Particles in a Potential

Project report in Computational Physics

Inke Jürgensen and Fabian Schwartzkopff

HISKP, University of Bonn

March 2011

Contents

1 Introduction

This project work considers a set of two oppositely charged particles which are located in a finite volume. All particles interact via a two-body potential, given in equation (1). Being in a heat bath the particles start to move at some temperature.
To study the behavior of the system of particles we implement a METROPOLIS algorithm and generate new particle configurations at different temperatures. Furthermore we measure some parameters which can be used to monitor the temperature dependence of the system.
In the following we are going to describe the used algorithm as well as the results we found. The complete program is given in appendix A2.

2 Implementation of the Monte-Carlo program

2.1 The potential and the heat bath

First we introduce the two-body potential the particles interact with.

$$V_{ij} = \frac{q_i q_j}{|\vec{r}_i - \vec{r}_j|} + \frac{1}{|\vec{r}_i - \vec{r}_j|^8} \ , \tag{1}$$

where q_i and q_j are the charges and $\vec{r}_i$ denotes the vectorial position of the particles i and j respectively. The long range part is given by the COULOMB force between the particles, the short range term, which does not distinguish between charges, is assumed to be a "hard core", defining the diameter of the particles later on.
Figure 1 (p. 2) shows a plot of the two-body potential for the cases of equally and oppositely charged particles.
To determine the potential energy of the whole system one has to sum over all particles without counting any pair twice:

$$V_{\text{tot}} = \sum_{i<j} V_{ij} \ . \tag{2}$$

Since we are going to neglect the kinetic energy of the particles, equation (2) is equal to the total energy

$$E = V_{\text{tot}}$$

of the system[1]. Further we assume the system to be in a heat bath such that the partition function is given by the following equation.

$$Z = \int \prod_{i=1}^{N} \left[\mathrm{d}\vec{r}_i \exp\left(-\frac{E}{T}\right) \right] \ , \tag{3}$$

with the current temperature T and the energy E of the system.
The task now is to determine the behavior of the particles in the heat bath at fixed and changing temperatures.

[1] It is realised in the function "sum" in A2.1

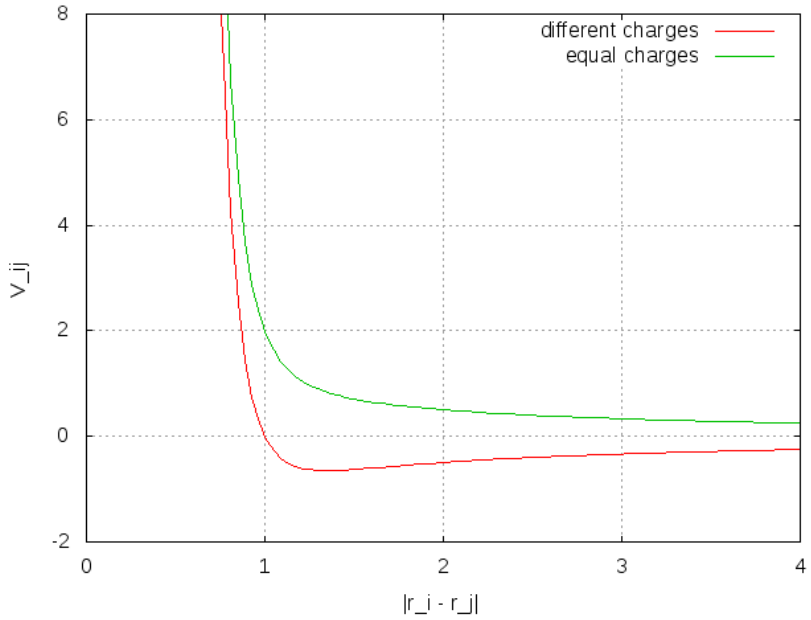


Figure 1: Two-body potential given in eq. (1) for particles with different and equal charges

2.2 Random movement of the particles

Starting the calculations we consider the A particles to build a 2 dimensional ($D = 2$) grid where the distance between next neighbours is

$$g = 1.5 \ .$$

Taking a look at the potential, one can see that it intersects the x-axis at 1. While one would consider this to be the particle diameter, some tests show that it appears to be at approximately 1.35, which turns out to be the minimum of the function. This might be caused by the the hard core term being rather sharp (see power of the denominator). Additional tests show that an initial particle lattice distance of 1.5 leads to reasonable results at low temperatures, where smaller distances, e.g. the particle diameter itself, cause the system to become too stable to move at all. This motivates the choice of this quite arbitrary value of 1.5. However, after some iterations, the particle distance reduces to a more stable value.
To save the actual position we create an array $X1$ of the size of $D * A$ where the x and y coordinates of the ith particle are written after each other[2]. We also define the charges of the particles, which are alternating for each one, $q_i \in \{-1, 1\}$. Now we want to move the particles randomly with a maximal step range d. Therefore we generate new x and y via the following formula[3]:

$$x_{i,\text{new}} = x_{i,\text{old}} + (-1)^j \cdot r \cdot d \ , \tag{4}$$

[2] i.e. $X1[0] = x_1, X1[1] = y_1, X1[2] = x_2, X1[3] = y_2, \ldots$
[3] see the function "ort" in appendix A2.1

where the i induces the x and y components. Further $r \in \mathbb{1}_{[0,1]}$ is a unitary distributed random variable and $j \in \{1, 2\}$. Each coordinate now is shifted between $+d$ and $-d$ and thus the particles can move inside a square of side length $2d$.

After moving all particles one time we calculate the energy difference ΔE between the old and the new particle configuration. The step will be accepted with the probability

$$\rho = \min\left(1, \exp\left(-\frac{\Delta E}{T}\right)\right) \ , \tag{5}$$

where the configuration will be accepted every time if $1 \leq \exp\left(-\frac{\Delta E}{T}\right)$ and with a probability of $\exp\left(-\frac{\Delta E}{T}\right)$ in the other case. If the exponent is smaller than 1 we generate a new random variable

$$\tilde{r} \in \mathbb{1}_{[0,1]}$$

and accept the new particle configuration if

$$\tilde{r} < \exp\left(-\frac{\Delta E}{T}\right) \ . \tag{6}$$

Otherwise the new positions of the particles are not accepted and the old coordinates will be shifted again[4].
In section 3.1 we present the results of moving the particles after several iterations at different, fixed temperatures.

2.3 Studying the temperature dependence

So far we are able to move the particles at a fixed temperature but the pure movement will not tell us a lot about the states the system is in. Therefore we introduce three observables which will change significantly if the system experiences a phase transition.

