Class 0x0A: Monte Carlo integration and simulation

Simple Monte Carlo integration

Suppose we have some function $f(x)$ of the n-dimensional parameter x. Pick N random points x_i, uniformly distributed in volume V. Then

$$I_{\rm MC} \equiv \frac{V}{N}\sum_{i=1}^{N} f(x_i) = V < f >$$

is itself a random number, with mean and standard deviation

$$\overline{I_{\rm MC}} = \int f dV, \qquad \sigma_{[I_{\rm MC}]} = \sqrt{\overline{I_{\rm MC}^2} - \overline{I_{\rm MC}}^2} \approx \frac{V}{\sqrt{N}}\sqrt{< f^2 > - < f >^2}.$$

(Here angle brackets are used to denote the mean over the samples, and overline is used to denote the true mean in the limit of infinite statistics.)

In plain English, I_{MC} is an estimate for the integral.

Integration over many dimensions

- Ordinary numerical integration over n dimensions is not easy due to the number of grid points required, $O(m^n)$ if m is the number of divisions on each axis.
- It often happens that $1/\sqrt{N}$ fractional accuracy from the MC method is better than what you can get from the non-MC methods using N grid points with $m \approx N^{1/n}$ divisions on each axis.

Integration over complicated volumes

If the volume V is hard to sample randomly (strange shape), just define a function that is zero outside the volume and equal to f inside, and integrate that over an easy-to-sample volume.

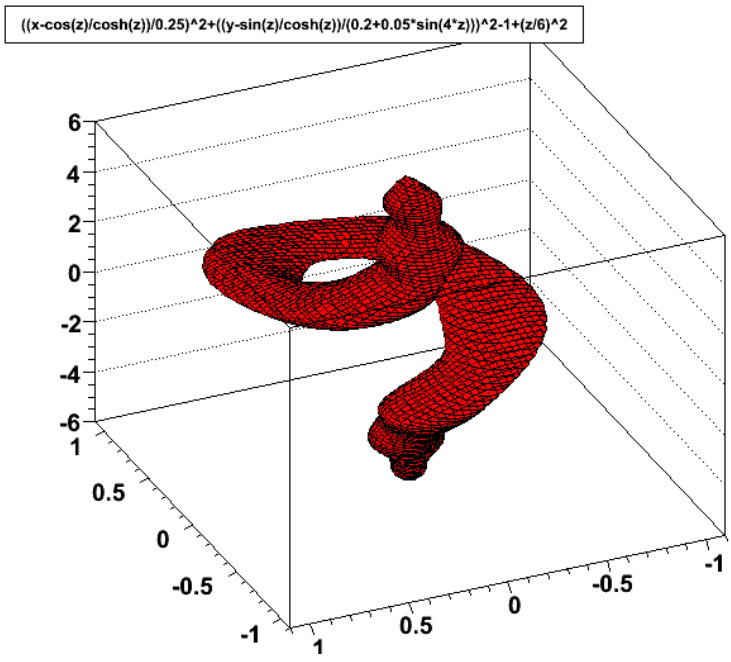


Figure 1: A complicated solid with fine-grained features that would be difficult to integrate analytically or on a numerical grid.

How to get uniform random numbers

Short answer: use a function that returns a (pseudo) random number uniform in the range $[0, 1)$, available from many libraries.

It used to be considered necessary to talk about the pros and cons of various random number generators (RNGs) a great deal, because bad RNGs were commonplace, good ones were hard to find, and computers were so slow it was worth considering using a RNG with poor randomness if it was faster.

The latter two factors are generally no longer true, so a little advice should suffice for the moment.

- Stay away from the standard library function `rand()`. It tends to be one of the poor ones, for historical reasons.
- The Mersenne Twistor RNG is widely available, and is considered a good choice.
- The Marsaglia-Zaman RNG is also widely available, also a good choice.
- ROOT, CLHEP, and GSL all define various good random number generators for C++, including those mentioned above.

See also: GSL manual section General comments on random numbers.

Example: Mass of an ellipsoidal cloud

The interior of an ellipsoid is defined by $(x/a)^2 + (y/b)^2 + (z/c)^2 < 1$. Let the density function be $\rho(x, y, z)$.

Algorithm:

initialize sum_rho1 = 0, sum_rho2 = 0
loop N times:
pick uniform x in $-a < x < a$, uniform y in $-b < y < b$, uniform z in $-c < z < c$
if $(x/a)^2 + (y/b)^2 + (z/c)^2 < 1$:
rho = $\rho(x, y, z)$
sum_rho1 += rho
sum_rho2 += rho * rho
V = 8*a*b*c
mean = sum_rho1/N
mean2 = sum_rho2/N
integral = V*mean
errorest = V*sqrt((mean2 - mean*mean)/N)

Weighting techniques (change of variable)

- If the function being integrated varies strongly on some variable of integration, it helps to redefine the variable to flatten the function.
- This is equivalent to concentrating the distribution of the variable in the more significant region.

Example:

- Let $\rho(x, y, z) = e^{\alpha z} f(x, y)$ in the previous example.
- Define $ds = e^{\alpha z} dz$, resulting in $z = \log(\alpha s)/\alpha$.
- Now integrate $f(x, y)$ over $dxdyds$. The ellipsoidal boundary is still defined by x, y, z, with z calculated from s.

