PHYS580 Homework 4

Yicheng Feng
PUID: 0030193826

October 27, 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:

- `random__walk__2d.h` and `random__walk__2d.cxx` for the class `RandomWalk2D` to simulate the 2D random walking with different modes including SAW.
- `random__walk__3d.h` and `random__walk__3d.cxx` for the class `RandomWalk3D` to simulate the 3D random walking with different modes including SAW.
- `diffusion__2d.h` and `diffusion__2d.cxx` for the class `Diffusion2D` to simulate the 2D diffusion with close/leaking container.
- `percolation__2d.h` and `percolation__2d.cxx` for the class `Percolation2D` to simulate the 2D percolaiton.
- `hw4.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/hw4`.

(1) *[Problem 7.6 (p.194)] in the Giordano-Nakanishi textbook. Also check whether you can reproduce the analytic result $\nu = 3/4$ for SAW on a 2D grid*
Simulate SAWs in three dimensions. Determine the variation of $\langle r^2 \rangle$ with step number and find the value of ν, where this parameter is defined through the relation (7.9). Compare your results with those in Figure. 7.6. You should find that ν decreases for successively higher dimensions. (It is 1 in one dimension and 3/4 in two dimensions.) Can you explain this trend qualitatively?

Physics explanation:

Figure 1 shows the 3D $\langle r^2 \rangle$ depends on the number of steps (time t). The right plot is log-log plot with good linear fitting. The slope is 1.2231, so the simulaiton result of ν_3 in this 3D case is

$$2\nu_3 = 1.2231, \quad \Rightarrow \quad \nu_3 = 0.6116. \tag{1}$$

Figure 2 shows the 2D $\langle r^2 \rangle$ depends on the number of steps (time t). The right plot is log-log plot with good linear fitting. The slope is 1.4503, so the simulaiton result of ν_3 in this 3D case is

$$2\nu_2 = 1.4503, \quad \Rightarrow \quad \nu_2 = 0.7252, \tag{2}$$

which is close to the theoretical result $\nu = 0.75$ with error 3.3%.

The trend of ν is that it decrease along the dimension. For d dimension space, the room increase along r is $r^{d-1}\mathrm{d}r$, so for higher dimension, the room increase faster along r, which means there is more room for the SAW. SAW trend to stand there longer without go further in r, so for the given time, the $\langle r^2 \rangle$ tends to be smaller, which explains why $\nu_1 > \nu_2 > \nu_3$.

Plots:

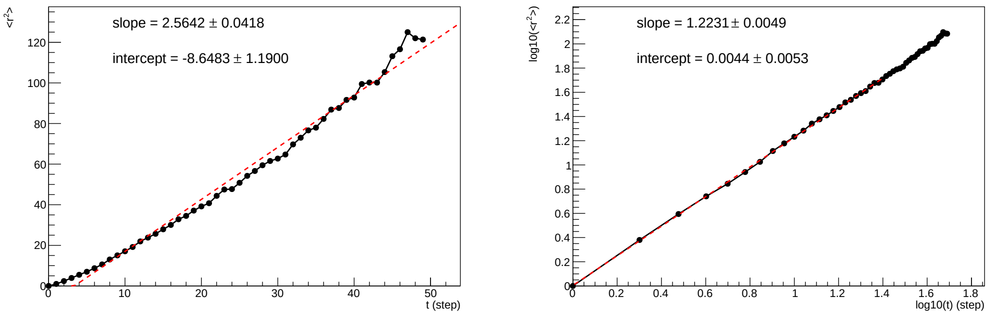

Figure 1: In the 3D case, the $\langle r^2 \rangle$ depends on the number of steps (time t). The right pad is the log-log plot. Ignore the fitting on the left pad.