1. The average pair distance
2. The average distance of a particle to its next neighbour
3. The average number of particles in the neighbourhood of a particle

In the following these observables will be explained in detail.

1. Average pair distance
The first value to measure is the average pair distance. It is given by equation (7) and states the average distance between all possible particle pairs i and j.

$$R = \frac{2}{A(A-1)} \sum_{i<j} |\vec{r}_i - \vec{r}_j| \tag{7}$$

[4]For the whole program see appendix A2.4 with the head folder in A2.1

In the program this calculation is realized by adding all absolute values of the particle distances[5].

2. Average distance to the next neighbour

$$B = \frac{1}{A} \sum_{i \neq j} \min \left(|\vec{r}_i - \vec{r}_j| \right) \tag{8}$$

The average distance to the next neighbour of a particle is again calculated using the absolute value of the particle distances. In one turn all distances are calculated and compared. The smallest values for each single particle are added to each other and then normalized with the total number of particles A [6].

3. Average number of particles in the neighbourhood
Before we calculate this number we have to define the neighbourhood we want to study. Therefore we consider a circle of radius 2 around the particles.

$$|\vec{r}_i - \vec{r}_j| \leq 2 \tag{9}$$

Next we only want to consider particles which are located inside the "particle box"[7](i.e. have the maximum number of neighbours when setting up the initial lattice). Now we calculate the distances between all particle pairs and rise an integer a if the distance is smaller or equal 2. Finally we have to subtract a by one because the program also will count the distance to the particle itself. This number is calculated for all particles and then normalized by the number of particles not being at the border of the system.

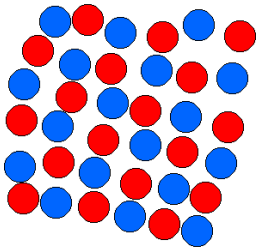
At a given temperature we now move all particles N times and calculate these three observables. To get better statistics we repeat this measurement 10 times at one temperature. After each run the particles are placed at the initial lattice again to get proper results for every run.
The program starts at an initial temperature and increases the temperature automatically to a given maximal temperature. For each temperature step the three observables are measured 10 times, as described before.

[5]see the function "dist" in appendix A2.2
[6]realized the function "NN" in A2.2
[7]we wrote a "border" function in A2.2

3 Results for the 2D two-body potential

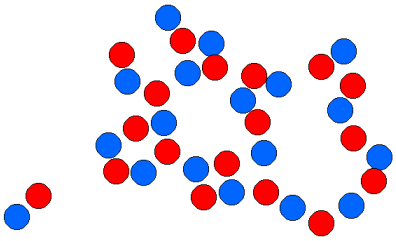
3.1 The graphical output: different phases

As described in section 2.2 the program moves the particles at a fixed temperature. In addition to the program we created a direct graphical output via the SDL engine [8]. It allows us to monitor the system with given parameters (like step range, temperature and number of particles) to find an optimal configuration of these. Having set the parameters, we can calculate the values that characterise the phase transitions. However, as told before, this graphical output alone cannot be used to find the exact temperature values of the different phases.
Next we show some significant snapshots of the SDL output which illustrate the behavior of the system at different phases.



When observing the system at low temperatures, it is in a solid state. The original lattice persists but will occasionally rotate or deform a bit. However it still stays box-like.

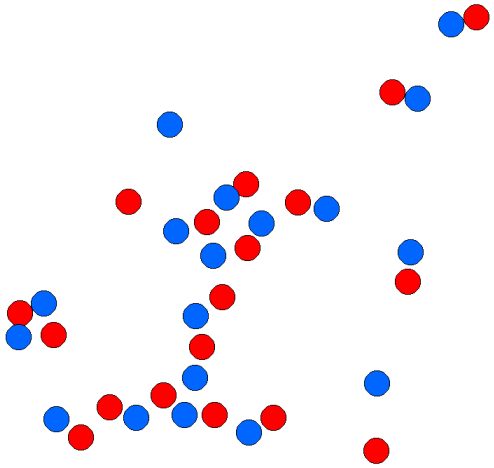
Figure 2: Solid state



Increasing the temperature, the lattice breaks open and starts to build strings of particles. The system goes into a liquid state where every particle still has a neighbour right next to it. As in a real system, some particle pairs move apart from the cluster what is equal to vaporisation.

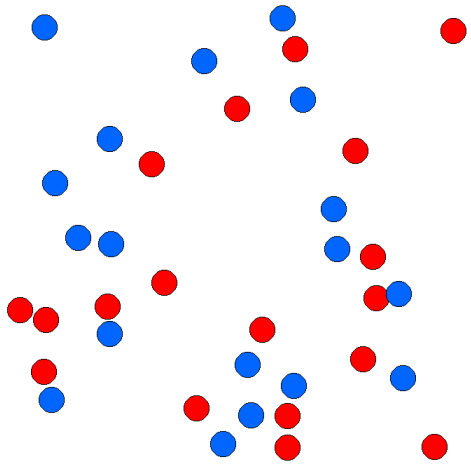
Figure 3: Liquid state

[8] http://www.libsdl.org/, see appendix A2.5 with the head folders in A2.1 and A2.3



With a rising temperature, the strings seen in the liquid state also start to break and the particles occur pairwise. One can still observe bunches or cluster like structures which are not bound over time though. This is what one calls a gaseous or molecular gas state.

Figure 4: Gaseous state



At even higher temperatures, the system goes into a saturated state where the charge of each particle can be neglected. Increasing the temperature furthermore does not change the behavior of it anymore. The gas is now completely ionized. The probability for any new random position is so high that even particles with identical charges move close to each other.

Figure 5: Plasma-like state

3.2 Locations of the phase transitions

As the transitions between the different phases cannot be located easily in the graphical output, they can be seen very well when taking a look at the observables described in section 2.3. In figure 6 all three of these observables are plotted against the logarithmical temperature.

The first phase transition from solid to liquid can be seen at the point where both, the average pair distance and the average number of particles in the neighbourhood of a particle change. As the lattice in figure 2 starts to break open, the distance between all possible pairs starts to rise while the number of particles in the neighbourhood decreases. Since there still are strings of particles (every particle has a neighbour right next to it), the distance to the next neighbour stays the same.

