

Random Walks and Diffusion

Monte Carlo Simulation

To simulate is to mimic or imitate. A simulation is especially useful when it is simpler or easier to control than the system being imitated. A computer simulation is the execution of a computer program that imitates the properties and behavior of another system.



Figure 1: Casino Monte Carlo from http://en.wikipedia.org/wiki/Monte_Carlo_Casino.

Computer simulations that make essential use of random sequences are often called Monte Carlo simulations. The name “Monte Carlo” was first used by scientists studying the propagation of neutrons through matter. They simulated the collisions of neutrons with atomic nuclei as random changes in velocity and direction sampled from quantum mechanical probability distributions. The simulations were given the code name “Monte Carlo” after the famous “Casino Monte Carlo” (gambling house in Mount Charles district) in the tiny European country Monaco, shown in Fig. 1. A computational experiment that uses Monte Carlo methods is often called a “Monte Carlo”.

Monte Carlo Sampling

There are many types of Monte Carlo algorithms, and they have been applied to many different types of systems. Monte Carlo methods are commonly used to simulate the properties or behavior of systems that can exist in a very large number of states.

Consider, for example, a system of 100 electrons, each of which can exist in two states, spin up \uparrow or spin down \downarrow . The total number of states of this relatively small system is $= 2^{100} = 1.27 \times 10^{30}$. To enumerate all these states at the rate of one billion per second will take 4.02×10^{13} years! If an exact analytic solution cannot be found, then randomly sampling typical states of the system might be used to find approximate answers.

Markov-Chain Monte Carlo

A Markov chain is a sequence of elements chosen from a fixed set according to a probabilistic rule. The chain is constructed by adding elements sequentially. Given the most recently added element of the chain, the choice of the next element depends only on the most recent addition and not on the previous history of the construction process.

Suppose that x and y are any two elements of the set. The Markov process is defined by a transition probability function

$$T(x \rightarrow y) , \quad \sum_y T(x \rightarrow y) = 1 . \quad (1)$$

Note that T is a function of ordered pairs of elements of the set. It does not depend on the step number of x or y in the chain.

If we imagine that the elements of the set represent the possible states or configurations of a physical system, and that a step in the chain corresponds to the system moving from one state to another, then the future of the system depends only on its present state, and not on its past.

Markov chains can be used to simulate a system in equilibrium. Suppose that x represents a state of the system and $P(x)$ is the probability distribution of equilibrium states, and suppose that $P(x)$ is invariant under a Markov transition probability $T(x \rightarrow y)$. A Markov process is called *ergodic* if the elements of any sufficiently long chain occur with probability $P(x)$. Ergodic Markov chains form the basis for the Markov-chain Monte Carlo method for simulating physical systems in statistical equilibrium.

Random Walk in One Dimension

Random walks provide simple examples of Markov chains.

Consider a very simple random walk in one dimension, described in the first chapter of Reif[?]. Suppose that the walker can occupy the sites of a lattice with constant lattice spacing. Given any location on the lattice, the walker decides on whether to move left or right by tossing a coin. The walker moves left if the coin lands heads up, and right if it lands heads down. The transition probability is

$$T(x \rightarrow y) = \begin{cases} \frac{1}{2} & \text{if } y = x - 1, \\ \frac{1}{2} & \text{if } y = x + 1, \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

To define an equilibrium probability distribution, suppose that the lattice is periodic with L lattice points arranged on a circle. The limit $L \rightarrow \infty$ can be taken later to compute properties of an infinite lattice. It can be shown a constant probability distribution

$$P(x) = \frac{1}{L} \quad (3)$$

is invariant under the Markov transition probability defined by Eq. 2. This is intuitively reasonable: the random walker will eventually visit every lattice site approximately the same number of times.

Statistical Behavior of 1-D Walks

The walker's position x_n after n steps depends on the sequence of coin tosses and cannot be predicted

$$x_n = \sum_{i=1}^n s_i , \quad (4)$$

where $s_i = \pm 1$.

The average over a large number of n -step walks

$$\langle x_n \rangle = 0 \quad (5)$$

by symmetry, because \pm are equally likely!

