

Metropolis Monte Carlo

Nonuniform Sampling

The simplest Monte Carlo applications make use of uniformly distributed random sequences. Random number generator algorithms normally produce sequences of integers from a finite set chosen with equal probabilities. A sequence from a large set of contiguous integers can be mapped to the unit interval $[0, 1]$ of real numbers to give a sequence of random deviates.

Uniform deviates can often be transformed by a simple change of variable $x \rightarrow y(x)$ to produce other probability distributions[1]

$$P(y) = P(x) \left| \frac{dx}{dy} \right|, \quad \int dy P(y) = \int dx P(x) = 1. \quad (1)$$

Uniform line segment: A uniform deviate $0 < x < 1$ can be transformed to the interval $[a, b]$ using the map

$$y(x) = a + (b - a)x, \quad \left| \frac{dx}{dy} \right| = \frac{1}{|b - a|}. \quad (2)$$

$$P(x) = \begin{cases} 1 & \text{for } 0 < x < 1 \\ 0 & \text{otherwise} \end{cases} \Rightarrow P(y) = \begin{cases} \frac{1}{|b-a|} & \text{for } a < y < b \\ 0 & \text{otherwise} \end{cases}. \quad (3)$$

Exponential distribution: To obtain the exponential distribution

$$P(y) = \frac{1}{\lambda} e^{-y/\lambda}, \quad 0 < y < \infty \quad (4)$$

use the map

$$y(x) = -\lambda \log(x), \quad \left| \frac{dx}{dy} \right| = \frac{x}{\lambda} = \frac{e^{-y/\lambda}}{\lambda} = P(y). \quad (5)$$

$$P(x) = \begin{cases} 1 & \text{for } 0 < x < 1 \\ 0 & \text{otherwise} \end{cases} \Rightarrow P(y) = \begin{cases} \frac{e^{-y/\lambda}}{\lambda} & \text{for } 0 < y < \infty \\ 0 & \text{otherwise} \end{cases}. \quad (6)$$

Gaussian distribution: This is also called the normal distribution because it occurs so frequently in statistical applications:

$$P(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/(2\sigma^2)}, \quad (7)$$

which has mean value $\langle x \rangle = \mu$ and variance $\langle (x - \langle x \rangle)^2 \rangle = \sigma^2$. The Gaussian distribution cannot be inverted using elementary functions, like the logarithm inverts the exponential distribution. However, consider the product of identical Gaussian distributions in x and y :

$$\int dx \int dy P(x)P(y) = \frac{1}{2\pi\sigma^2} \int dx \int dy e^{-[(x-\mu)^2 + (y-\mu)^2]/(2\sigma^2)} = \int_0^{2\pi} \frac{d\theta}{2\pi} \int_0^\infty \frac{dr^2}{2\sigma^2} e^{-r^2/(2\sigma^2)}, \quad (8)$$

where the cartesian pair x, y have been changed to polar coordinates

$$x - \mu = r \cos \theta, \quad y - \mu = r \sin \theta. \quad (9)$$

Thus the product of two identical Gaussians is equivalent to the product of a uniform distribution in $[0, 2\pi]$ with an exponential distribution in the the variable r^2 . To generate independent normally

distributed points in x and y , generate pairs of random deviates r_1, r_2 , determine the corresponding random polar coordinates

$$\theta = 2\pi r_1, \quad r = \sqrt{-2\sigma^2 \log r_2}, \quad (10)$$

and determine the corresponding x and y values. This is called the Box-Muller algorithm.

The Metropolis Algorithm

While numerous ingenious algorithms for generating random sequences with non-uniform probability distributions have been invented, few of these have achieved the degree of popularity enjoyed by the method of Metropolis *et al.*[2]. The method is sometimes called the Metropolis-Hastings algorithm to acknowledge the work of Hasting[3] who generalized it for mathematical statistics applications.

The Metropolis algorithm can be viewed as a type of random walk. Suppose that we are to generate a random sequence distributed according to the probability function $P(\vec{x})$. Imagine a mountainous terrain on a d -dimensional plane, with $P(\vec{x})$ representing the elevation of the surface at point \vec{x} . A random walker is started at some point \vec{x}_0 . The walker then steps successively from point to point and generates a sequence of points which we represent as follows:

$$\begin{aligned} \vec{x}_0 &\rightarrow \vec{x}_{1_t} \rightarrow \vec{x}_{2_t} \rightarrow \cdots \rightarrow \vec{x}_{N_t} \rightarrow \vec{x}_1 \rightarrow \vec{x}_{1_2} \rightarrow \cdots \rightarrow \vec{x}_{1_\nu} \rightarrow \vec{x}_2 \rightarrow \vec{x}_{2_2} \rightarrow \cdots \\ &\rightarrow \vec{x}_{2_\nu} \rightarrow \vec{x}_3 \rightarrow \cdots \rightarrow \vec{x}_4 \rightarrow \cdots \rightarrow \vec{x}_{N-1} \rightarrow \vec{x}_{(N-1)_2} \rightarrow \cdots \rightarrow \vec{x}_{(N-1)_\nu} \rightarrow \vec{x}_N. \end{aligned} \quad (11)$$