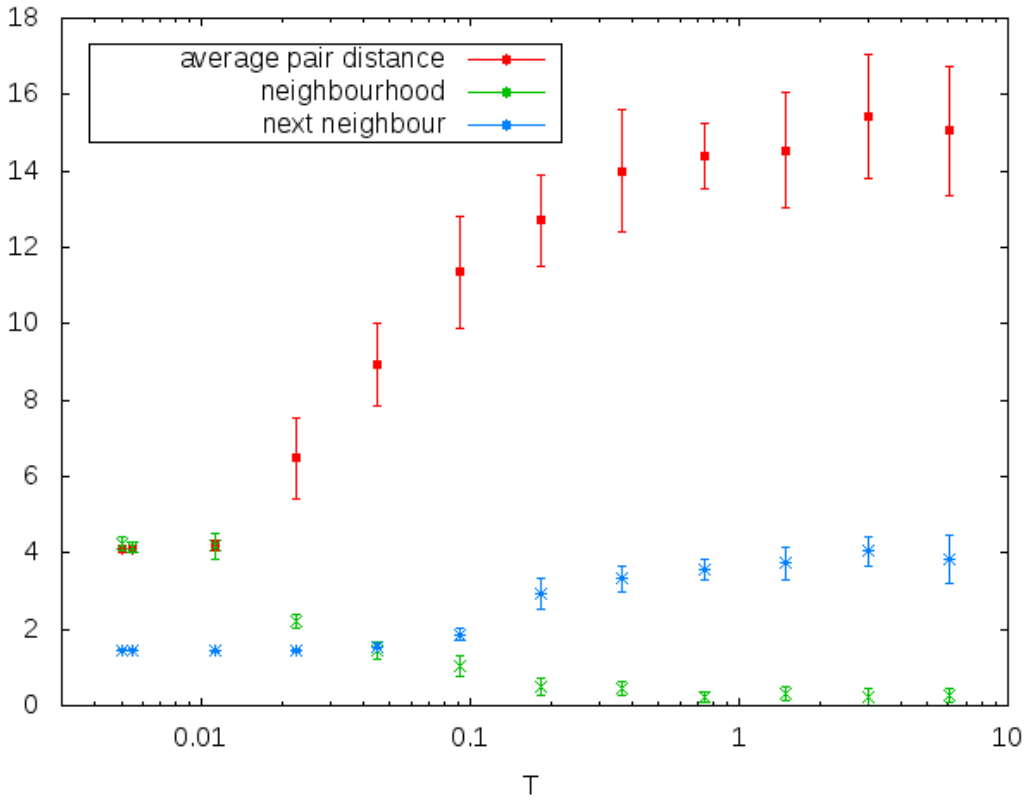


Figure 6: Average pair distance, average distance to next neighbour and average number of particles in the neighbourhood plotted against the temperature; 2-dimensional case for the two-body potential given in eq. (1)

Going into a gaseous state, the particles will occasionally move away from their partner. Thus, the distance to the next neighbour increases so that the transition from liquid to gas-like can be located at the rising point of that specific observable.
The transition from molecular to ionized gas is rather hard to locate since the phases are very similar in their behavior. Taking a look at the slope of the observables, one can see the predicted saturation value. The observables are fluctuating around it, even if the temperature is increased. Thus, the starting point of the saturated value can be seen as the last phase transition.

Regarding both, the graphical output and the plotted observables, we located the phase transitions at the values given in the following tabular.

Transition	Temperature range
crystal↔liquid	$0.01 - 0.02$
liquid↔molecular gas	$0.06 - 0.08$
molecular gas↔ionic gas	$\gtrsim 0.3$

4 Modifications of the program

4.1 3D particle system

As a first modification we generalise the system to the three dimensional case.

Modification of the functions
The global variable 'D' in the C-programs is increased from 2 to 3. Now the y coordinate in the initial grid reaches the point where the z component will be risen too. The initial two dimensional grid will be placed l times in a row, where l is the length of the lattice. Thus, the new lattice is three dimensional.
Since we constructed all loops and conditions in the program with an end-point at $D * A$, as well as the variable arrays, we do not need to change anything for the program to work. The potential stays the same due to its dependence on the absolute value of the difference between the particle positions. Since the program works the same now we are able to simulate the 3D particle system at different temperatures and we can calculate the three observables introduced in section 2.3.

Phase transitions
Before presenting the temperature dependency of the new system we take a look at a picture of the particles after they have moved some time. Figure 7 shows the system at $T = 0.03$ being in a liquid state.

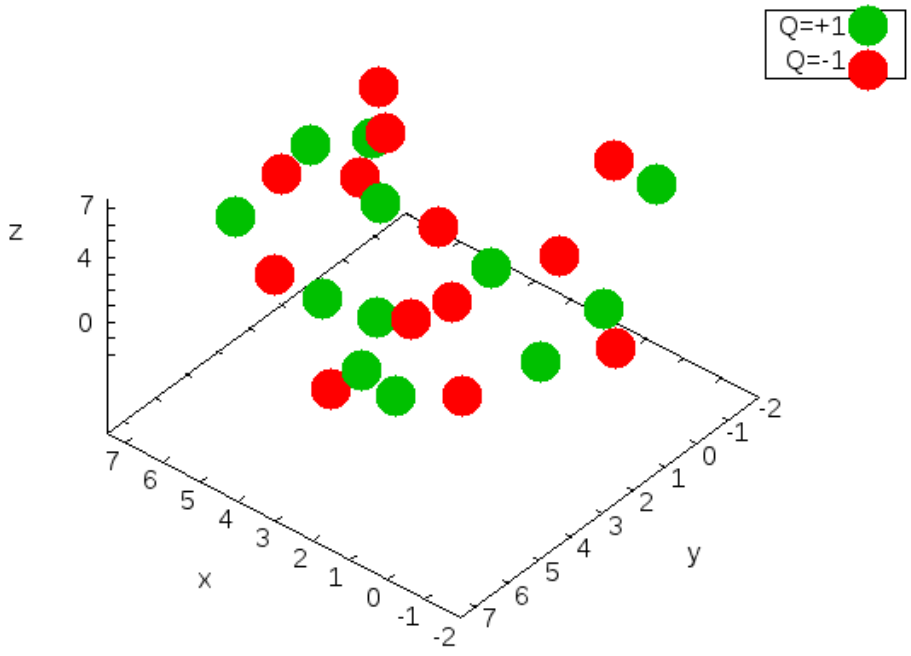


Figure 7: Screenshot of the particle system at $T = 0.03$ in a liquid state

It seems that some equally charged particles are next to each other but in fact they are not.

Due to the additional dimension, it is hard to create a clear visualisation of the system or to find a good perspective. For that reason, we skipped implementing an additional graphical output via SDL or OpenGL for the three-dimensional case.
In the picture, at the top right, one can see a pair of particles being separated from the others like in the two dimensional case, see figure 3.

To compare the predictions of the three-dimensional case with the two-dimensional one, we calculate the average pair distance, the average distance to the next neighbour and the average number of particles in the neighbourhood and plot them against the temperature (see figure 8). As described in section 2.3 we average over 10 calculations.

