PHYS580 Lab09 Report

Yicheng Feng
PUID: 0030193826

October 20, 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:

- `monte_carlo_integrate.h` and `monte_carlo_integrate.cxx` for the class `MonteCarloIntegrate` to do the Monte Carlo integration.
- `random_walk_2d.h` and `random_walk_2d.cxx` for the class `RandomWalk2D` to calculate the random walks of different mods, including unit step length case, continuous step length and self-avoiding walk.
- `lab9.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/lab9`.

(1) The first starter program provided for this lab, montecarlo.m , performs Monte Carlo integration of a simple function, $f(x) = \sqrt{4 - x^2}$ (by default). Use the code (or your own equivalent one) to integrate this $f(x)$ from $x = 0$ to 2, and thus compute numerically, to 3, 4, and 5 significant digits. Observe how the average error (standard error of the mean) decreases as a function of the total number N of random numbers used (N = numbers generated per trial $\times$ the number of trials). Then, try to compute via Monte Carlo the following two integrals: i) $\int_{-2}^{2} \frac{\mathrm{d}x}{\sqrt{4-x^2}}$ and ii) $\int_0^{-\infty} e^{-x} \ln x \mathrm{d}x$. Take care to handle any divergences in the integrand as well as the range of integration extending to infinity.

Physics explanation:

It is easy to know the integral is $\int_0^2 \sqrt{4 - x^2}\mathrm{d}x = \pi$, because it is one quarter of the circule area with radius 2 ($2^2\pi/4 = \pi$). We can take this integration as an example to talk about the process of Monte Carlo integration.

Firstly, we use uniform distribution to randomly sample x from the domain and get x_i as well as $f(x_i)$. Secodly, we repeat the first step n times and get the average value $\langle f(x)\rangle = \sum_{i=1}^{n} f(x_i)$. Then, the integration could be estimated by the volume of the domain times the average.

$$x_i \sim U(a, b), \qquad I = \int_a^b f(x)\mathrm{d}x \approx \frac{b-a}{n} \sum_{i=1}^{n} f(x_i) \tag{1}$$

The Monte Carlo integration can be easily extended to higher-dimension integrations with integration domain V.

$$\vec{x}_i \sim U(V), \qquad I = \int_V f(\vec{x})\mathrm{d}\vec{x} \approx \frac{V}{n} \sum_{i=1}^{n} f(\vec{x}_i) \tag{2}$$

All the steps above is called a trial. The last step is to do many trials m and get the average of the them as a more presice estimation. Moreover, the error of the estimation could be calculated from the difference of those trials.

Now, we estimate π from the integration $\int_0^2 \sqrt{4 - x^2}\mathrm{d}x = \pi$. We fix the number of sample the same in each trial ($n = 1000$), and change the number of trials m to change the $N = mn$.

n	1000	1000	1000	1000	1000	1000
m	5	10	100	500	1000	20000
N	5000	10000	100000	500000	1000000	20000000
mean	3.13370	3.14399	3.13912	3.14051	3.14053	3.14198
error	0.01375	0.00876	0.00285	0.00129	0.00092	0.00020

In Fig. 1, we plot the graph $\log_{10}$(error) vs. $\log_{10}(N)$ and fit it by a straight line to find the relationship between the error and the N. The slope is about -0.5, so the relationship between the error and the N should be

$$\text{error} \propto \frac{1}{\sqrt{N}}. \tag{3}$$

For the other integrations:

i) To avoid the divergences at $x = \pm 2$, we set cutoffs for the domain by 10^{-8}, and the domain becomes $-2+10^{-8} \leq x < 2-10^{-8}$. We set the number of samples per trial $n = 1000$, and the number of trials $m = 10000$, and get the monte carlo integration

$$\int_{-2}^{2} \frac{\mathrm{d}x}{\sqrt{4-x^2}} \approx \int_{-2+10^{-8}}^{2-10^{-8}} \frac{\mathrm{d}x}{\sqrt{4-x^2}} \approx 3.14042 \pm 0.00163323. \tag{4}$$

which is close to the exact value π.