But each $x_n \neq 0$ and in fact increases with n . To see this, compute the variance

$$\langle x_n^2 \rangle = \left\langle \sum_{i=1}^n \sum_{j=1}^n s_i s_j \right\rangle \quad (6)$$

$$= \left\langle \sum_{i=1}^n s_i^2 \right\rangle + \left\langle \sum_{i=1}^n \sum_{\substack{j=1 \\ j \neq i}}^n s_i s_j \right\rangle \quad (7)$$

$$= n, \quad (8)$$

where the double sum with $i \neq j$ must average to zero by symmetry. The root-mean-square (r.m.s.) displacement in n steps is

$$\sqrt{\langle x_n^2 \rangle} = \sqrt{n}. \quad (9)$$

The *diffusion constant* D for the walk is defined by

$$\langle x_n^2 \rangle = 2Dn. \quad (10)$$

Thus $D = \frac{1}{2}$ for this one-dimensional walk.

It not difficult to show that the variance of the variance

$$\langle x_n^4 \rangle - \langle x_n^2 \rangle^2 = (3n^2 - 2n) - n^2. \quad (11)$$

Thus the average separation of two random walkers grows like

$$\sqrt[4]{2n^2 - 2n} \sim \sqrt{\sqrt{2n}} = 1.189\sqrt{n}, \quad (12)$$

a little faster than the r.m.s. displacement!

C++ Program to Simulate 1-D Walks

_____ Program 1: <http://www.physics.buffalo.edu/phy410-505/topic2/rwalk.cpp> _____

```
#include <math>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <vector>
using namespace std;

#include "random.hpp"
#include "datafit.hpp"
using namespace cpl;
```

```

vector<double> model(      // model function to fit a linear model
    const double step
) {
    vector<double> f(2);
    f[0] = 1;
    f[1] = step;
    return f;
}

int main() {
    cout << " Random walk in 1 dimension\n"
         << " -----\n"
         << " Enter number of walkers: ";
    int n_walkers;
    cin >> n_walkers;
    cout << " Enter number of steps: ";
    int n_steps;
    cin >> n_steps;

    // vector of walker positions initialized to zero
    vector<double> x(n_walkers);

    // data vectors
    vector<double> steps(n_steps);    // to save step number i (time)
    vector<double> x2ave(n_steps);    // to accumulate x^2 values
    vector<double> sigma(n_steps);    // to accumulate fluctuations in x^2

    // create a random number generator object
    Random r;
    r.set_seed_time();
    cout << " using " << r.get_algorithm()
         << " and seed " << r.get_seed() << endl;

    // Loop over walkers
    for (int walker = 0; walker < n_walkers; walker++) {

        // Loop over number of steps
        for (int step = 0; step < n_steps; step++) {

            // take a random step
            if (r.rand() < 0.5)
                x[walker] += 1;
            else
                x[walker] -= 1;

            // accumulate data
            steps[step] = step + 1;
            double xx = x[walker] * x[walker];
            x2ave[step] += xx;
            sigma[step] += xx * xx;
        }
    }
}

```

```

}

// average the squared displacements and their variances
for (int step = 0; step < n_steps; step++) {
    x2ave[step] /= n_walkers;
    sigma[step] /= n_walkers;
}
for (int step = 0; step < n_steps; step++) {
    sigma[step] = sqrt(sigma[step] - x2ave[step] * x2ave[step]);
    if (sigma[step] == 0) // happens for first step!
        sigma[step] = 1;
    if (n_walkers > 1)
        sigma[step] /= sqrt(double(n_walkers - 1));
}

// fit data to a straight line
Datafit df;
df.set_x_vector(steps);
df.set_y_vector(x2ave);
df.set_y_error_vector(sigma);
df.set_model_function(model);
df.fit();
vector<double> ab = df.get_parameter_vector();
vector<double> sigma_ab = df.get_sigma_vector();
double chisqr = df.get_chi_square();

cout << " Fit to a straight line <x^2> = a + b n\n"
    << " Intercept a = " << ab[0] << " +- " << sigma_ab[0] << '\n'
    << " Slope    b = " << ab[1] << " +- " << sigma_ab[1] << '\n'
    << " Chisqr/dof = " << chisqr / (n_steps - 2.0) << endl;

// store in file for plotting
ofstream file("rwalk.data");
for (int step = 0; step < n_steps; step++)
    file << steps[step] << '\t' << x2ave[step]
        << '\t' << sigma[step] << '\n';
file.close();
cout << " t, <x^2>, sigma in file rwalk.data" << endl;
}

```

Random Walks in Two Dimensions

Random walks can be used to model non-equilibrium ensembles.