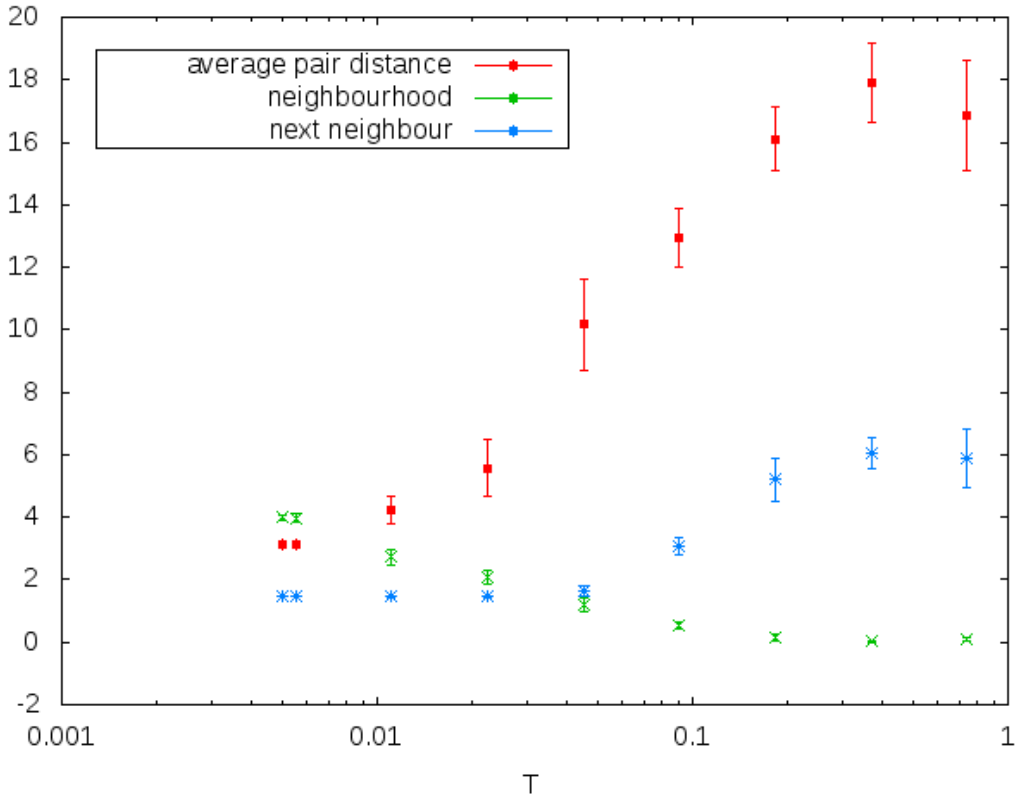


Figure 8: Average pair distance, average distance to next neighbour and average number of particles in the neighbourhood plotted against the temperature; 3-dimensional case for the two-body potential given in eq. (1)

As expected, the behavior of all three observables is the same as in the 2D case. The average pair distance increases with the temperature while the average number of particles in the neighbourhood decreases. The average distance of a particle to its next neighbour stays constant first but increases at the next phase transition.

4.2 Lennard-Jones potential

Next we want to study the behavior of particles interacting through another potential than the one introduced in eq. (1), the LENNARD-JONES potential:

$$V_{ij} = \frac{1}{|\vec{r}_i - \vec{r}_j|^{12}} - \frac{1}{|\vec{r}_i - \vec{r}_j|^{6}} \ . \tag{10}$$

The potential describes a two-body interaction of uncharged particles. The attractive part of the potential is given through VAN DER WAALS forces where the repulsive part is caused by the so called exchange interaction [9]. Figure 9 shows a plot of the LENNARD-JONES potential. As comparison the two-body potential for oppositely charged particles is shown again.

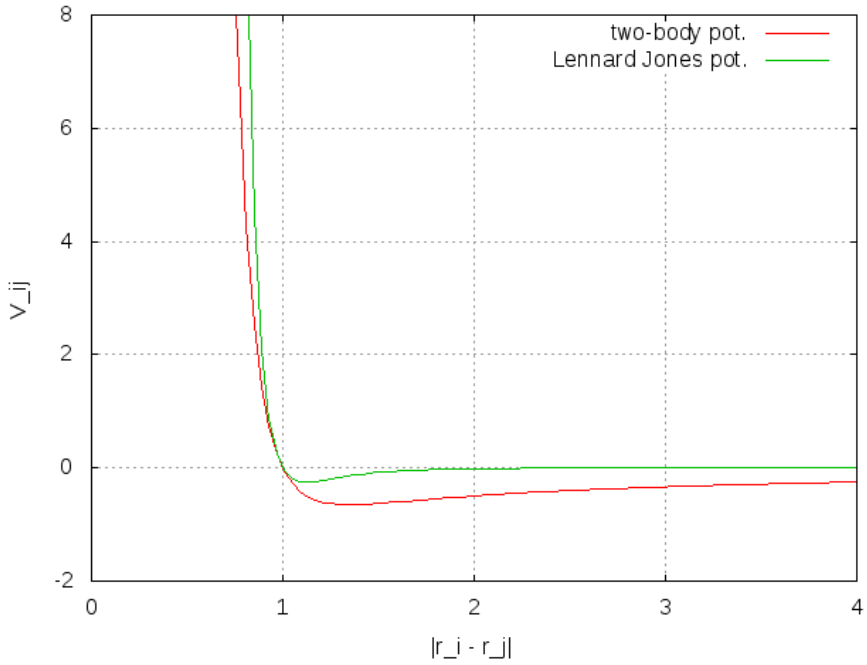


Figure 9: LENNARD-JONES potential (eq. (10)) for uncharged particles and two-body potential (eq. (1)) for oppositely charged particles

Setting the integer 'len' in the program to 1 the LENNARD-JONES potential will be used to calculate the energy of the system. The rest does not need to be changed to calculate the average pair distance, the average distance to next neighbour or the average number of particles in the neighbourhood of a particle.
Figure 10 shows the three observables plotted against the temperature. One can clearly see that the system of particles interacting via a LENNARD-JONES potential only shows one phase transition. While the next neighbour distance stays constant for the 2-body

[9] If the electron clouds of two particles overlap, fermions with same quantum numbers repel each other.

potential used before, in this case all three observables change at the same temperature scale. This leads to our assumption that the system starts in a liquid state and becomes gaseous rather fast. Since the particles do not have any charge at all, they will not stay or align in the lattice which the systems started from.