The first N_t steps starting at \vec{x}_0 are called *thermalization steps*: these steps will, in general, not be characteristic of the desired distribution $P(\vec{x})$, and they are to be discarded. The thermalization steps can be viewed as being exploratory in nature. The walker needs to take these initial steps to scout the terrain and learn where the highest mountains and peaks are located. After thermalization, the walker begins to generate the random sequence $\vec{x}_1, \vec{x}_2, \dots, \vec{x}_N$ to be used in the Monte Carlo application. Note that not every step after \vec{x}_1 is used in the sequence. There is a *sampling frequency* $\nu \geq 1$ which determines the number $\nu-1$ of steps that are discarded before the next point is added to the random sequence. The reason that a sampling frequency $\nu > 1$ might be required is that successive steps made by the walker are often highly correlated. This will be the case if, for example, the walker's step size is much smaller than typical distances in the terrain: A truly random sequence will have separations of all possible magnitudes between successive random points; and, to emulate this property, the walker will have to take and discard some number of steps ν after each point in the sequence.

We now specify the algorithm used by the walker in taking a single step. Suppose that the walker is at some point \vec{x}_{current} . Then, the next point \vec{x}_{next} in the sequence (11) is determined as follows:

1. Choose a *trial* point \vec{x}_{trial} at random in a neighborhood of \vec{x}_{current} . There is considerable freedom in choosing this random point. In one dimension, a fixed *maximum step size* δ might be used, and x_{trial} chosen uniformly at random in the interval $[x_{\text{current}} - \delta, x_{\text{current}} + \delta]$. In d -dimensions, a random point could be chosen in a sphere of fixed radius δ centered at \vec{x}_{current} . The one requirement on the choice of \vec{x}_{trial} is that the probability $P_{\text{try}}(\vec{x}_{\text{current}} \rightarrow \vec{x}_{\text{trial}})$ should be *symmetric*, i.e., $P_{\text{try}}(\vec{x} \rightarrow \vec{y}) = P_{\text{try}}(\vec{y} \rightarrow \vec{x})$ for every pair of points $\{\vec{x}, \vec{y}\}$ in the space.
2. The ratio $r = P(\vec{x}_{\text{trial}})/P(\vec{x}_{\text{current}})$ is calculated, and the following tests are performed:
 - (a) If $r \geq 1$, the walker is moving "uphill" towards a region of higher probability, or is maintaining elevation: the trial step is then accepted, and $\vec{x}_{\text{next}} = \vec{x}_{\text{trial}}$. Since the step is definitely accepted if $r \geq 1$, the *acceptance probability* $P_{\text{accept}}(\vec{x}_{\text{current}} \rightarrow \vec{x}_{\text{trial}}) = 1$ in this case.
 - (b) If $r < 1$, the walker is moving "downhill" towards a region of lower probability: the trial step must not be rejected outright because the walker might then not be able to cross a valley and

explore another mountain elsewhere; the downhill move cannot be accepted outright either because there would then be no distinction between uphill and downhill. The step is therefore accepted only if the downward vertical displacement is not too large: a random number n_{random} is generated with uniform probability in the interval $[0, 1]$; if $r > n_{\text{random}}$ then the move is accepted and $\vec{x}_{\text{next}} = \vec{x}_{\text{trial}}$; otherwise the trial step is rejected and $\vec{x}_{\text{next}} = \vec{x}_{\text{current}}$. $P_{\text{accept}}(\vec{x}_{\text{current}} \rightarrow \vec{x}_{\text{trial}}) = r$ in this case.

Proof that Metropolis walkers thermalize

Imagine that we have a large ensemble of independent Metropolis walkers that are distributed with density $\rho(\vec{x})$ in the d -dimensional plane underlying the mountainous terrain. Suppose that these walkers all simultaneously take a step using the Metropolis algorithm. We wish to show that the ensemble density tends to become proportional to the probability function $P(\vec{x})$ and then remain invariant. According to the *ergodic hypothesis*, the average properties of an ensemble of systems should be the same as the average over time of the corresponding quantities in a single system in thermal equilibrium. A proof that the equilibrium density of an ensemble of walkers is proportional to $P(\vec{x})$ then implies that the sequence of points generated by a single Metropolis walker is also distributed asymptotically according to $P(\vec{x})$.

