PHYS580 Homework 5

Yicheng Feng
PUID: 0030193826

November 16, 2019

Workflow: I use the Linux system, and code C++ in terminals. For visualization, I use the C++ package – ROOT (made by CERN) to make plots. At the end, the reports are written in LaTeX.

The codes for this lab are written as the following files:

- `ising_model_2d.h` and `ising_model_2d.cxx` for the class `IsingModel2D` to simulate the 2D Ising model.
- `molecular_dynamics_2d.h` and `molecular_dynamics_2d.cxx` for the class `IsingModel2D` to simulate the 2D molecular dynamics.
- `hw5.cxx` for the main function to make plots.

To each problem of this lab report, I will attach the relevant parts of the code. If you want to check the validation of my code, you need to download the whole code from the link `https://github.com/YichengFeng/phys580/tree/master/hw5`.

(1) *[Problem 8.3 (p.257). It is enough if you calculate either for the square grid,* ***or*** *the triangle one (the latter takes a little more thought). Doing both cases is optional.]*
Calculate M for the Ising model on a square lattice and try to estimate β. You should find a value close to 1/8. Repeat this calculation for a triangle lattice. It turns out that β is the same for all regular two dimensional lattices. However, its value does depend on the dimensionality, as studied in the next problem.

Physics explanation:

The lattice is a 50×50 square (the triangle case is not included). The simulation time (Monte Carlo steps, MCS) is from 0 to 4000. To ensure the initial transient is finished, we only count the observables after $t = 1000$. The temperature scan is from $T = 2.0$ to $T = 2.8$, and the step size is $\Delta T = 0.01$.

In each temperature T scan, we calculate the mean vale and fluctuation of magnetization per spin (M) and energy per spin (E). Then, the specific energy can also be calculated from the energy fluctuation $C = \langle \Delta E^2 \rangle / k_B T^2$. The peak position of C indicates the critical temperature T_C, so we can get the accurate T_C of this specific configuration. (Figure 1 gives $T_C \approx 2.34$.)

As we have known T_C, we can still use the power low fitting: slope of a plot of $\log(M)$ versus $\log(T_C - T)$. Finally, β could be gotten.

The relationship between M and T is shown in Fig. 2. The left pad shows the M–T curve, while the right pad shows the $\log(M)$–$\log|T - T_C|$ plot. We use the power law fitting to get β from the right pad. The slope of the straight fitting gives an estimation $\beta \approx 0.1319$, whereas the exact value is $\beta = 0.125$. The relative difference is 5.5%, so the simulation result is quite close to the exact value.

Plots:

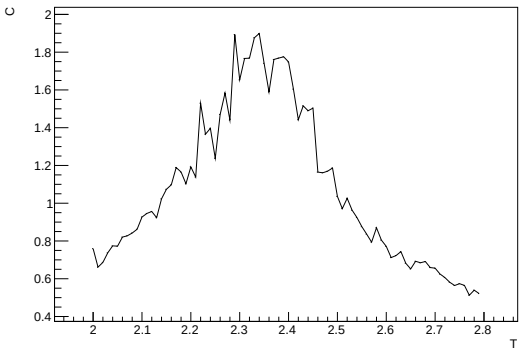

Figure 1: The relationship between C and T is shown above. The critical temperature T_C is estimated by the position of maximum C, which is $T_C \approx 2.34$.