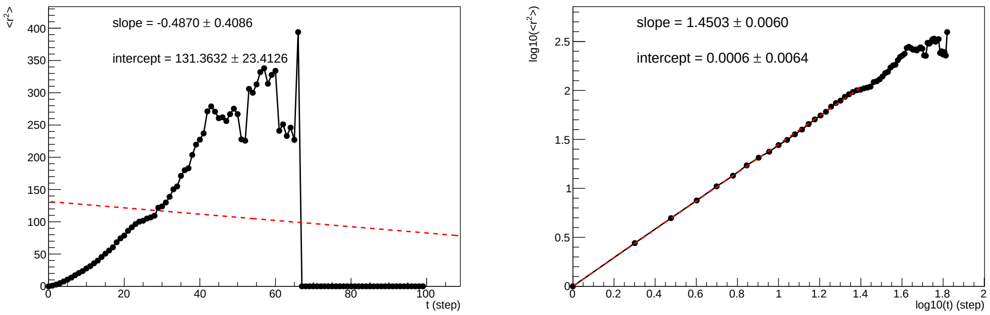

Figure 2: In the 2D case, the $\langle r^2 \rangle$ depends on the number of steps (time t). The right pad is the log-log plot. Ignore the fitting on the left pad.

Relevant code:

For the 3D SAW

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

bool RandomWalk3D::cal_saw() {

        vector< vector< vector<bool> > > occupied;
        for(int i=0; i<_n_step*2+1; i++) {
                vector< vector<bool> > tmp2;
                for(int j=0; j<_n_step*2+1; j++) {
                        vector<bool> tmp1;
                        for(int k=0; k<_n_step*2+1; k++) {
                                tmp1.push_back(false);
                        }
                        tmp2.push_back(tmp1);
                }
                occupied.push_back(tmp2);
        }

        _x.clear();
        _y.clear();
        _z.clear();

        _x.push_back(0);
        _y.push_back(0);
        _z.push_back(0);
        occupied[_n_step][_n_step][_n_step] = true;

        _x.push_back(1);
        _y.push_back(0);
        _z.push_back(0);
        occupied[_n_step+1][_n_step][_n_step] = true;

        int dir = 2;
        int nodir = 1;
        double delta[6][3] = {{1,0,0}, {-1,0,0}, {0,1,0}, {0,-1,0}, {0,0,1}, {0,0,-1}};
        int nx;
        int ny;
        int nz;
        int idx;

        for(int i=2; i<_n_step; i++) {

                idx = int(5.0*rand()/RAND_MAX);
                if(idx>=nodir) idx++;

                nx = int(_x[i-1]+delta[idx][0]);
                ny = int(_y[i-1]+delta[idx][1]);
                nz = int(_z[i-1]+delta[idx][2]);

                dir = idx;
                if(idx==0) nodir = 1;
                if(idx==1) nodir = 0;
                if(idx==2) nodir = 3;
                if(idx==3) nodir = 2;
                if(idx==4) nodir = 5;
                if(idx==5) nodir = 4;

                if(occupied[_n_step+nx][_n_step+ny][_n_step+nz]) {
                        break;
                } else {
                        occupied[_n_step+nx][_n_step+ny][_n_step+nz] = true;
                        _x.push_back(1.0*nx);
                        _y.push_back(1.0*ny);
                        _z.push_back(1.0*nz);
                }
        }

        if(_x.size()<_min_step*_n_step) return false;

        return true;
}

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

For the 2D SAW

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

bool RandomWalk2D::cal_saw() {

        vector< vector<bool> > occupied;
        for(int i=0; i<_n_step*2+1; i++) {
                vector<bool> tmp;
                for(int j=0; j<_n_step*2+1; j++) {
                        tmp.push_back(false);
                }
                occupied.push_back(tmp);
        }

        _x.clear();
        _y.clear();

        _x.push_back(0);
        _y.push_back(0);
        occupied[_n_step][_n_step] = true;