ii) To avoid the divergences at $x = 0$, we set cutoff for the domain by 10^{-8}, then $x > 10^{-8}$. To avoid the infinity, we set an upper limit for the domain to be 30, then $10^{-8} < x < 30$, which is reasonable as following:

$$\int_{30}^{\infty} e^{-x} \ln(x) \mathrm{d}x < \int_{30}^{\infty} e^{-x} x = e^{-x}(-1-x)|_{30}^{\infty} \approx 2.9 \times 10^{-12} \tag{5}$$

the cutoff affection is small. We set the number of samples per trial $n = 1000$, and the number of trials $m = 10000$, and get the monte carlo integration

$$\int_{0}^{\infty} e^{-x} \ln(x) \mathrm{d}x \approx \int_{10^{-8}}^{30} e^{-x} \ln(x) \mathrm{d}x \approx -0.579221 \pm 0.00219467. \tag{6}$$

Plot:

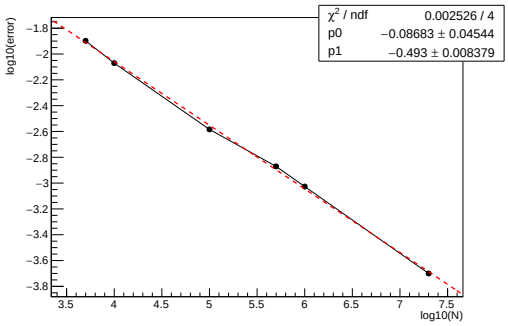

Figure 1: The relationship between the error and N.

Relevant code:

For the initialization of the Monte Carlo integration

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

MonteCarloIntegrate::MonteCarloIntegrate(int dim, vector<double> x1,
 vector<double> x2, double (*f)(const vector<double> &x)) {

        _x1 = x1;
        _x2 = x2;
        _f  = f;
        _n  = 1000;
        _alg = 0;
        _dim = dim;
        _trial = 1000;

        _vol = 1;
        for(int i=0; i<_dim; i++) {
                _vol *= _x2[i]-_x1[i];
        }

        srand(time(nullptr));
}

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

For the Monte Carlo integration

```cpp
//-------------------------------------------------------------------------//

double MonteCarloIntegrate::cal_once() {

        if(!check()) return 0;

        double tmpy = 0;
        double sumy = 0;
        double sumyy = 0;

        for(int i=0; i<_n; i++) {
                vector<double> x_tmp;
                for(int j=0; j<_dim; j++) {
                        //srand(time(nullptr));
                        double tmp = _x1[j] + (_x2[j]-_x1[j])*rand()/RAND_MAX;
                        x_tmp.push_back(tmp);
                }

                tmpy = _f(x_tmp);

                sumy += tmpy;
        }

        return sumy*_vol/_n;
}

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

For the integrands

```cpp
//-------------------------------------------------------------------------//

double f1(const vector<double> &x) {

        return sqrt(4-x[0]*x[0]);
}

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

double f2(const vector<double> &x) {

        return 1.0/sqrt(4-x[0]*x[0]);
}

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

double f3(const vector<double> &x) {

        return exp(-x[0])*log(x[0]);
}

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

(2) Second, use the two starter programs (`rw2d.m`/`generate_rw.m`) to generate random walks on the square lattice in two dimensions (the same programs and their outputs have been discussed in class already). For this lab, modify the programs (or write your own equivalent ones) to walks of random step length $0 < d < 1$ in (continuously) random directions (i.e., the walk is still in two dimensions, but no longer on a lattice). Analyze the mean square displacement $\langle r_n^2 \rangle$ for n-step continuous random walks, and calculate the mean fluctuation (standard deviation) of r_n^2 (i.e., the square root of the variance of r n2 defined by $\sqrt{\langle (r_n^2)^2 \rangle - \langle r_n^2 \rangle^2}$. Is the latter of the same order as $\langle r_n^2 \rangle$ itself? If that is true, then the fluctuation is just as large as the mean, and thus you cannot tell its statistical properties by generating just a few random walks, however long (i.e., many steps) they may be.

