

Importance Sampling

We have seen that *uniform sampling* can be used to estimate an integral

$$\int_a^b f(x) \, dx \simeq \frac{b-a}{N} \sum_{i=1}^N f(x_i) \pm \frac{(b-a)\sigma_f}{\sqrt{N}} ,$$

where the random numbers x_i are uniformly distributed in the interval $[a, b]$, and the variance of the function f is

$$\sigma_f^2 = \left(\frac{1}{b-a} \int_a^b f^2(x) \, dx \right) - \left(\frac{1}{b-a} \int_a^b f(x) \, dx \right)^2 \simeq \left(\frac{1}{N} \sum_{i=1}^N f^2(x_i) \right) - \left(\frac{1}{N} \sum_{i=1}^N f(x_i) \right)^2 .$$

Consider however, the following integral:

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} x^2 e^{-x^2/2} \, dx = 1 .$$

There are two problems with using uniform sampling to estimate this integral:

- $(b-a) = \infty$, so a cut-off such as $b = -a = L \gg 1$ must be used, and
- $f(x)$ is exponentially small almost everywhere except near $x = 0$, so the variance of f will be large.

These problems can be avoided by using *importance sampling*. Let $w(x)$ be a *weight function* which is *positive definite* everywhere in the integration interval and normalized to unity:

$$w(x) > 0 \quad \text{for} \quad a < x < b , \quad \int_a^b w(x) \, dx = 1 .$$

The weight function is chosen to reduce the variance of the integrand by means of a change of variable

$$y = \int_a^x w(x') \, dx' , \quad \int_a^b f(x) \, dx = \int_0^1 \frac{f(x(y))}{w(x(y))} \, dy .$$

If $w(x)$ is chosen to be large where $|f(x)|$ is large, and small where $|f(x)|$ is small, then $f(x)/w(x)$ can have a much smaller variance than $f(x)$.

The integral can now be estimated by uniform sampling in y , or equivalently, by sampling in x with probability $w(x)$. For example, to estimate the Gaussian integral example on the previous page, let

$$w(x) = \frac{e^{-x^2/2}}{\sqrt{2\pi}}, \quad \text{so} \quad \frac{f(x)}{w(x)} = x^2.$$

The function `gasdev` in the header file `rng.h` can be used to generate random numbers with probability $w(x)$, as shown in the following program in the file `gauss.cpp`:

```
// estimate Gaussian integral using uniform and importance sampling

#include <cmath>
#include <cstdlib>
#include <iostream>
#include "rng.h"

using namespace std;

const double pi = 4 * atan(1.0);

double f(double x) {
    return x * x * exp(- x * x / 2) / sqrt(2 * pi);
}

double f_over_w(double x) {
    return x * x;
}
```

```
int main() {

    // get input parameters from user
    cout << "Enter number of points N: ";
    int N;
    cin >> N;
    cout << "Enter cut-off L for uniform sampling: ";
    double L;
    cin >> L;
    cout << "Enter integer seed for ran2: ";
    int seed;
    cin >> seed;

    // uniform sampling
    double avg = 0;
    double var = 0;
    for (int i = 0; i < N; i++) {
        double x = (2 * ran2(seed) - 1) * L;
        double fx = f(x);
        avg += fx;
        var += fx * fx;
    }
    avg /= N;
    var /= N;
    var = var - avg * avg;
    cout << "\n    Uniform sampling: " << 2 * L * avg << "    +-    "
        << 2 * L * sqrt(var / N) << endl;
```

```
// importance sampling
avg = var = 0;
for (int i = 0; i < N; i++) {
    double x = gasdev(seed);
    double fx = f_over_w(x);
    avg += fx;
    var += fx * fx;
}
avg /= N;
var /= N;
var = var - avg * avg;
cout << "Importance sampling: " << avg << " +- "
      << sqrt(var / N) << endl;

cout << "          Exact answer: " << 1.0 << endl;
}
```

Run this program and note the following:

- The error estimate for importance sampling agrees with the actual error, and decreases like $1/\sqrt{N}$.
- If the cut-off L for uniform sampling is chose too small, i.e., ~ 1 , then the error estimate is small but does not agree with the actual error which is much larger.
- If the cut-off L for uniform sampling is chosen $\gg 1$, then the error estimate agrees with the actual error, but they are both much bigger than the error in importance sampling.

It is possible to get good results using uniform sampling by choosing the cut-off carefully so that it is not too big or too small. By contrast, importance sampling does not require a cut-off.

The Metropolis Monte Carlo Algorithm

This algorithm was invented by N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller, *J. Chem. Phys.*, **21**, 1087 (1953). It was listed as one of the The Top 10 Algorithms of the 20th Century in a recent issue of *Computing in Science and Engineering*.