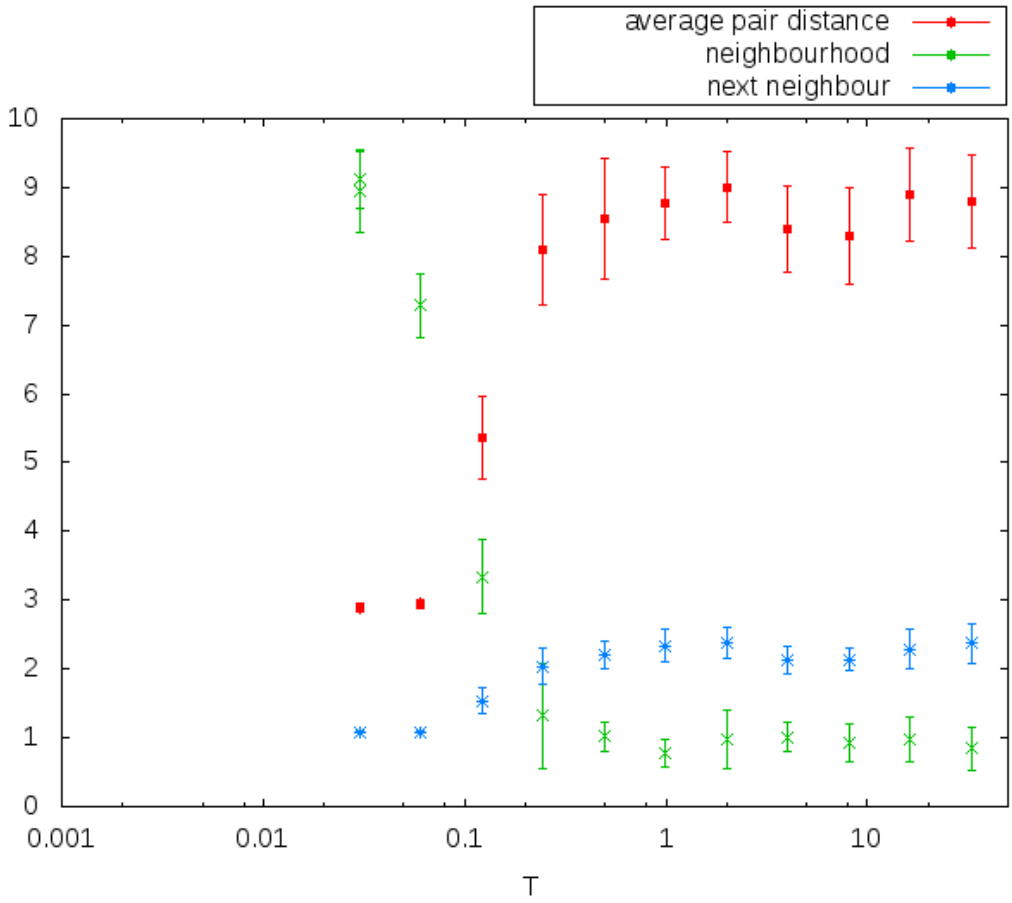


Figure 10: Average pair distance, average distance to next neighbour and average number of particles in the neighbourhood plotted against the temperature; 2-dimensional case for the Lennard-Jones potential given in eq. (10)

In appendix A1 there are two snapshots of the particles interacting via the Lennard-Jones potential. One can see the system being in a liquid (fig. 11) and a gaseous (fig. 12) state.

5 Conclusion

We simulated a system of many particles to study its behavior at different temperatures but also for different variable configurations. We implemented a direct graphical output and calculated observables to get more information about the different phases, the system's behavior in these phases and the phase transitions.
After locating ranges for these phase transitions, we regarded this behavior in a system with an additional dimension as well as in one with a different potential and compared the results of these with the original problem.
Although we first had some problems related to calculation time or graphical output, we were able to use a Metropolis algorithm to study a particle system at different temperatures and the Monte-Carlo program delivers us the expected results.

Appendix

A1 Additional screenshots

System of particles in a Lennard-Jones potential

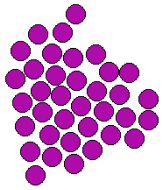


Figure 11: System in a liquid state

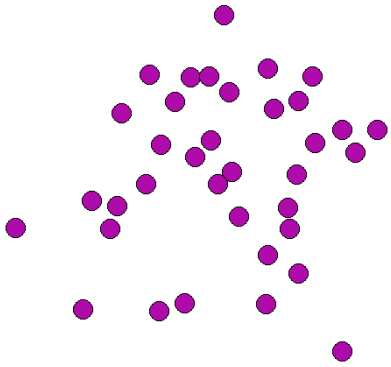


Figure 12: System in a gaseous state

A2 Monte-Carlo program

A2.1 head.h : calculate the energy

```
1  #define D 2 // dimension
2
3  // calculate the absolute value of vectorial coordinates
4  double betr(double *X1, int i, int j){
5          double b=0;
6          int l;
7          for(l=0;l<D;l++) b+=pow(X1[l+i]-X1[l+j],2);
8          return sqrt(b);
9  }
10
11 //potential for particles i and j
12 double pot(double *X, int *Q, int i, int j, int l){
13         double b,v;
14         b=betr(X,i,j);
15         v=(double)(Q[i/D]*Q[j/D])/b+pow(b,-8);
16 //lennard jones potential
17         if(l==1) v=(double)(pow(b,-12)-pow(b,-6));
18         return v;
19 }
20
21 //sum over all potentials for i<j
22 double sum(double *X, int *Q, int A, int l){
23         int i=0,j;
24         double s=0;
25         while(i<D*A){
26                 j=i+D;
27                 while(j<D*A){
28                         s+=pot(X,Q,i,j,l);
29                         j+=D;
30                 }
31                 i+=D;
32         }
33         return s;
34 }
35
36 //calculate a new (random) position, for each coordinate seperately:
37 //used for the observable#define PI 3.14159265-function
38 double ort1(double x1, double d, int l,double c,double g){
39         double r,x2=g*l+c+1;
40         while((x2>=(g*l+c))||(x2<=(1-c))){
41                 r=(double)rand()/RAND_MAX;
42                 j=rand()%2+1;
43                 x2=x1+pow(-1,j)*r*d;
44         }
45         return x2;
46 }
47
```

```
48 //used for the graphical-output-function
49 double ort2(double x1, double d, int B){
50         int j;
51         double x2=0,r;
52         while((x2>=B-1.35/2.)||(x2<=1.35/2.)){
53                 r=(double)rand()/RAND_MAX;
54                 j=rand()%2+1;
55                 x2=x1+pow(-1,j)*r*d;
56         }
57         return x2;
58 }
59
60 //calculate the energydifference between new(X2) and old(X1) positions
61 double DE(double *X1,double *X2,int *Q, int A,int l){
62         double E=sum(X2,Q,A,l)-sum(X1,Q,A,l);
63         return E;
64 }
```