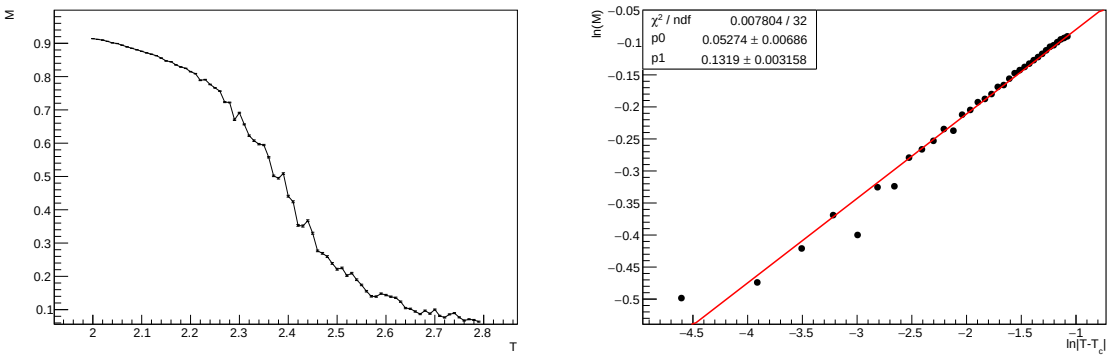

Figure 2: The relationship between M and T is shown above. The left pad shows the M–T curve, while the right pad shows the $\log(M)$–$\log|T - T_C|$ plot. We use the power law fitting to get β from the right pad.

Relevant code:

For the simulation of the 2D Ising model

```
//------------------------------------------------------------------------//

IsingModel2D::IsingModel2D(int N, double T, double H, int MCS, int c) {

        _N = N;
        _T = T;
        _H = H;
        _MCS = MCS;
        _c = c; // (1: all up; -1: all down; 2: random)

        _now = 0;

        _spin.clear();
        for(int i=0; i<_N; i++) {
                vector<int> tmp;
                for(int j=0; j<_N; j++) {
                        int one_spin;
                        if(_c == 1) {
                                one_spin = 1;
                        } else if(_c == -1) {
                                one_spin = -1;
                        } else {
                                one_spin = 1.0*rand()/RAND_MAX<0.5?-1:1;
                        }
                        tmp.push_back(one_spin);
                }
                _spin.push_back(tmp);
        }

        _m = 0;
        _E = 0;
        for(int i=0; i<_N; i++) {
                int il = i-1<  0? i-1+_N:i-1;
                int ir = i+1>=_N? i+1-_N:i+1;
                for(int j=0; j<_N; j++) {
                        int jl = j-1<  0? j-1+_N:j-1;
                        int jr = j+1>=_N? j+1-_N:j+1;

```

```
                        _m += _spin[i][j];
                        double sum = _spin[il][j] + _spin[ir][j] + _spin[i][jl] + _spin[i][jr];
                        _E -= _spin[i][j]*(_H + 0.5*sum);
                }
        }
}

//------------------------------------------------------------------------//

void IsingModel2D::cal_once() {

        for(int i=0; i<_N; i++) {
                int il = i-1<  0? i-1+_N:i-1;
                int ir = i+1>=_N? i+1-_N:i+1;
                for(int j=0; j<_N; j++) {
                        int jl = j-1<  0? j-1+_N:j-1;
                        int jr = j+1>=_N? j+1-_N:j+1;

                        double sum = _spin[il][j] + _spin[ir][j] + _spin[i][jl] + _spin[i][jr];
                        double DeltaE = 2*_spin[i][j]*(sum + _H);
                        if(DeltaE<0 || exp(-DeltaE/_T)>1.0*rand()/RAND_MAX) {
                                _spin[i][j] = -_spin[i][j];
                                _m += 2*_spin[i][j];
                                _E += DeltaE;
                        }
                }
        }
}

//------------------------------------------------------------------------//

void IsingModel2D::cal_until(int t) {

        if(!check()) return;

        for(int i=_now; i<t; i++) {
                cal_once();
        }

        _now = t;
}
```

(2) *[Problem 8.7 (p.258)]*
Obtain the specific heat as a function of temperature for a 10×10 square lattice by differentiating the energy and through the fluctuation-dissipation theorem. Show that the two methods give the same result. Which approach is more accurate (for a given amount of computer time)?

Physics explanation:

The lattice is a 10×10 square. The simulation time (Monte Carlo steps, MCS) is from 0 to 3000. To ensure the initial transient is finished, we only count the observables after $t = 1000$. The temperature scan is from $T = 1.0$ to $T = 4.0$, and the step size is $\Delta T = 0.015$.

There are two methods are used in calculating the specific heat C per spin (fixed volumn).

- From the differentiation of energy per spin E (definition), it has

$$C = \frac{\mathrm{d}E}{\mathrm{d}T}. \tag{1}$$

  In the numerical calculation, we use $(E_{i+1} - E_{i-1})/(2\Delta T)$ to estimate this differentiation at T_i except for the two edges.