Suppose we wish to generate random numbers distributed according to a positive definite function in one dimension, for example

$$P(x) = e^{-x^2/2}.$$

The function need not be normalized for the algorithm to work, and the same algorithm works just as easily in a many dimensional space. The random number sequence $x_i, i = 0, 1, 2, \dots$ is generated by a *random walk* as follows:

- Choose a starting point x_0 .
- Choose a *fixed* maximum step size δ .
- Given an x_i , generate the next random number as follows:
 - Choose x_{trial} uniformly and randomly in the interval $[x_i - \delta, x_i + \delta]$.
 - Compute the ratio

$$w = \frac{P(x_{\text{trial}})}{P(x_i)}.$$

Note that P need not be normalized to compute this ratio.

- If $w > 1$ the trial step is in the *right* direction, i.e., towards a region of higher probability. Accept the step $x_{i+1} = x_{\text{trial}}$.
- If $w < 1$ the trial step is in the *wrong* direction, i.e., towards a region of lower probability. We should not unconditionally reject this step! (Why?) So accept the step *conditionally* if the decrease in probability is smaller than a random amount:
 - Generate a random number r in the interval $[0, 1]$.
 - If $r < w$ accept the trial step $x_{i+1} = x_{\text{trial}}$.

- If $w \leq r$ reject the step $x_{i+1} = x_i$. Note that we don't discard this step! The two steps have the same value.

This algorithm is implemented in the following program `metropolis.cpp` to compute the same Gaussian integral as previously.

```
// estimate Gaussian integral using Metropolis algorithm

#include <cmath>
#include <cstdlib>
#include <iostream>
#include "rng.h"

using namespace std;

const double pi = 4 * atan(1.0);

double P(double x) {
    return exp(- x * x / 2) / sqrt(2 * pi);
}

double f_over_w(double x) {
    return x * x;
}

double x = 0;           // initial position of walker
double delta = 1.0;     // step size
int seed = -123456789;  // seed for ran2
```

```
int accepts = 0;           // number of steps accepted

void MetropolisStep() {
    double xTrial = x + (2 * ran2(seed) - 1) * delta;
    double w = P(xTrial) / P(x);
    if (w >= 1) {           // uphill
        x = xTrial;        // accept the step
        ++accepts;
    } else {               // downhill
        if (ran2(seed) < w) { // but not too far
            x = xTrial;      // accept the step
            ++accepts;
        }
    }
}

int main() {

    // get input parameters from user
    cout << "Enter step size  delta:  ";
    cin >> delta;
    cout << "Enter number of trials:  ";
    int M;
    cin >> M;
    cout << "Enter steps per  trial:  ";
    int N;
    cin >> N;

    double sum = 0;        // accumulator for <f>
```

```
double sqdSum = 0;      // accumulator for  $\langle f \rangle * \langle f \rangle$ 
double errSum = 0;      // accumulator error estimates
for (int i = 0; i < M; i++) {
    double avg = 0;      // accumulator for  $f(x)$ 
    double var = 0;      // accumulator for  $f(x) * f(x)$ 
    for (int j = 0; j < N; j++) {
        MetropolisStep();
        double fx = f_over_w(x);
        avg += fx;
        var += fx * fx;
    }
    avg /= N;
    var /= N;
    var = var - avg * avg;
    double err = sqrt(var / N);
    sum += avg;
    sqdSum += avg * avg;
    errSum += err;
}
double ans = sum / M;
double stdDev = sqrt(sqdSum / M - ans * ans);
stdDev /= sqrt(double(M));
double err = errSum / M;
err /= sqrt(double(M));
cout << "\n Exact  answer:  " << 1.0 << endl;
cout << "      Integral:  " << ans << "  +-  " << err << endl;
cout << "      Std. Dev.:  " << stdDev << endl;
cout << "  Accept ratio:  " << accepts / double(N * M) << endl;
}
```

There are essentially two important choices to be made:

- The initial point x_0 must be chosen carefully. A good choice is close to the maximum of the desired probability distribution. If this maximum is not known (as is usually the case in multi dimensional problems), then the random walker must be allowed to *thermalize* i.e., to find a good starting configuration: the algorithm is run for some large number of steps which are then discarded.
- The step size δ must be carefully chosen.
 - If δ is too small, then most of the trial steps will be accepted, which will tend to give a uniform distribution that converges very slowly to $P(x)$.
 - If δ is too large the random walker will step right over and not “see” important peaks in the probability distribution. If the walker is at a peak, too many steps will be rejected.

A rough criterion for choosing the step size is for the

$$\text{Acceptance ratio} = \frac{\text{Number of steps accepted}}{\text{Total number of trial steps}}$$

to be around 0.5.