Physics explanation:

For the random walk with **unit step length**, the results are shown in Fig. 2.

- The left pad shows the average of r^2 ($\langle r^2 \rangle$) depending on time t. We fit it with a straight line and get the slope roughly equal to 1 and intercept about 0.
- The middle pad shows the standard deviation of r^2 ($\sigma(r^2)$) depending on time t, we can see the standard deviation $\sigma(r^2)$ is nearly equal to the average $\langle r^2 \rangle$. Then the fluctuation is just as large as the mean, and thus you cannot tell its statistical properties by generating just a few random walks, however long (i.e., many steps) they may be.
- The right pad shows $\log_{10}\langle r^2 \rangle$ depending on $\log_{10} t$. We fit it with a straight line and get the slope roughly equal to 1 and intercept about 0, which support the linear fit on the left pad.

For the random walk with **continuous step length**, the results are shown in Fig. 3.

- The left pad shows the average of r^2 ($\langle r^2 \rangle$) depending on time t, which is smaller than the unit step length case. We fit it with a straight line and get the slope roughly equal to 0.32 and intercept about 0.27.
- The middle pad shows the standard deviation of r^2 ($\sigma(r^2)$) depending on time t, we can see the standard deviation $\sigma(r^2)$ is nearly equal to the average $\langle r^2 \rangle$. Then the fluctuation is just as large as the mean, and thus you cannot tell its statistical properties by generating just a few random walks, however long (i.e., many steps) they may be.
- The right pad shows $\log_{10}\langle r^2 \rangle$ depending on $\log_{10} t$. We fit it with a straight line and get the slope roughly equal to 1 and intercept about 0, which support the linear fit on the left pad.

Plots:

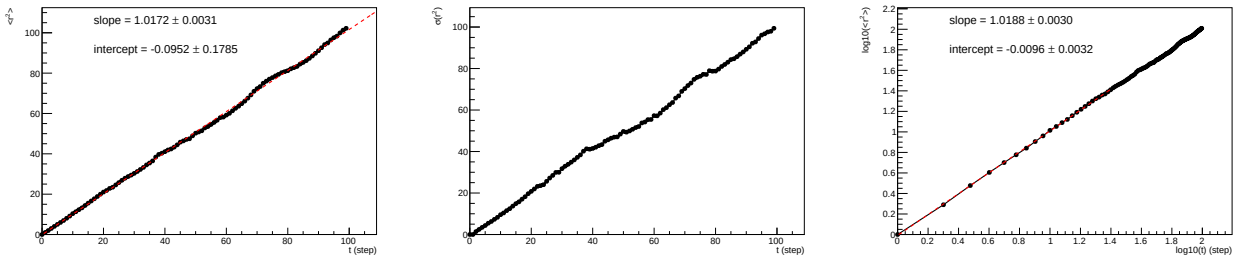

Figure 2: Random walk with **unit step length**: left pad: average of r^2 ($\langle r^2 \rangle$) depending on time t; middle pad: standard deviation of r^2 ($\sigma(r^2)$) depending on time t; right pad: $\log_{10}\langle r^2 \rangle$ depending on $\log_{10} t$