A2.2 head-obs.h : calculate the observables

```
1  #define D 2 // dimension
2
3  // calculate the absolute value of vectorial coordinates
4  double betr(double *X1, int i, int j){
5          double b=0;
6          int l;
7          for(l=0;l<D;l++) b+=pow(X1[l+i]-X1[l+j],2);
8          return sqrt(b);
9  }
10
11
12 //average distance
13 double dist(double *X, int A){
14         int i=0,j=0;
15         double R=0;
16         while(i<D*A){
17                 j=i+D;
18                 while(j<D*A){
19                         R+=betr(X,i,j);
20                         j+=D;
21                 }
22                 i+=D;
23         }
24         R*=2./(A*(A-1));
25         return R;
26 }
27 //2dim border of the system
28 int border(int i,int A){
29   int j=1,l;
30   l=(int)(sqrt(A));
31   if((i==0)||(i==(l-1))||(i==(A-l))||(i==(A-1))) j=0;    //corners
```

```
32    if((i>0)&&(i<(l-1)))  j=0;                                  //lower border
33    if((i%l==0)&&(i<(A-l)))  j=0;                               //left border
34    if(((i+1)%l==0)&&(i<(A-l))&&(i>(l-1)))  j=0;                //rhight border
35    if((i>(A-l))&&(i<(A-1)))  j=0;                              //upper border
36    return j;
37  }
38
39  // calculate number of particles in the neighbourhood of particle i
40  double umf_i(double *X, int A, int i,double g){
41    int j=0;
42    double a=0,b;
43    while(j<D*A){
44      b=betr(X,i,j);
45      if(b<=2) a++;
46      j+=D;
47    }
48    a=a-1;
49    return a;
50  }
51
52  /* calculate number of particles in the neighbourhood for all particles,
53          if they are note located at the border of the system*/
54  double umf_all(double *X,int A,double g){
55    int i=0,j=0,r=0;
56    double a=0;
57    while(i<D*A){
58      if(((border(j,A)==1)&&(D==2))||(D==3)){
59        a+=umf_i(X,A,i,g);
60        r++;
61      }
62      i+=D;
63      j++;
64    }
65    a=(double)a/r;
66    return a;
67  }
68
69  //distance to the next neighbour for all particles
70  double NN(double *X, int A){
71    int i=0,j;
72    double b=0,b1,b2;
73    while(i<D*A){
74      if(i==0) j=D;
75      if(i>0) j=0;
76      b1=betr(X,i,j);
77      while(j<D*A){
78        b2=betr(X,i,j);
79        if((b2<b1)&&(i!=j)) b1=b2;
80        j+=D;
81      }
82      b+=b1;
```

```
83        i+=D;
84      }
85      b=(double)(b/A);
86      return b;
87  }
88  //calculate mean and standard deviation
89  double mean(double *A,int K, int var){
90      double zb=0,zqb=0,zbq;
91      int z;
92      for(z=0;z<K;z++){
93        zb+=A[z];
94        zqb+=A[z]*A[z];
95      }
96      zb=zb/K;
97      zqb=zqb/K;
98      zbq=zb*zb;
99      if(var==1) zb=sqrt(zqb-zbq);
100     return zb;
101 }
```

A2.3 head-sdl.h

```
1  #define WIDTH 700
2  #define BPP 4
3  #define DEPTH 32
4
5  //pixel drawing function
6  void set_pixel(SDL_Surface *surface, int x, int y, Uint32 pixel){
7          Uint8 *target_pixel = (Uint8 *)surface->pixels + y * surface->pitch
               + x * 4;
8          *(Uint32 *)target_pixel = pixel;
9  }
10
11 //white background drawing function
12 void WBackground(SDL_Surface* screen, int B){
13         int x, y;
14         for(y=0;y<B;y++){
15                 for(x=0;x<B;x++){
16                         set_pixel(screen, x, y, 0xffffffff);
17                 }
18         }
19 }
20
21 //drawing the border of the circle
22 void draw_circle(SDL_Surface *surface, int cx, int cy, int radius, Uint32
       pixel){
23         int error=-radius;
24         int x=radius;
25         int y=0;
26         while(x>=y){
27                 set_pixel(surface, cx + x, cy + y, pixel);
```

```
28                  set_pixel(surface, cx + y, cy + x, pixel);
29                  if(x!=0){
30                          set_pixel(surface, cx - x, cy + y, pixel);
31                          set_pixel(surface, cx + y, cy - x, pixel);
32                  }
33                  if(y!=0){
34                          set_pixel(surface, cx + x, cy - y, pixel);
35                          set_pixel(surface, cx - y, cy + x, pixel);
36                  }
37                  if(x!=0&&y!=0){
38                          set_pixel(surface, cx - x, cy - y, pixel);
39                          set_pixel(surface, cx - y, cy - x, pixel);
40                  }
41                  error+=y;
42                  ++y;
43                  error+=y;
44                  if(error>=0){
45                          --x;
46                          error-=x;
47                          error-=x;
48                  }
49          }
50  }
51
52  //filling the circle with a specific color
53  void fill_circle(SDL_Surface *surface, int cx, int cy, int radius, Uint32
        pixel){
54          double r=(double)radius,dy;
55          for(dy=1;dy<=r;dy+=1.0){
56                  double dx=floor(sqrt((2.0*r*dy)-(dy*dy)));
57                  int x=cx-dx;
58                  Uint8 *target_pixel_a=(Uint8 *)surface->pixels+((int)(cy+r-
                        dy))*surface->pitch+x*BPP;
59                  Uint8 *target_pixel_b=(Uint8 *)surface->pixels+((int)(cy - r
                         + dy))*surface->pitch+x*BPP;
60                  for (;x<=cx+dx;x++){
61                          *(Uint32 *)target_pixel_a=pixel;
62                          *(Uint32 *)target_pixel_b=pixel;
63                          target_pixel_a+=BPP;
64                          target_pixel_b+=BPP;
65                  }
66          }
67  }
```

A2.4 Program for the temperature dependence

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <time.h>
5  #include "head.h"
```