        _x.push_back(1);
        _y.push_back(0);
        occupied[_n_step+1][_n_step] = true;

        int dir = 2;
        int nodir = 1;
        double delta[4][2] = {{1,0}, {-1,0}, {0,1}, {0,-1}};
        int nx;
        int ny;
        int idx;

        for(int i=2; i<_n_step; i++) {

                idx = int(3.0*rand()/RAND_MAX);
                if(idx>=nodir) idx++;

                nx = int(_x[i-1]+delta[idx][0]);
                ny = int(_y[i-1]+delta[idx][1]);

                dir = idx;
                if(idx==0) nodir = 1;
                if(idx==1) nodir = 0;
                if(idx==2) nodir = 3;
                if(idx==3) nodir = 2;

                if(occupied[_n_step+nx][_n_step+ny]) {
                        break;
                } else {
                        occupied[_n_step+nx][_n_step+ny] = true;
                        _x.push_back(1.0*nx);
                        _y.push_back(1.0*ny);
                }
        }

        if(_x.size()<_min_step*_n_step) return false;

        return true;
}

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

(2) *[Problem 7.12 (p.205)] If you have trouble with running time, then try first on courser grids (e.g.,* 50×50 *for the random walk,* 4×4 *for the entropy, and only 100 particles)*
Calculate the entropy for the cream-in-your-coffee problem, and reproduce the results in Figure 7.16.

Physics explanation:

In this problem, we set the 2D "cup" to be a 200×200 square. At the beginning, 300 cream particle is dropped at the origin $(0, 0)$. In each time step, we randomly pick one of the particles and let it walk one step randomly. Figure 3 shows the diffusion status of various time from $t = 10^0$ to $t \approx 10^7$.

For the entropy, we divide the 200×200 random walking lattice into 5×5 grids (each one has 40×40 of the original lattice). Figure 4 shows the entropy dependence on time t. We can see that it increases with time, and becomes stable after sufficiently long time $t = 10^6$.

Plots:

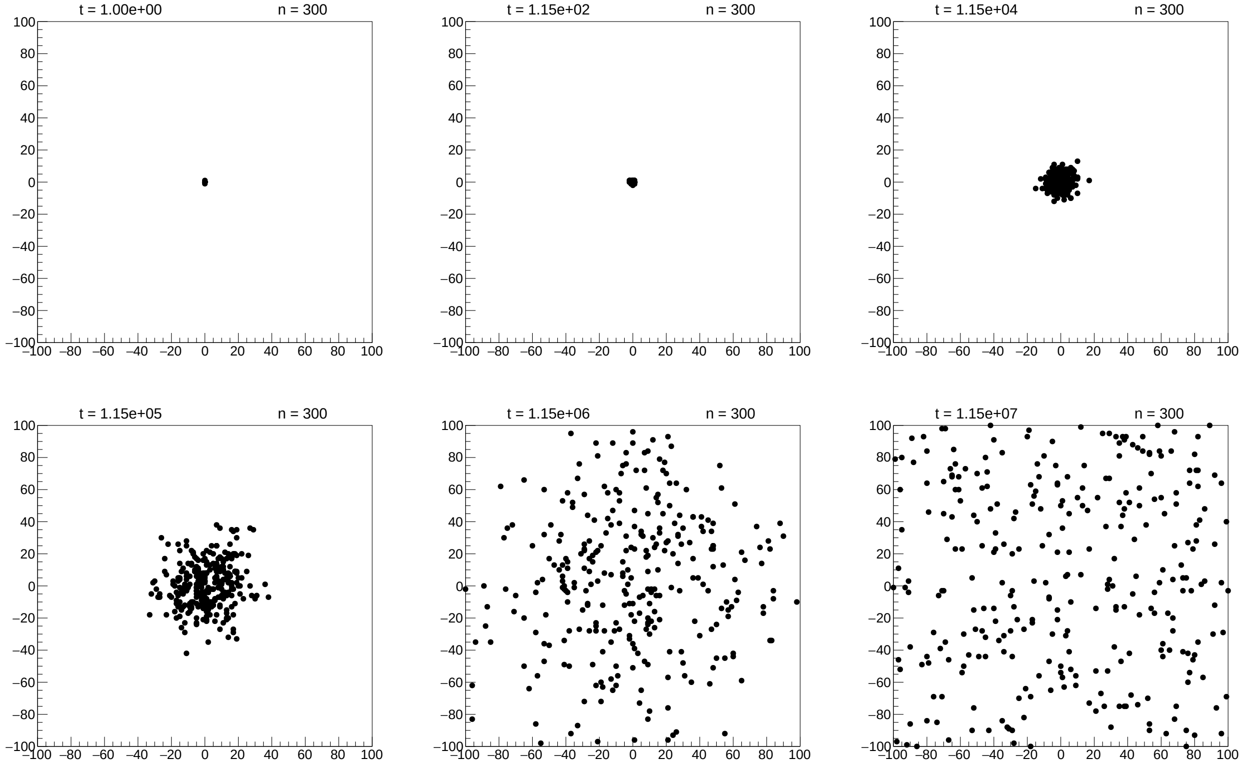

Figure 3: The diffusion changes with time.

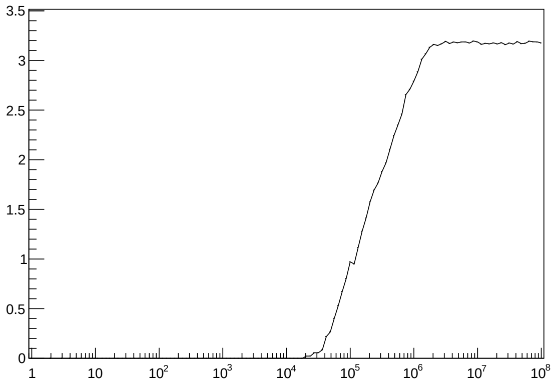

Figure 4: The entropy changes with time.

Relevant code:

For the simulation of diffusion

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

void Diffusion2D::cal_until(double t_stop) {

        double delta[4][2] = {{1,0}, {-1,0}, {0,1}, {0,-1}};

        for(int i=int(_t); i<t_stop; i++) {
                int pid = int(1.0*_P*rand()/RAND_MAX);
                int idx = int(4.0*rand()/RAND_MAX);
                _x[pid] += delta[idx][0];
                _y[pid] += delta[idx][1];
                // periodic condition
                if(_x[pid]<-_M) _x[pid] += _N;
                if(_x[pid]> _M) _x[pid] -= _N;
                if(_y[pid]<-_M) _y[pid] += _N;
                if(_y[pid]> _M) _y[pid] -= _N;
        }

        _t = t_stop;

        cal_entropy();
}

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

For the calculation of entropy

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

double Diffusion2D::cal_entropy() {

        int dx = _N/_D;
        int dy = _N/_D;

        vector< vector<double> > ndist;
        for(int i=0; i<_D; i++) {
                vector<double> tmp;
                for(int j=0; j<_D; j++) {
                        tmp.push_back(0);
                }
                ndist.push_back(tmp);
        }

        for(int i=0; i<_P; i++) {
                int ix = int((_x[i]+_M)/dx);
                if(ix<0) ix = 0;
                if(ix>=_D) ix = _D-1;
                int iy = int((_y[i]+_M)/dy);
                if(iy<0) iy = 0;
                if(iy>=_D) iy = _D-1;

                ndist[ix][iy] += 1.0;
        }

        double tmpS = 0;
        for(int ix=0; ix<_D; ix++) {
                for(int iy=0; iy<_D; iy++) {
                        double ntmp = ndist[ix][iy];
                        if(ntmp == 0) {
                                tmpS -= 0;
                        } else {
                                tmpS -= 1.0*ntmp/_P*log(1.0*ntmp/_P);
                        }
                }
        }

        _S = tmpS;
        return _S;
}

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

(3) *[Problem 7.15 (p.205)]*

Perform the random-walk simulation of spreading cream (Figure 7.13 and 7.14), and let one of the walls of the container possess a small hole so that if a cream particle enters the hole, it leaves the container. Calulate the number of particles in the container as a function of time. Show that this number, which is proportional to the partial pressure of the cream particles varies as $e^{-t/\tau}$, where τ is the effective time constant for the escape. *Hint:* Reasonable parameter choices are a 50×50 container lattice and a hole 10 units in the length along one of the edges.