- From the fluctuation-dissipation theorem, it has

$$C = \frac{\langle \Delta E^2 \rangle}{k_B T^2}. \tag{2}$$

  In the actual numerical calculation, the quantities (T, E, C) are re-scaled to unitless, so we don't need to take care of the Boltzmann constant k_B here. For ΔE, we need to notice that E for each T is already the average over N^2 spins, so we need to multiply N^2 with the variance calculated directly from E.

Figure 3 shows the specific heat per spin (C) depends on temperature (T). The left pad shows the specific heat C calculated from the differentiation of energy per spin E. The right pad shows the specific heat C calculated from the fluctuation of energy per spin E. They are calculated from the same simulation, so the computer times for them are the same. We can see the two curves have the similar shapes, trends and mean values, but the fluctuation-dissipation theorem (right pad) gives a much more accurate C.

Plots:

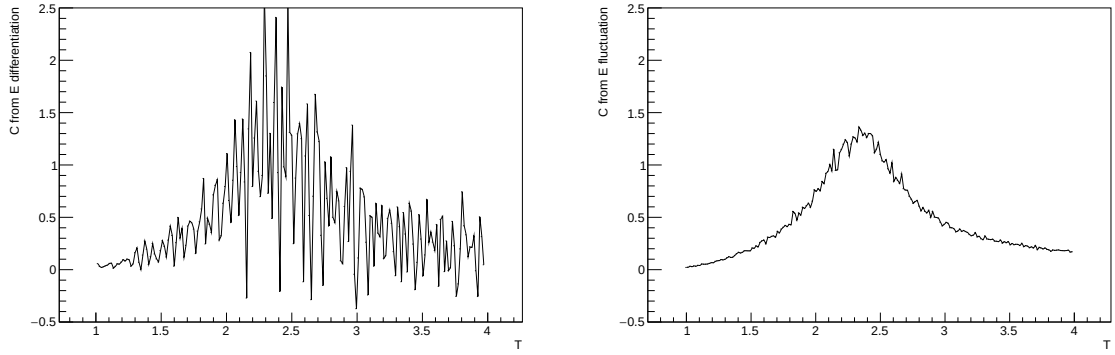

Figure 3: The two methods to calculate the specific heat C depending on temperature T. The left pad shows the specific heat C calculated from the differentiation of energy per spin E. The right pad shows the specific heat C calculated from the fluctuation of energy per spin E.

Relevant code:

For the simulation code, it has already shown in problem (1).

For the two approaches to calculate C

```
//-------------------------------------------------------------------------//

void ising_model_2d_specific_heat() {

        const int n = 200;
        const double Tmax = 4;
        const double Tmin = 1;
        const double dT = (Tmax - Tmin)/n;

        double T[n];
        double Te[n] = {0};
        double M[n];
        double Me[n];
        double Ms[n];
        double E[n];
        double Ee[n];
        double Es[n];
        double C_fluctuation[n];
        double C_differentiation[n];

        int t_max = 3000;
        int t_stb = 1000;
        int N = 10;
        double DeltaE = 0;

        for(int i=0; i<n; i++) {
                double Ttmp = Tmin + dT*i;
                T[i] = Ttmp;
                vector<double> v_tmp = ising_model_2d_plot(N, Ttmp, 0.00, t_max, t_stb, false);
                M[i]  = v_tmp[0];
                Me[i] = v_tmp[1];
                Ms[i] = v_tmp[1]*sqrt(t_max-t_stb);
                E[i]  = v_tmp[2];
                Ee[i] = v_tmp[3];
                Es[i] = v_tmp[3]*sqrt(t_max-t_stb);
```

```
36                  C_fluctuation[i]  = N*N*Es[i]*Es[i]/Ttmp/Ttmp;
37          }
38
39          for(int i=0; i<n; i++) {
40                  if(i==0)   C_differentiation[i] = (E[i+1]-E[i])/dT;
41                  if(i==n-1) C_differentiation[i] = (E[i]-E[i-1])/dT;
42                  C_differentiation[i] = (E[i+1]-E[i-1])/dT/2;
43                  if(!isnan(C_fluctuation[i])) {
44                          DeltaE += C_fluctuation[i];
45                  } else {
46                          cout << i << endl;
47                  }
48          }
49          DeltaE *= dT;
50
51          // ... plot code ...
52  }
53
54  //------------------------------------------------------------------------//
```