```
6  #include "head-obs.h"
7
8  #define PI 3.14159265
9  #define D 2 // dimension
10
11
12 int main(int argc, char **argv){
13   srand(time(NULL));
14   double x,y,z;                    //coordinates
15   double g=1.5,d=0.1;              //grating and steprange
16   int N,n=0,A,l=5,B=30;            //number of steps N & number of particles A
17   double E=-1., T=0.005,Tmax;      //energy, temperature
18   double r,t,c;                    //random variable r & 'temp. steprange' t
19   double R1,VR1,b1,Vb1,a1,Va1;     //measurements
20   int i,j,k,K,len;                 //counting integers
21   FILE *temp;                      //file to save all measurements
22     temp=fopen("temp.txt","w");
23     if(temp==NULL) return 1;
24
25
26   if(argc>5){
27     N=atoi(argv[1]);
28     T=atof(argv[2]);
29     Tmax=atof(argv[3]);
30     K=atoi(argv[4]);
31     len=atoi(argv[5]);
32   }else{
33     printf("␣steps?␣T_initial?␣T_final?␣number␣of␣calc?␣lennard␣jones?\n");
34     return 0;
35   }
36   if(len==1) g=1.0; //different lattice for lennard-jones pot.
37   t=log(T);
38   A=pow(l,D);    //number of particles depending on dimension
39   c=(B-(l-1));  //constant to create a 'border' for the system
40 //arrays for D coordinates for A particles
41   double *X1;
42         X1=malloc(D*A*sizeof(double));
43   double *X2;
44         X2=malloc(D*A*sizeof(double));
45   int *Q;
46         Q=malloc(D*A*sizeof(int));
47 //arrays for calculating behavior under tamperature change
48   double *R;
49   double *a;
50   double *b;
51         R=malloc(K*sizeof(double));
52         a=malloc(K*sizeof(double));
53         b=malloc(K*sizeof(double));
54
55 //define the charges
56   i=1;
```

```
57    j=0;
58    while(j<D*A){
59      for(beta=0;beta<D;beta++){
60           Q[j]=i;
61           j++;
62      }
63      beta=0;
64      i=i*(-1);
65    }
66
67  //move of the particles:
68    while(T<Tmax){
69      for(k=0;k<K;k++){
70            //initial (lattice) coordinates
71            x=1;
72            y=1;
73            z=1;
74            i=0;
75            while(i<D*A){
76                  if((x==g*l+1)&&(i>0)){
77                        y+=g;
78                        x=1;
79                  }
80                  if((y==g*l+1)&&(i>0)){
81                  //if D=2 this point will not be reached
82                        z+=g;
83                        y=1;
84                        x=1;
85                  }
86                  for(j=0;j<D;j++){
87                        if(j==0) X1[j+i]=x;
88                        if(j==1) X1[j+i]=y;
89                        if(j==2) X1[j+i]=z;
90                  }
91                  x+=g;
92                  i+=D;
93            }
94            // sum over n for N steps
95            while(n<N){
96  /*in one step we calculate the new position of all particles,then accept all
        new positions or not*/
97                  label1:;
98            //calculate new posititon of all particles for each coordinate
99                  for(i=0;i<D*A;i++){
100                       r=(double)rand()/(double)RAND_MAX;
101                       X2[i]=ort1(X1[i],d,l,c,g);
102                 }
103           //calculate energy difference between new and old positions
104                 E=DE(X1,X2,Q,A,len);
105           //accepting with prob. 1
106                 if(1<exp(-E/T)){
```

```
107                             for(i=0;i<D*A;i++){
108                                     X1[i]=X2[i];
109                             }
110                     }else{
111             //accepting with prob. r
112                             r=(double)rand()/(double)RAND_MAX;
113                             if(r<exp(-E/T)){
114                                     for(i=0;i<D*A;i++){
115                                             X1[i]=X2[i];
116                                     }
117                             }else{
118                                     if(T>0) goto label1;
119                             }
120
121                     }
122                     n++;
123             }
124             n=0;
125             R[k]=dist(X1,A);
126             a[k]=umf_all(X1,A,g);
127             b[k]=NN(X1,A);
128       }
129       R1=mean(R,K,0);
130       VR1=mean(R,K,1);
131       a1=mean(a,K,0);
132       Va1=mean(a,K,1);
133       b1=mean(b,K,0);
134       Vb1=mean(b,K,1);
135       fprintf(temp,"%f %f %f %f %f %f\n",T,R1,VR1,a1,Va1,b1,Vb1);
136
137       t+=0.7;
138       T=exp(t);
139     }
140
141     void free(void *X1);
142     void free(void *X2);
143     void free(void *R);
144     void free(void *a);
145     void free(void *b);
146
147     fclose(temp);
148
149     FILE *ts;               // file for gnuplot plotting the measurement
150           ts=fopen("ph-sk.txt","w");
151           if(ts==NULL) return 1;
152     fprintf(ts,"set xlabel 'T'\n"
153       "set ylabel ''\n"
154       "set xrange [0.001:%f]\n"
155       "set autoscale y\n"
156       "set title ''\n"
157       "set logscale x\n"
```