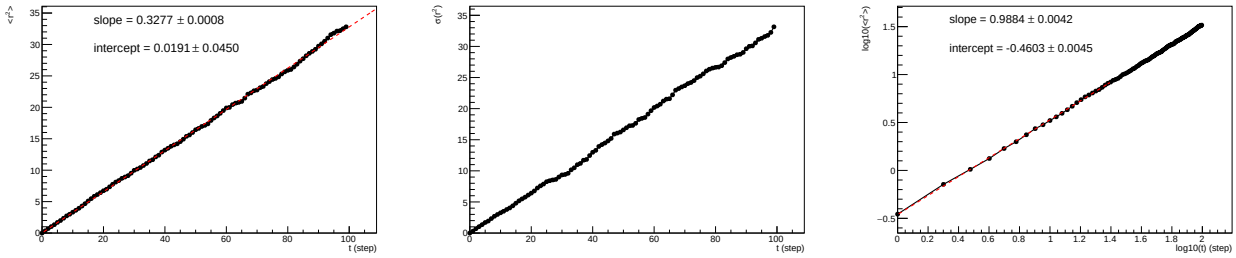

Figure 3: Random walk with **continuous step length**: left pad: average of r^2 ($\langle r^2 \rangle$) depending on time t; middle pad: standard deviation of r^2 ($\sigma(r^2)$) depending on time t; right pad: $\log_{10}\langle r^2 \rangle$ depending on $\log_{10} t$

Relevant code:

For the initialization of the random walk

```
1  //------------------------------------------------------------------------//
2
3  RandomWalk2D::RandomWalk2D(int n_step, int n_walk, int mod) {
4
5          //srand(time(nullptr));
6
7          _n_step = n_step;
8          _n_walk = n_walk;
9          _mod = mod;
10         _d = 1;
11         _min_step = 0.1;
12
13         for(int i=0; i<_n_step; i++) {
14                 _r2.push_back(0);
15                 _r4.push_back(0);
16                 _r2_std.push_back(0);
17                 _t.push_back(i);
18                 _good_walk.push_back(0);
19         }
20 }
21
22 //------------------------------------------------------------------------//
```

For the unit step length random walk

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

void RandomWalk2D::cal_rw_1() {

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

        _x.push_back(0);
        _y.push_back(0);

        for(int i=1; i<_n_step; i++) {

                double tmprand = 1.0*rand()/RAND_MAX;

                if(tmprand<0.25) {
                        _x.push_back(_x[i-1]+1);
                        _y.push_back(_y[i-1]);
                } else if(tmprand<0.5) {
                        _x.push_back(_x[i-1]);
                        _y.push_back(_y[i-1]+1);
                } else if(tmprand<0.75) {
                        _x.push_back(_x[i-1]-1);
                        _y.push_back(_y[i-1]);
                } else {
                        _x.push_back(_x[i-1]);
                        _y.push_back(_y[i-1]-1);
                }
        }
}

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

For the continuous step length random walk

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

void RandomWalk2D::cal_rw_d() {

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

        _x.push_back(0);
        _y.push_back(0);

        for(int i=1; i<_n_step; i++) {

                double d = _d*1.0*rand()/RAND_MAX;

                double tmprand = 1.0*rand()/RAND_MAX;

                if(tmprand<0.25) {
                        _x.push_back(_x[i-1]+d);
                        _y.push_back(_y[i-1]);
                } else if(tmprand<0.5) {
                        _x.push_back(_x[i-1]);
                        _y.push_back(_y[i-1]+d);
                } else if(tmprand<0.75) {
                        _x.push_back(_x[i-1]-d);
                        _y.push_back(_y[i-1]);
```

```
                } else {
                        _x.push_back(_x[i-1]);
                        _y.push_back(_y[i-1]-d);
                }
        }
}

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

For the calculation of $\langle r_n^2 \rangle$ and $\sigma(r_n^2)$

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

void RandomWalk2D::cal() {

        for(int i_walk=0; i_walk<_n_walk; i_walk++) {

                if(_mod == 0) cal_rw_1();
                if(_mod == 1) cal_rw_d();
                if(_mod == 2) {
                        if(!cal_saw()) continue;
                }

                for(int i_step=0; i_step<_x.size(); i_step++) {
                        double r2tmp = _x[i_step]*_x[i_step] + _y[i_step]*_y[i_step];
                        _r2[i_step] += r2tmp;
                        _r4[i_step] += r2tmp*r2tmp;
                        _good_walk[i_step] += 1.0;
                }
        }
        for(int i_step=0; i_step<_n_step; i_step++) {
                if(_good_walk[i_step] == 0) continue;
                _r2[i_step] /= _good_walk[i_step];
                _r4[i_step] /= _good_walk[i_step];
                _r2_std[i_step] = sqrt(_r4[i_step]-_r2[i_step]*_r2[i_step]);
        }
}