(3) *[Problem 8.15 (p.267)]*

Scaling behavior is found for thermodynamic quantities other than the magnetization. Calculate the susceptibility χ at various of T and H around the critical point of the Ising model on a square lattice, and study data collapsing using your results. The scaling form for χ is

$$\chi(t,h) = |t|^{-\gamma} g_{\pm}\left(\frac{h}{|t|^{\beta\delta}}\right), \tag{3}$$

where the critical exponent is $\gamma = 7/4$.

Physics explanation:

From the textboob, we know that for the 2D Ising model $\beta = 1/8$, $\delta = 15$, and $\gamma = 7/4$.

The lattice is a 20×20 square (the triangle case is not included). The simulation time (Monte Carlo steps, MCS) is from 0 to 3000. To ensure the initial transient is finished, we only count the observables after $t = 1000$. The temperature scan is from $T = 1.5$ to $T = 3.0$, and the step size is $\Delta T = 0.015$.

The critical temperature T_C is calculated from the peak position of C.

In Fig. 4, the left pad shows the magnetization per spin m depending on temperature T with various input H. As H becomes closer to *zero* from positive, m becomes smaller at high temperature. The right pad shows the scaled depending on scaled $h/|t|^{15/8}$ with various input H. We can see that the curves with various H coincident with each other. There are two branches indicates $T > T_C$ and $T < T_C$.

Figure 5 shows the scaled $m/|t|^{1/8}$ depends on scaled $\chi|t|^{7/4}$ with various input H, where

$$\chi = \frac{(\Delta M)^2}{k_B T}. \tag{4}$$

We can see that the curves with various H coincident with each other. There are two branches indicates $T > T_C$ and $T < T_C$.

Plots:

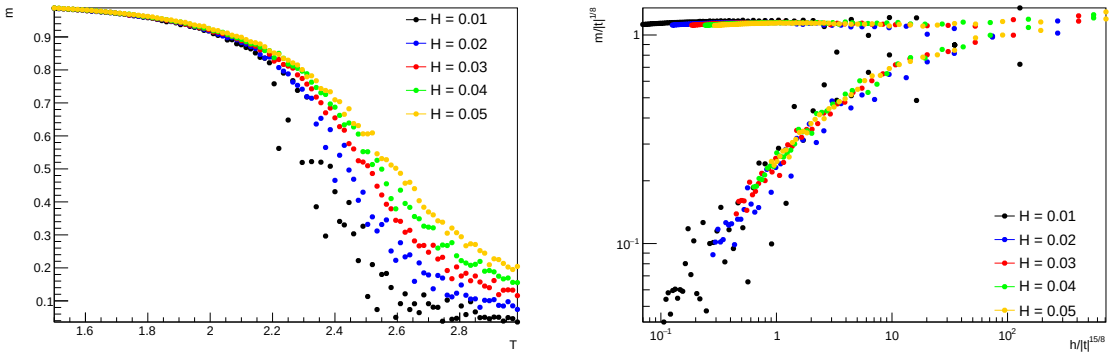

Figure 4: The left pad shows the magnetization per spin m depending on temperature T with various input H. The right pad shows the scaled depending on scaled $h/|t|^{15/8}$ with various input H.

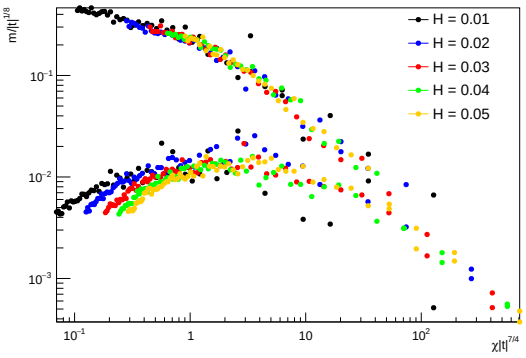

Figure 5: The scaled $m/|t|^{1/8}$ depends on scaled $\chi|t|^{7/4}$ with various input H.