Recall that the transition probability $T(\vec{x} \rightarrow \vec{y})$ that a Metropolis walker moves from \vec{x} to \vec{y} is determined by two consecutive and independent decisions:

$$T(\vec{x} \rightarrow \vec{y}) = P_{\text{try}}(\vec{x} \rightarrow \vec{y}) \times P_{\text{accept}}(\vec{x} \rightarrow \vec{y}). \quad (12)$$

The Metropolis algorithm is an example of a Markov chain process. Since $P_{\text{try}}(\vec{x} \rightarrow \vec{y}) = P_{\text{try}}(\vec{y} \rightarrow \vec{x})$ is symmetric in the Metropolis algorithm, we have

$$\frac{T(\vec{x} \rightarrow \vec{y})}{T(\vec{y} \rightarrow \vec{x})} = \frac{P_{\text{accept}}(\vec{x} \rightarrow \vec{y})}{P_{\text{accept}}(\vec{y} \rightarrow \vec{x})} = \frac{P(\vec{y})}{P(\vec{x})}. \quad (13)$$

The second equality in this equation can be verified by considering the two cases: (i) $P(\vec{y}) > P(\vec{x})$, for which $P_{\text{accept}}(\vec{x} \rightarrow \vec{y}) = 1$ and $P_{\text{accept}}(\vec{y} \rightarrow \vec{x}) = P(\vec{x})/P(\vec{y})$; and (ii) $P(\vec{y}) < P(\vec{x})$ for which $P_{\text{accept}}(\vec{x} \rightarrow \vec{y}) = P(\vec{y})/P(\vec{x})$ and $P_{\text{accept}}(\vec{y} \rightarrow \vec{x}) = 1$.

We now compute the change in the number density of walkers at \vec{x} when all walkers in the ensemble simultaneously take a step:

$$\Delta\rho(\vec{x}) = \int d^d y [\rho(\vec{y})T(\vec{y} \rightarrow \vec{x}) - \rho(\vec{x})T(\vec{x} \rightarrow \vec{y})] = \rho(\vec{x}) \int d^d y T(\vec{y} \rightarrow \vec{x}) \left[\frac{\rho(\vec{y})}{\rho(\vec{x})} - \frac{P(\vec{y})}{P(\vec{x})} \right]. \quad (14)$$

This formula clearly shows that if $\rho(\vec{y})/\rho(\vec{x})$ exceeds its “equilibrium value” $P(\vec{y})/P(\vec{x})$, i.e., if there are too many walkers at \vec{y} and too few at \vec{x} , then $\Delta\rho(\vec{x})$ receives a positive contribution from the neighborhood of \vec{y} . Conversely, if there are too few walkers at \vec{y} and too many at \vec{x} , then there is a net flow from \vec{x} to \vec{y} . When ρ is proportional to P , the ensemble is in equilibrium: there is no net change in the number density at any point, which then satisfies the *detailed balance condition*: $\rho(\vec{y})T(\vec{y} \rightarrow \vec{x}) = \rho(\vec{x})T(\vec{x} \rightarrow \vec{y})$.

Practical considerations in a Metropolis walk

There are three major considerations in applying the Metropolis algorithm to any particular problem:

1. How large a step size δ should the walker attempt? If the step size is too small, then most steps will be accepted no matter what the probability function is. If the step size is too large, too many steps

will be rejected, and the walker will find it difficult to explore steep terrain. Since neither extreme is desirable, a step size should be chosen by trial and error so that the *acceptance ratio* of the number of steps accepted to the total number of trial steps is on the order of 0.5.

2. How many thermalization steps N_t should be taken and discarded? There is, in general, no rigorous answer to this question. The appropriate number of steps should be determined by trial and error. The random sequence might be divided into bins of equal size, and the average values of physical observables monitored in these bins to determine when they have settled down to their equilibrium values.
3. How large should the sampling frequency ν be chosen? A simple criterion is to choose ν sufficiently large so that the Monte Carlo error estimate is consistent with a statistical (standard deviation) estimate from making a series of independent runs. Sequential correlations between successive sampling points can also be monitored and ν adjusted to minimize them.

C++ Program for Metropolis Algorithm

The following code uses the Metropolis algorithm to generate a sum of two Gaussian distributions.