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

(3) Finally, use the starter programs `saw2d.m/generate_saw.m` (or your own equivalent codes) to simulate self-avoiding walks (SAW) on the square lattice. Obtain an estimate of the Flory exponent , defined in terms of the relation $\langle r_n^2 \rangle = \text{const} \times n^{2\nu}$ between the mean squared displacement and the length n of the SAW. Also study the fluctuations of $\langle r_n^2 \rangle$ as you did for the random walks. Make sure that you understand the algorithm that implements self-avoidance, i.e., how the algorithm keeps the desired properties of the ensemble (namely, equal probability for each SAW of equal number of steps).

Physics explanation:

For the **Self-avoiding walk (SAW)**, the results are shown in Fig. 4.

- The left pad shows the average of r^2 ($\langle r^2 \rangle$) depending on time t. (Ignore the fitting).
- The middle pad shows the standard deviation of r^2 ($\sigma(r^2)$) depending on time t, we can see the standard deviation $\sigma(r^2)$ is smaller than, but of the same order as, the average $\langle r^2 \rangle$. Then the fluctuation is just as large as the mean, and thus you cannot tell its statistical properties by generating just a few random walks, however long (i.e., many steps) they may be.
- The right pad shows $\log_{10}\langle r^2 \rangle$ depending on $\log_{10} t$. We fit it with a straight line and get the slope roughly equal to 1.445 and intercept about 0. Then the Flory exponet is

$$2\nu \approx 1.445, \quad \Rightarrow \quad \nu \approx 0.723. \tag{7}$$

The algorithm remembers the direction of the previous step `nodir`, and avoid going back in the current random walking by `if(r >= nodir)r++;`. The walk has equal probability to go to the rest three directions. The algorithm remembers the all previously occupied lattices. If it goes to any previously occupied lattices, the walking stops, which can ensure every SAW walk has the equal probability.

Plots:

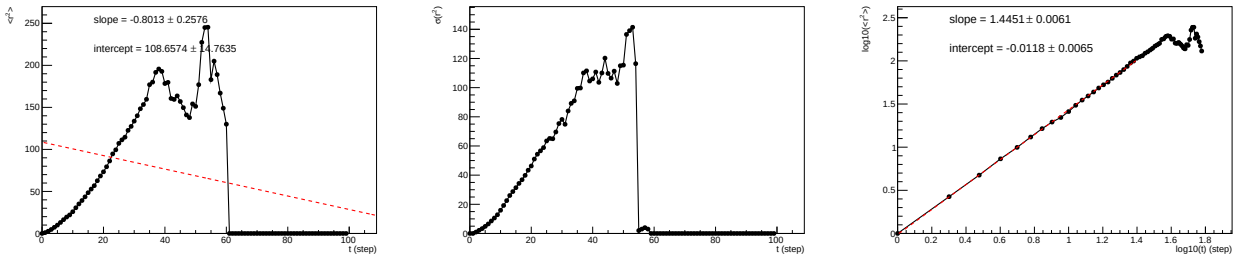

Figure 4: **Self-avoiding walk (SAW)**: left pad: average of r^2 ($\langle r^2 \rangle$) depending on time t; middle pad: standard deviation of r^2 ($\sigma(r^2)$) depending on time t; right pad: $\log_{10}\langle r^2 \rangle$ depending on $\log_{10} t$

Relevant code:

For the self-avoiding walk (SAW)

```
1  //------------------------------------------------------------------------//
2
3  bool RandomWalk2D::cal_saw() {
4
5          vector< vector<bool> > occupied;
6          for(int i=0; i<_n_step*2+1; i++) {
7                  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;
}

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