Relevant code:

For the simulation code, it has already shown in problem (1).

For the calculation of χ

```
//-------------------------------------------------------------------------//

void ising_model_2d_scaling() {

        const int n = 100;
        const double Tmax = 3;
        const double Tmin = 1.5;
        const double dT = (Tmax - Tmin)/n;
        const int nH = 5;
        const double dH = 0.01;
        double H[nH];
        Int_t color[nH] = {kBlack, kBlue, kRed, kGreen, kOrange};

        double T[nH][n];
        double Te[nH][n] = {0};
        double M[nH][n];
        double Me[nH][n];
        double Ms[nH][n];
```

```
        double E[nH][n];
        double Ee[nH][n];
        double Es[nH][n];
        double C[nH][n];
        double Chi[nH][n];
        int    iTc[nH] = {0};
        double Tc[nH];

        int t_max = 3000;
        int t_stb = 1000;
        int N = 20;

        for(int iH=0; iH<nH; iH++) {
                H[iH] = 0.01*(iH+1);
                for(int i=0; i<n; i++) {
                        double Ttmp = Tmin + dT*i;
                        T[iH][i] = Ttmp;
                        vector<double> v_tmp
                        = ising_model_2d_plot(N, Ttmp, H[iH], t_max, t_stb, false);
                        M[iH][i]  = v_tmp[0];
                        Me[iH][i] = v_tmp[1];
                        Ms[iH][i] = v_tmp[1]*sqrt(t_max-t_stb);
                        E[iH][i]  = v_tmp[2];
                        Ee[iH][i] = v_tmp[3];
                        Es[iH][i] = v_tmp[3]*sqrt(t_max-t_stb);
                        C[iH][i]  = N*N*Es[iH][i]*Es[iH][i]/Ttmp/Ttmp;
                        Chi[iH][i] = Ms[iH][i]*Ms[iH][i]*N*N/Ttmp;
                }
                int imax = 0;
                double Cmax = 0;
                for(int i=1; i<n; i++) {
                        if(Cmax >= C[iH][i]) continue;
                        Cmax = C[iH][i];
                        imax = i;
                }
                iTc[iH] = imax;
                Tc[iH] = T[iH][imax];
        }

        vector<double> T_scaled[nH];
        vector<double> M_scaled[nH];
        vector<double> H_scaled[nH];
        vector<double> Chi_scaled[nH];
        for(int iH=0; iH<nH; iH++) {
                for(int i=0; i<n; i++) {
                        if(T[iH][i] == Tc[iH]) continue;
                        if(M[iH][i]  <0) continue;
                        if(Chi[iH][i]<0) continue;
                        double Ttmp = fabs(T[iH][i] - Tc[iH])/Tc[iH];
                        T_scaled[iH].push_back(Ttmp);
                        M_scaled[iH].push_back(M[iH][i]*pow(Ttmp,-0.125));
                        H_scaled[iH].push_back(H[iH]*pow(Ttmp, -1.875));
                        Chi_scaled[iH].push_back(Chi[iH][i]*pow(Ttmp, 1.75));
                }
        }

        // ... plot code ...
}
```

(4) *[Problem 9.1 (p.283)]*
Calculate the speed distributions for a dilute gas as Figure 9.4 and compare the results quantitatively with the Maxwell distribution. (For example, perform the χ^2 analysis described in Appendix G.) This analysis also yields the temperature; compare the value you find with the result calculated directly from the equipartition theorem.

Physics explanation:

As used in the textbook Figure 9.3 and 9.4, we follow the simulation setting: 20 particles in a 10×10 2D box. To make sure it would converge, we use small time step $\Delta t = 0.005$ and maximum initial velocity 0.05.