```
158       "set pointsize 1\n"
159       "set key outside bottom\n"
160       "plot 'temp.txt' u 1:2:3 w yerrorbars title 'average pair distance', '
              temp.txt' u 1:4:5 w yerrorbars title 'neighbourhood', 'temp.txt' u
              1:6:7 w yerrorbars title 'next neighbour'",Tmax);
161     fclose(ts);
162     printf("\n wrote a skript for the average distance: 'ph-sk.txt'\n");
163     void free(void *Q);
164
165
166     return 0;
167
168   }
```

A2.5 Main function of the SDL graphical output

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <math.h>
 4 #include <time.h>
 5 #include "SDL/SDL.h"
 6 #include "head.h"
 7 #include "head-sdl.h"
 8
 9 #define WIDTH 700
10 #define BPP 4
11 #define DEPTH 32
12 #define DEG2RAD PI/180
13 #define PI 3.14159265
14 #define D 2
15
16 int main(int argc, char **argv){
17   SDL_Init(SDL_INIT_EVERYTHING);
18   double E=-1.,T,r,g,d,x,y,c,p,z;
19   int keypress=0,N,n=0,i,j,A,B,l,q,potent;
20   char *key;
21   SDL_Event event;
22   srand(time(NULL));
23   //reading in the variables
24   if(argc>7){
25     l=atoi(argv[1]);
26     B=atoi(argv[2]);
27     d=atof(argv[3]);
28     T=atof(argv[4]);
29     N=atoi(argv[5]);
30     z=atof(argv[6]);
31     potent=atoi(argv[7]);
32   }else{
33     printf("Lattice length - Boundary length - Steprange - Temperature -
           Number of iterations - Temperature changing parameter - Lennard
           potential?\n");
```

```
34      return 0;
35    }
36  //number of particles
37    A=l*l;
38  //initial lattice distance
39    if(potent==0){
40      g=1.5;
41    }else{
42      g=1;
43    }
44  //space between boundary and lattice
45    c=(B-(l-1)*g);
46  //lattice length+particle diameter must not exceed the boundary
47    if(l*g>=B-1.35){
48      printf("Lattice too wide! %lf\n",B-WIDTH/B*1.35);
49      return 1;
50    }
51  //initialisation of the graphical output
52    if (SDL_Init(SDL_INIT_VIDEO) != 0){
53      return 1;
54    }
55    atexit(SDL_Quit);
56  //setting the graphical output screen
57    SDL_Surface *screen = SDL_SetVideoMode(WIDTH, WIDTH, 0, SDL_DOUBLEBUF);
58    SDL_WM_SetCaption("Particles in a potential", "Particles in a potential");
59    printf("\n\nWelcome to Particles in a Potential!\nPress Esc to quit, +,1
        or 3 to increase and -,2 or 4 to decrease the temperature. Have fun. ;)
        \n\nThe starting temperature is %.3lf\n",T);
60  //allocating the location arrays and charge array
61    double *X1;
62    X1=malloc(D*A*sizeof(double));
63    double *X2;
64    X2=malloc(D*A*sizeof(double));
65    int *Q;
66    Q=malloc(A*sizeof(int));
67   //defining the charges
68    j=0;
69    if(l%2==0){
70      for(i=0;i<A;i++){
71          Q[i]=pow(-1,i+j);
72          if(((i+1)%l==0)&&(i>0)) j++;
73      }
74    }else{
75      for(i=0;i<A;i++){
76          Q[i]=pow(-1,i);
77      }
78    }
79  //initial grid beginning and writing
80    x=c/2.;
81    y=c/2.;
82    i=0;
```

```
83    while(i<D*A){
84      if((x==g*l+c/2.)&&(i>0)){
85            y+=g;
86            x=c/2.;
87      }
88      for(j=0;j<D;j++){
89            if(j==0){
90                  X1[j+i]=x;
91                  X2[j+i]=X1[j+i];
92            }else{
93                  X1[j+i]=y;
94                  X2[j+i]=X1[j+i];
95            }
96      }
97      x+=g;
98      i+=D;
99    }
100
101   while(1){
102     while(n<N){
103 //calculating the new position of all particles
104           for(i=0;i<D*A;i+=2){
105                   X2[i]=ort2(X1[i],d,B);
106                   X2[i+1]=ort2(X1[i+1],d,B);
107           }
108           //calculating energy difference between new and old positions
109                   E=DE(X1,X2,Q,A,potent);
110                   //accept-reject method
111                   if(1.<exp(-E/T)){
112                     for(i=0;i<D*A;i++){
113                       X1[i]=X2[i];
114                     }
115                   }else{
116                     r=(double)rand()/(double)RAND_MAX;
117                     if(r<exp(-E/T)){
118                       for(i=0;i<D*A;i++){
119                             X1[i]=X2[i];
120                       }
121                     }
122                   }
123                   n++;
124     }
125 //locking the screen and drawing the particles
126     SDL_LockSurface(screen);
127     WBackground(screen, WIDTH);
128     for(i=0;i<D*A;i+=2){
129           if(potent==1){
130             fill_circle(screen, (int)(WIDTH/B*X1[i]), (int)(WIDTH/B*X1[i+1]),
                  (int)(WIDTH/B*0.5), 0xffaf0aaa);
131             draw_circle(screen, (int)(WIDTH/B*X1[i]), (int)(WIDTH/B*X1[i+1]),
                  (int)(WIDTH/B*0.5), 0xff000000);
```

```
132           }else{
133             if(Q[i/2]<0){
134               fill_circle(screen, (int)(WIDTH/B*X1[i]), (int)(WIDTH/B*X1[i+1])
                      , (int)(WIDTH/B*1.35/2), 0xffff0000);
135               draw_circle(screen, (int)(WIDTH/B*X1[i]), (int)(WIDTH/B*X1[i+1])
                      , (int)(WIDTH/B*1.35/2), 0xff000000);
136             }else{
137               fill_circle(screen, (int)(WIDTH/B*X1[i]), (int)(WIDTH/B*X1[i+1])
                      , (int)(WIDTH/B*1.35/2), 0xff0066ff);
138               draw_circle(screen, (int)(WIDTH/B*X1[i]), (int)(WIDTH/B*X1[i+1])
                      , (int)(WIDTH/B*1.35/2), 0xff000000);
139             }
140           }
141       }
142       SDL_FreeSurface(screen);
143       SDL_Flip(screen);
144       n=0;
145       while(SDL_PollEvent(&event)){
146   //defining the keys to control the program
147         switch (event.type){
148           case SDL_QUIT:
149           exit(1);
150           break;
151           case SDL_KEYDOWN:
152           key=SDL_GetKeyName(event.key.keysym.sym);
153           if(event.key.keysym.sym==SDLK_ESCAPE){
154             exit(1);
155           }
156           if(event.key.keysym.sym==SDLK_PLUS){
157             T+=z;
158             printf("Temperature increased to %.3lf\n",T);
159           }
160           if(event.key.keysym.sym==SDLK_MINUS){
161             T=T-z;
162             if(T<=0){
163                   T+=z;
164                   printf("Sorry, no negative temperatures...\nTemperature at
                          %.3lf\n",T);
165             }else{
166                   printf("Temperature decreased to %.3lf\n",T);
167             }
168           }
169           if(event.key.keysym.sym==SDLK_1){
170             T+=z*10;
171             printf("Temperature increased to %.3lf\n",T);
172           }
173           if(event.key.keysym.sym==SDLK_2){
174             T=T-z*10;
175             if(T<0){
176                   T+=z*10;
```

```
177                     printf("Sorry, no negative temperatures...\nTemperature at
                            %.3lf\n",T);
178             }else{
179                     printf("Temperature decreased to %.3lf\n",T);
180             }
181           }
182           if(event.key.keysym.sym==SDLK_3){
183             T+=z*50;
184             printf("Temperature increased to %.3lf\n",T);
185           }
186           if(event.key.keysym.sym==SDLK_4){
187             T=T-z*50;
188             if(T<0){
189                     T+=z*50;
190                     printf("Sorry, no negative temperatures...\nTemperature at
                            %.3lf\n",T);
191             }else{
192                     printf("Temperature decreased to %.3lf\n",T);
193             }
194           }
195        }
196       }
197    }
198    void free(void *X1);
199    void free(void *X2);
200    void free(void *Q);
201
202
203    return 0;
204
205 }
```