Physics explanation:

As suggested, we set the 2D container to be 50×50 and the position of the hole is $-5 \leq x \leq 5$, $y = 25$, on the upper edge. At the beginning, 400 cream particle is dropped at the origin $(0, 0)$. In each time step, we randomly pick one of the particles and let it walk one step randomly.

Figure 5 shows the diffusion and leaking evolute along time from $t \approx 8 \times 10^4$ to $t \approx 9 \times 10^6$. Figure 6 shows the number of cream particles in the container depends on time. The particle number is of log-scale. We fit the curve with a straight line. From the slope k of the linear fitting, we can calculate $\tau = -1/k = 2.90 \times 10^6$. This fitting is quite good, so it proves that the residual particle number is proportional to the partial pressure of the cream particles varies as $e^{-t/\tau}$.

Plots:

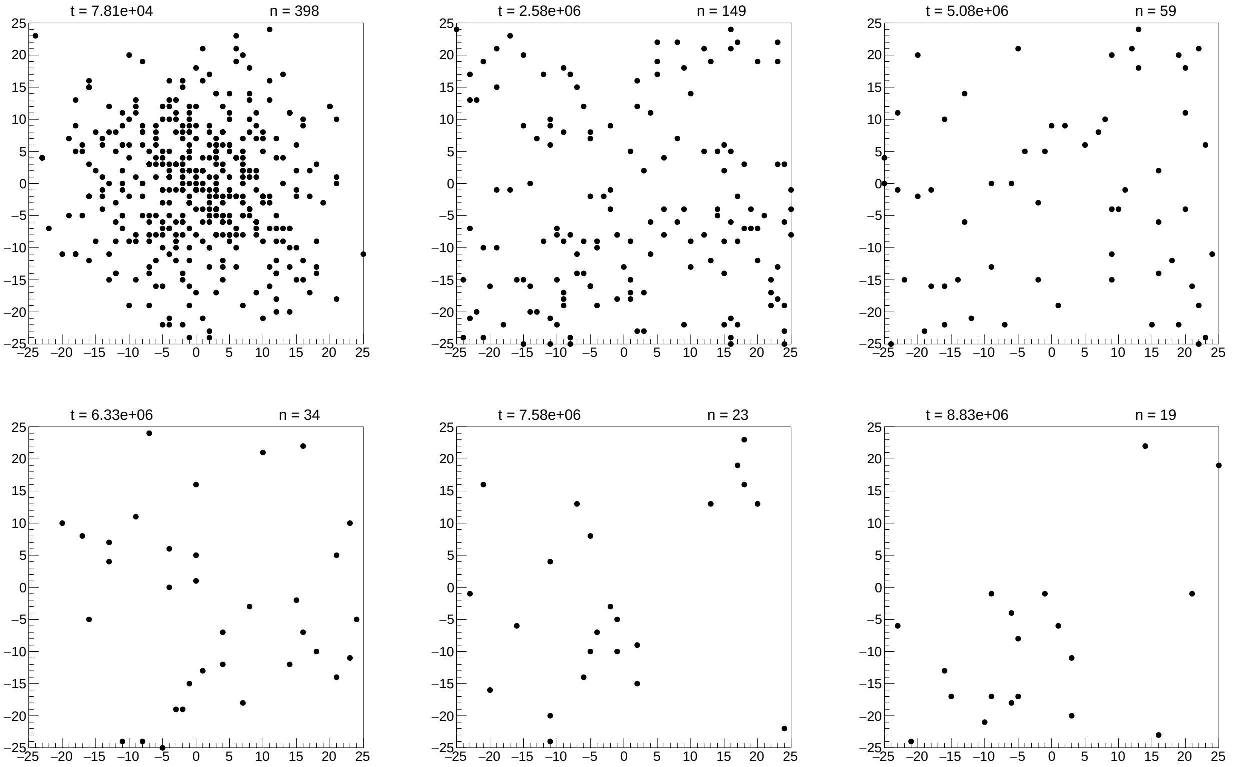

Figure 5: The diffusion and leaking of cream particles evolutes with time.