Figure 6 shows the velocity distribution (left pad v_x, middle pad v_y, right pad v) in various time range (red $0 < t < 20$, green $20 < t < 40$, blue $40 < t < 60$, and black $0 < t < 60$). To compare the v results with Maxwell distribution quantitatively, we perform the χ^2 analysis and fit the v distribution. The 2D Maxwell distribution (PDF, probability density distribution) is

$$P\mathrm{d}v = \frac{mv}{k_B T} e^{-mv^2/(2k_B T)} \mathrm{d}v. \tag{5}$$

In our numerical unit system, we can equivalently regard $m = 1$ and $k_B = 1$. For the convenience of fitting, we the function with one fitting parameter p_0

$$p_0 v e^{-0.5 p_0 v^2}, \tag{6}$$

where $p_0 = 1/T$. We fit the whole time range ($0 < t < 60$) in the right pad of Fig. 6, and get

$$\chi^2/\mathrm{ndf} = 440.4/27, \quad p_0 = 0.5028 \pm 0.0031, \tag{7}$$

where ndf means number of degree of freedom. The fitting quality is not ideal but acceptable. We can then get the temperature from this χ^2 analysis

$$T = 1/p_0 = 1.989 \pm 0.012. \tag{8}$$

We can also calculate the temperature directly in the simulation from the equipartition theorem

$$\left\langle \frac{1}{2} m (v_x^2 + v_y^2) \right\rangle = k_B T. \tag{9}$$

Figure 7 shows this temperature in the right pad. We can get the mean value roughly $T \approx 2$, which is consistent with temperature from fitting.

Plots:

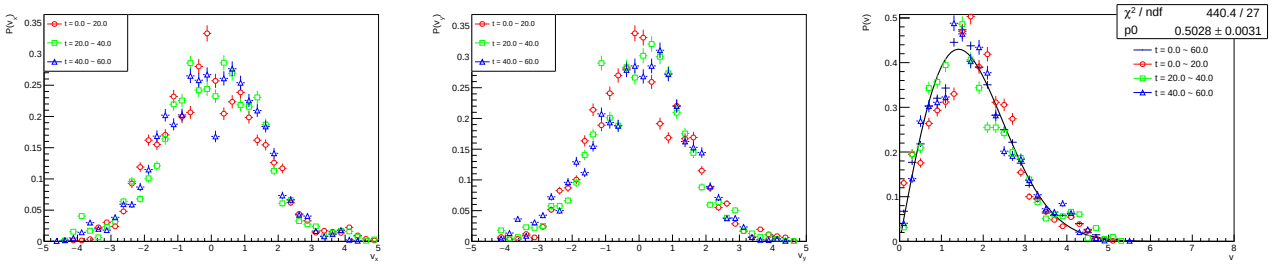

Figure 6: The plots above show the velocity distribution (left pad v_x, middle pad v_y, right pad v) in various time range (red $0 < t < 20$, green $20 < t < 40$, blue $40 < t < 60$, and black $0 < t < 60$). In the right pad, the whole time range ($0 < t < 60$) is fitted by the 2D Maxwell distribution.

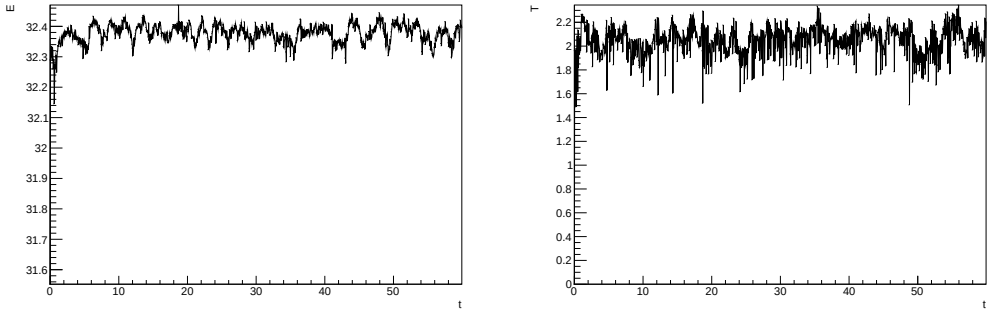

Figure 7: The plots above show the total energy and temperature depending on time.