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

```
#include <cmath>
#include <cstdlib>
#include <cstdio>
#include <ctime>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
using namespace std;

#include "random.hpp"
using namespace cpl;

const string title("Metropolis Algorithm for Two Gaussians");

Random rg; // random number generator

class Gnuplot { // Gnuplot object
public:

    Gnuplot(){

        gnuplot_pipe =
# ifdef _MSC_VER
        _popen("C:\\gnuplot\\bin\\pgnuplot.exe", "w");
# else
        popen("/usr/bin/gnuplot", "w");
# endif
```

```

    if (!gnuplot_pipe) {
        cerr << "cannot open pipe to gnuplot" << endl;
        exit(EXIT_FAILURE);
    }
}

void command(
    const string command
) {
    fprintf(gnuplot_pipe, "%s\n", command.c_str());
    fflush(gnuplot_pipe);
}

void close() {
    fprintf(gnuplot_pipe, "exit\n");
    fclose(gnuplot_pipe);
}

private:

    FILE *gnuplot_pipe;
};

// Gaussians

double
A[2]= {1.0, 0.75},           // amplitudes
sigma[2] = {1.0, 0.5},     // widths
center[2] = {0.0, 6.0};    // positions of centers

double P (double x) {

    double p = 0;
    for (int i = 0; i < 2; i++)
        p += A[i]*exp(-(x-center[i])*(x-center[i])/(2*sigma[i]*sigma[i]));
    return p;
}

// Metropolis
double xWalker0 = -5;       // initial position
double xWalker = xWalker0; // current position
double delta = 1.0;        // step size

bool MetropolisStep ( ) { // is the step accepted?

    double xTrial = xWalker + delta*(2*rg.rand()-1);
    double ratio = P(xTrial)/P(xWalker);

    if (rg.rand() < ratio) {
        xWalker = xTrial;
        return true;
    }
}

```

```

    } else
        return false;
}

int stepsToDiscard = 0;           // steps to discard
unsigned long steps = 0;         // steps so far
unsigned long accepts = 0;       // steps accepted

void MonteCarloStep ( ) {

    for (int i = 0; i < stepsToDiscard; i++)
        MetropolisStep();
    if (MetropolisStep())
        ++accepts;
    ++steps;
}

double xMax = 10.0;              // maximum extent for plotting
const int bins = 400;           // number of bins in histogram
vector<double> histogram(bins); // histogram bins

Gnuplot gnuplot;                // initialize Gnuplot instance
int stepsToSkip = 200;          // when plotting

void takeStep ( ) {

    clock_t t0 = clock();        // to time animation

    for (int i = 0; i <= stepsToSkip; i++) {
        MonteCarloStep();
        int bin = int((xWalker+xMax)/(2*xMax)*bins);
        if (bin >= 0 && bin < bins)
            ++histogram[bin];
    }

    string file_name("gnuplot.data");
    ofstream data_file(file_name.c_str());

    for (int bin = 0; bin < bins; bin++) {
        double x = -xMax + (bin + 0.5) * (2*xMax)/bins;
        data_file << x << '\t' << histogram[bin] << '\n';
    }
    data_file.close();

    ostringstream os;
    double percent = int(accepts / double(steps) * 100.0);
    os << "set title \"' << title
        << ", Metropolis steps = " << steps
        << ", accepts = " << percent << "%\''";
    gnuplot.command(os.str());
    gnuplot.command("plot \"' + file_name + "\" with impulses");
}

```

```

while (true) {
    // slow down to 15 frames/sec
    if ((clock() - t0)/double(CLOCKS_PER_SEC) > 1.0 / 15)
        break;
}

int main () {

    cout << " " << title << endl;
    cout << " Enter starting position x_0: ";
    cin >> xWalker;
    cout << " Enter step size delta: ";
    cin >> delta;
    cout << " Enter number of Monte Carlo steps: ";
    int n;
    cin >> n;
    cout << " Enter number of Metropolis steps per Monte Carlo step: ";
    cin >> stepsToDiscard;
    cout << " Enter number of Monte Carlo steps to skip between each plot: ";
    cin >> stepsToSkip;

    // set random number generator seed
    rg.set_seed_time();
    cout << " Using " << rg.get_algorithm()
        << " and seed " << rg.get_seed() << endl;

    for (int i = 0; i < n; i++)
        takeStep();
}

```

Homework Problem

Write a program to generate the normal Gaussian distribution

$$P(x) = \frac{e^{-x^2/2}}{\sqrt{2\pi}} \quad (15)$$

using two algorithms, the Box-Muller change of variable algorithm, and the Metropolis algorithm. Compare the efficiencies of the two algorithms by measuring their rates of convergence to $P(x)$ as functions of the number of random deviates used.

References

- [1] W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, “Numerical Recipes in C” (Cambridge University Press 1992), §7.2 Transformation Method: Exponential and Normal Deviates,

<http://www.nrbook.com/a/bookcpdf/c7-2.pdf>.

- [2] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller and E. Teller, "Equation of State Calculations by Fast Computing Machines", *J. Chem. Phys.* **21**, 1087 (1953), <http://link.aip.org/link/?JCPA6/21/1087/1>.
- [3] W.K. Hastings, "Monte Carlo sampling methods using Markov chains and their applications", *Biometrika*, **57**, 97 (1970), <http://biomet.oxfordjournals.org/cgi/content/abstract/57/1/97>.