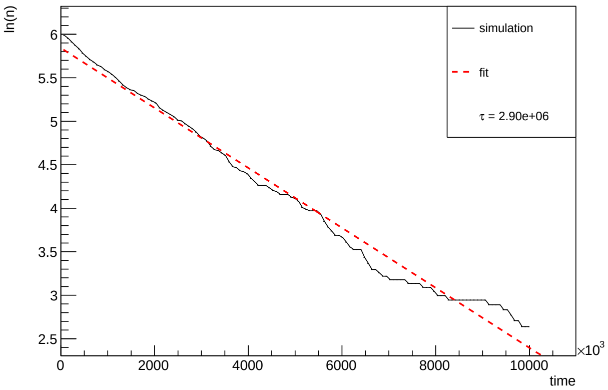

Figure 6: The number of cream particles in the container depends on time. The particle number is of log-scale.

Relevant code:

For the simulation of diffusion and leaking

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

void Diffusion2D::cal_leak_until(double t_stop) {

        double delta[4][2] = {{1,0}, {-1,0}, {0,1}, {0,-1}};

        for(int i=int(_t); i<t_stop; i++) {
                if(_P == 0) break;
                int pid = int(1.0*_P0*rand()/RAND_MAX);
                if(pid>=_P) continue;
                int idx = int(4.0*rand()/RAND_MAX);
                _x[pid] += delta[idx][0];
                _y[pid] += delta[idx][1];
                if(_x[pid]>=-_L && _x[pid]<=_L && _y[pid]>=_M) {
                        _x.erase(_x.begin()+pid);
                        _y.erase(_y.begin()+pid);
                        _P -= 1;
                        continue;
                }
                // reflection condition
                if(_x[pid]<-_M) _x[pid] = -2*_M - _x[pid];
                if(_x[pid]> _M) _x[pid] =  2*_M - _x[pid];
                if(_y[pid]<-_M) _y[pid] = -2*_M - _y[pid];
                if(_y[pid]> _M) _y[pid] =  2*_M - _y[pid];
        }

        _t = t_stop;

        cal_entropy();
}

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

(4) *[Problem 7.30 (p.228)]*
Generate a spanning cluster for a two dimensional square lattice at $p = p_c$ using any of the search methods discussed in connection with Figure 7.29. Estimate the fractial dimensionality of the cluster. You should find a value of d_f, which is slightly smaller than 2 (the expected value is $91/48 \approx 1.90$).

Physics explanation:

I wrote the program with C++ from scratch with the following steps

- Generate all occupied sites with p. (time complexity $\mathcal{O}(N)$, $N = L^2$)
- Use depth first search (DFS) to find all clusters. ($\mathcal{O}(N)$)
- Find the largest cluster, and calculate the P and S.

The scan of L is did from $L = 50$ to $L = 1000$. In each case, the simulation is repeated 50 times, and the percolation probability is averaged over all those trials.

In the textbook, the relationship between the mass m (the site number of the spanning cluster) and the edge length L is studied to get the fractal dimension d_f.

$$m = \text{site number of the spanning cluster} = P(p) \times \text{number of all occupied sites} \approx P(p) \times L^2 p. \quad (3)$$

For convenience, we can just study the relationship between $L^2 P(p_c)$ and L to get d_f, which is effectively equivalent to the m–L.

Figure 7 shows $L^2 P(p_c)$ depending on L. The right pad is the log-log plot with very good linear fitting. The slope is $d_f = 1.89929$ which is very close to the expected value $91/48 \approx 1.90$.