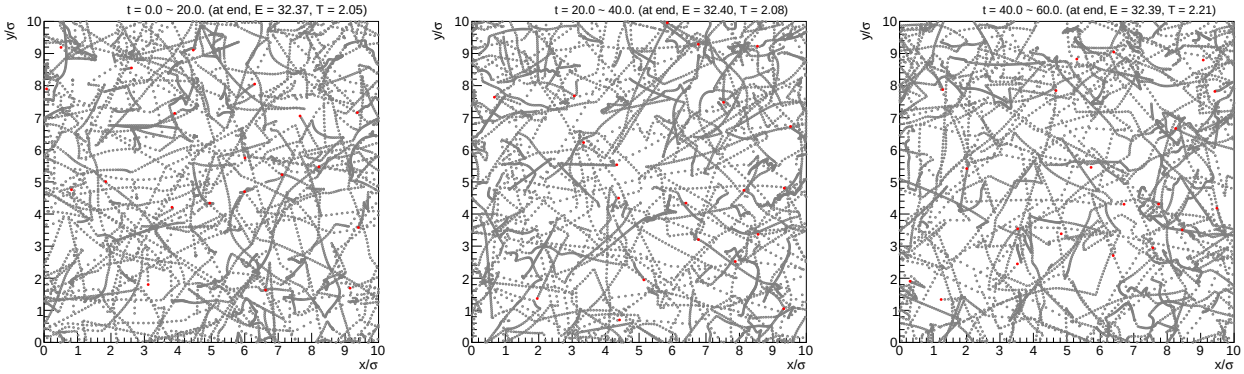

Figure 8: The plots above show the position of moleculars in different time periods: left pad $0 < t < 20$; middle pad $20 < t < 40$; right pad $40 < t < 60$. The red dots are positions at the end point of each time period.

Relevant code:

For the initialization of the simulation

```
//------------------------------------------------------------------------//

MolecularDynamics2D::MolecularDynamics2D(int N, double L, double dt, double dmax, double vmax) {

        _N = N;
        _rec = 10;
        _L = L;
        _dt = dt;
        _dmax = dmax;
        _vmax = vmax;
        _t_now = 0;

        _tag_displace_now = 0;
        _tag_distance_now = 0;

        _x.clear();
        _y.clear();
        _vx.clear();
        _vy.clear();

        for(int i=0; i<_N; i++) {
                _x_old.push_back(0);
                _x_now.push_back(0);
                _x_new.push_back(0);
                _y_old.push_back(0);
                _y_now.push_back(0);
                _y_new.push_back(0);

                _vx_now.push_back(0);
                _vy_now.push_back(0);
        }

        double sqN = sqrt(_N);
        double grid = sqN==floor(sqN)?_L/sqN:_L/(sqN+1);

        int n = 0;
        double i = 0;
        while(i < _L) {
                double j = 0;
                while(j < _L) {
                        if(n >= _N) break;
                        _x_now[n] = i + 0.5*grid + _dmax*(1.0*rand()/RAND_MAX-0.5)*grid*sqrt(2);
                        _y_now[n] = j + 0.5*grid + _dmax*(1.0*rand()/RAND_MAX-0.5)*grid*sqrt(2);
                        _vx_now[n] = _vmax*(1.0*rand()/RAND_MAX-0.5)*sqrt(2);
                        _vy_now[n] = _vmax*(1.0*rand()/RAND_MAX-0.5)*sqrt(2);
                        _x_old[n] = _x_now[n] - _vx_now[n]*_dt;
                        _y_old[n] = _y_now[n] - _vy_now[n]*_dt;

                        n ++;
                        j += grid;
                }
                i += grid;
        }
        _x_start = _x_old;
        _y_start = _y_old;

        cout << "initialization completed" << endl;
```

```
}

//------------------------------------------------------------------------//
```

