  int succes_count=0 ;
  int mc_steps = 1000000;
  // the file that will be considerd
  char *init_filename= "cubic_xyz.dat";

  Loaded_Data Loaded_Data = load_data(init_filename);
  printf("starting displacement loop\n");
  for (int k=0; k< mc_steps; k++){

    // printf(" run %i\n",k);
    displacement displacement = move_particle(0.1, Loaded_Data);

    // printf("%i\n",displacement.disp);
    if(displacement.disp == 1){
      // printf("overlap found no displacement\n");
      continue;
    }
    else if (displacement.disp == 0)
    {
      // printf("from: \t%f\t%f\t%f\n",Loaded_Data.r[displacement.index][0],
      //     Loaded_Data.r[displacement.index][1],Loaded_Data.r[displacement.index][2]);
      Loaded_Data.r[displacement.index][0] += displacement.d_r[0];
      Loaded_Data.r[displacement.index][1] += displacement.d_r[1];
      Loaded_Data.r[displacement.index][2] += displacement.d_r[2];

      for(int l=0; l<3;l++){
        float box_len = abs(Loaded_Data.box[l][0]-Loaded_Data.box[l][1]) ;

        if(Loaded_Data.r[displacement.index][l]<Loaded_Data.box[l][0]){
          Loaded_Data.r[displacement.index][l]+= box_len;
        }
        if(Loaded_Data.r[displacement.index][l]>Loaded_Data.box[l][1]){
          Loaded_Data.r[displacement.index][l] -= box_len;
        }
```

```
      }
      // printf("to: \t%f\t%f\t%f\n",Loaded_Data.r[displacement.index][0],
      //     Loaded_Data.r[displacement.index][1],Loaded_Data.r[displacement.index][2]);
      // printf("done a displacement\n");

      succes_count+=1;
    }

  }
  printf("finnished displacement loop\n");

  printf("fracction succes: %i/%i\n",succes_count,mc_steps);
  write_to_file(Loaded_Data);
  return 0;
}
```