Plots:

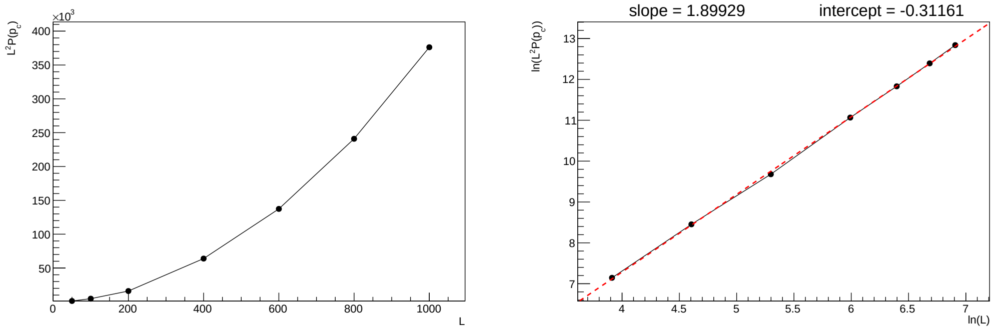

Figure 7: The $L^2 P(p_c)$ depends on the L. The right pad is the log-log plot. This variable is directly related to the mass m: $L^2 P(p_c) = m/p_c$.

Relevant code:

For the generation of the occupied sites

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

void Percolation2D::cal_perculation() {

        for(int i=0; i<_L; i++) {
                for(int j=0; j<_L; j++) {
                        if(1.0*rand()/RAND_MAX<_p) {
                                _occupied[i][j] = true;
                        } else {
                                _occupied[i][j] = false;
                        }
                }
        }
}

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

For the DFS to find all clusters

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

void Percolation2D::cal_DFS(int ix, int iy, int cid) {

        if(ix<0 || ix>=_L) return;
        if(iy<0 || iy>=_L) return;

        if(_cluster_id[iy][ix] == 0 && _occupied[iy][ix]) {
                _cluster_id[iy][ix] = cid;
        } else {
                return;
        }

        cal_DFS(ix+1, iy, cid);
        cal_DFS(ix-1, iy, cid);
        cal_DFS(ix, iy+1, cid);
        cal_DFS(ix, iy-1, cid);
}

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

void Percolation2D::cal_cluster() {

        for(int i=0; i<_L; i++) {
                for(int j=0; j<_L; j++) {
                        _cluster_id[i][j] = 0;
                }
        }

        int cid = 0;
        for(int iy=0; iy<_L; iy++) {
                for(int ix=0; ix<_L; ix++) {
                        if(_cluster_id[iy][ix]==0 && _occupied[iy][ix]) {
                                cid ++;
                                cal_DFS(ix, iy, cid);
                        }
                }
        }

        _cluster_size.clear();
        _cluster_x.clear();
        _cluster_y.clear();
        _spanning_cluster_id.clear();
        for(int i=0; i<=cid; i++) {
                _cluster_size.push_back(0);
                _cluster_x.push_back(vector<int>());
                _cluster_y.push_back(vector<int>());
        }
        for(int iy=0; iy<_L; iy++) {
                for(int ix=0; ix<_L; ix++) {
                        int idx = _cluster_id[iy][ix];
                        _cluster_size[idx] ++;
                        _cluster_x[idx].push_back(ix);
                        _cluster_y[idx].push_back(iy);
                }
        }

        int max = 0;
        int max_id = 1;
        for(int i=1; i<=cid; i++) {
                if(_cluster_size[i]>max) {
                        max = _cluster_size[i];
                        max_id = i;
                }
        }
        for(int i=1; i<=cid; i++) {
                if(_cluster_size[i]==max) {
                        _spanning_cluster_id.push_back(i);
                }
        }
        _n_cluster = cid;
}

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

For calculation of the P and S

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

void Percolation2D::cal_PS() {

        if(!check()) return;

        _P = 1.0*_cluster_size[_spanning_cluster_id[0]]/_n_occupied;

        double sum2 = 0;
        double tmpi = 0;
        for(int i=1; i<_cluster_size.size(); i++) {
                if(tmpi < _spanning_cluster_id.size() && i == _spanning_cluster_id[tmpi]){
                        tmpi ++;
                        continue;
                }
                sum2 += 1.0*_cluster_size[i]*_cluster_size[i];
        }
        _S = sum2/_L/_L;
}

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