For the simulation process

```
//------------------------------------------------------------------------//

void MolecularDynamics2D::cal_once() {

        for(int i=0; i<_N; i++) {
                double fx = 0;
                double fy = 0;

                for(int j=0; j<_N; j++) {
                        if(j == i) continue;

                        double dx = _x_now[i] - _x_now[j];
                        double dy = _y_now[i] - _y_now[j];

                        if(fabs(dx) > 0.5*_L) dx -= dx>0?_L:-_L;
                        if(fabs(dy) > 0.5*_L) dy -= dy>0?_L:-_L;

                        double r = sqrt(dx*dx + dy*dy);
                        if(r < 3) {
                                double fij = 24.0*(2.0/pow(r,13) - 1.0/pow(r,7));
                                fx += fij*dx/r;
                                fy += fij*dy/r;
                        }
                }

                _x_new[i] = 2*_x_now[i] - _x_old[i] + fx*_dt*_dt;
                _y_new[i] = 2*_y_now[i] - _y_old[i] + fy*_dt*_dt;
                _vx_now[i] = (_x_new[i] - _x_old[i])/(2.0*_dt);
                _vy_now[i] = (_y_new[i] - _y_old[i])/(2.0*_dt);

                if(i == 0) {
                        double dx = _x_new[i] - _x_start[i];
                        double dy = _y_new[i] - _y_start[i];
                        if(fabs(dx) > 0.5*_L) dx -= dx>0?_L:-_L;
                        if(fabs(dy) > 0.5*_L) dy -= dy>0?_L:-_L;
                        _tag_displace_now = sqrt(dx*dx + dy*dy);
                } else if(i == 1) {
                        double dx = _x_new[i] - _x_new[0];
                        double dy = _y_new[i] - _y_new[0];
                        if(fabs(dx) > 0.5*_L) dx -= dx>0?_L:-_L;
                        if(fabs(dy) > 0.5*_L) dy -= dy>0?_L:-_L;
                        _tag_distance_now = sqrt(dx*dx + dy*dy);
                }

                if(_x_new[i]<0) {
                        _x_new[i] += _L;
                        _x_now[i] += _L;
                } else if(_x_new[i]>_L) {
                        _x_new[i] -= _L;
                        _x_now[i] -= _L;
                }
                if(_y_new[i]<0) {
                        _y_new[i] += _L;
                        _y_now[i] += _L;
                } else if(_y_new[i]>_L) {
```

```
                        _y_new[i] -= _L;
                        _y_now[i] -= _L;
                }
        }

        _x_old = _x_now;
        _x_now = _x_new;
        _y_old = _y_now;
        _y_now = _y_new;
}

//------------------------------------------------------------------------//

void MolecularDynamics2D::cal_ET() {

        double Ekin = 0;
        double Epot = 0;

        for(int i=0; i<_N; i++) {
                Ekin += 0.5*(_vx_now[i]*_vx_now[i] + _vy_now[i]*_vy_now[i]);

                for(int j=i+1; j<_N; j++) {

                        double dx = _x_old[j] - _x_old[i];
                        double dy = _y_old[j] - _y_old[i];

                        if(fabs(dx) > 0.5*_L) dx -= dx>0?_L:-_L;
                        if(fabs(dy) > 0.5*_L) dy -= dy>0?_L:-_L;

                        double r = sqrt(dx*dx + dy*dy);
                        if(r < 3) {
                                double invr6 = 1.0/pow(r,6);
                                double invr12 = invr6*invr6;
                                Epot += 4*(invr12 - invr6);
                        }
                }
        }

        _E_now = Ekin + Epot;
        _T_now = Ekin/_N;
}

//------------------------------------------------------------------------//

void MolecularDynamics2D::cal_until(double t_end) {

        if(!check()) {
                cout << "ERROR: check() not pass!" << endl;
                return;
        }
        cout << "check passed" << endl;

        _t.clear();
        _x.clear();
        _y.clear();
        _vx.clear();
        _vy.clear();
        _tag_displace.clear();
        _tag_distance.clear();

```

```
116         _E.clear();
117         _T.clear();
118 
119         int n = 0;
120 
121         while(_t_now < t_end) {
122                 if(n%_rec == 0) {
123                         _t.push_back(_t_now);
124                         _x.push_back(_x_now);
125                         _y.push_back(_y_now);
126                         _vx.push_back(_vx_now);
127                         _vy.push_back(_vy_now);
128                         _tag_displace.push_back(_tag_displace_now);
129                         _tag_distance.push_back(_tag_distance_now);
130 
131                         cal_ET();
132                         _E.push_back(_E_now);
133                         _T.push_back(_T_now);
134                 }
135 
136                 cal_once();
137 
138                 n ++;
139                 _t_now += _dt;
140         }
141 }
142 
143 //--------------------------------------------------------------------------//
```