Example: Mass of an ellipsoidal cloud (revisited)

The interior of an ellipsoid is defined by $(x/a)^2 + (y/b)^2 + (z/c)^2 < 1$. Let the density function be $\rho(x, y, z) = f(x, y)e^{\alpha s}$.

Algorithm:

initialize sum_1 = 0, sum_2 = 0

$s_1 = e^{-\alpha c}/\alpha, s_2 = e^{\alpha c}/\alpha$
loop N times:
    pick uniform x in $-a < x < a$, uniform y in $-b < y < b$, uniform s in $s_1 < s < s_2$
    $z = \log(\alpha s)/\alpha$
    if $(x/a)^2 + (y/b)^2 + (z/c)^2 < 1$:
        f = $f(x, y)$
        sum_1 += f
        sum_2 += f * f
V = $4ab(s_2 - s_1)$
mean = sum_1/N
mean2 = sum_2/N
integral = V*mean
errorest = V*sqrt((mean2 - mean*mean)/N)

Simulation of "unknown" probability distributions

Sometimes you know everything about a random process, except it's too hard to evaluate the resulting probability distribution analytically.

> Example: a game of snakes and ladders: you know all the rules, but what is the probability distribution of the number of moves in a game?

One can simulate the process using random numbers and keep track of the distributions of the quantities of interest.

This turns out to be mathematically equivalent to evaluating probability integrals using the Monte Carlo method, where the n dimensions correspond to all the random variables involved. It may be that n is very very large.

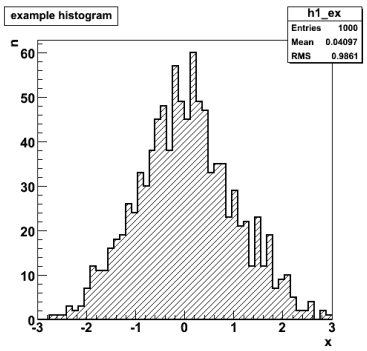
Example: Variability of time to sort card deck

If you have a randomly sorted deck of 52 cards, how many steps does it take to sort them using a particular sorting algorithm? Find mean and standard deviation.

Algorithm to find mean and std. dev. of sorting times:
    initialize `sum1` = 0, `sum2` = 0
    loop N times:
        set up array of integers 1 to 52
        shuffle it
        call sort algorithm (which must tell you the number of steps, nsteps)

    sum1 += nsteps
    sum2 += nsteps*nsteps
mean = sum1/N
mean2 = sum2/N
stddev = sqrt((mean2 - mean*mean)/(N-1))

Storing distributions in histograms



- A histogram is just an array of integers representing a frequency distribution, usually drawn as a bar chart.
- Each integer represents the number of times a particular thing happened.
- For a 1-d histogram of a continuous random number, the "thing that happened" is that the value of the random number fell in a particular range. These ranges are called "bins", and the division of the continuously distributed number into bins is called "binning".

Simple 1-d histogram class

Note: ROOT, GSL, many other packages provide histogram functions, but just to show how easy it is:

```
class SimpleHisto1 {
 private:
  vector<int> bins;   // stores histogram contents
  double xlow;        // lower end of histogram range
  double xhigh;       // upper end of histogram range

 public:

  // set or reset the histogram range and number of bins
  void Initialize(int nbin, double xlow_, double xhigh_) {
```

```
    bins.clear();
    bins.resize(nbin);
    xlow= xlow_;
    xhigh= xhigh_;
  }

  // add 1 to the appropriate bin
  int Fill(double x) {
    if (x<xlow || x>=xhigh)
      return 0;
    int ibin= int( (x-xlow)/(xhigh-xlow)*bins.size() );
    bins[ibin] += 1;
    return 1;
  }

  // get contents of bin i
  int BinContents(int i) {
    return bins[i];
  }
};
```

Example: time to sort card deck (revisited)

If you have a randomly sorted deck of 52 cards, how many steps does it take to sort them using a particular sorting algorithm? Find the full probability distribution.

Algorithm to find distribution of sorting times:

initialize histogram h
loop N times:
set up array of integers 1 to 52
shuffle it
call sort algorithm (which must tell you the number of steps)
h.Fill(nsteps)
print histogram contents

Simulation of known random distributions

So far, we've used a RNG which gives a uniform random number u in the range $[0, 1)$.

Given such a number u, it's easy to get a number x that is uniform in *[a,b)*: $x = a + (b - a)u$.

Likewise, it's easy to get an integer i in the range *0..(n-1)*: $i = \text{int}(u * n)$.

And it's easy to get a boolean b with probability p for true, $1-p$ for false: $b = (u < p)$.

Suppose you need a random number with an exponential probability distribution, or some other distribution?

Inverse distribution function method

Given a uniform random number u in the range $[0, 1)$, we have

$$\frac{dP}{du} = 1, \qquad P_u(u) = u.$$

Here P_u is the integral of the probability density, called the "cumulative distribution function" or simply the "distribution function".* It gives the probability that a given random number will be less than or equal to a given number.

Suppose we want a non-uniform random number x, with properties

$$\frac{dP}{dx} = f(x), \qquad P_x(x) = \int_{-\infty}^{x} f(x')dx'.$$

We can obtain that as follows:

$$x = P_x^{-1}(u).$$

The proof is simple and pleasant, so I leave it as an exercise to the student.

Example: exponential distribution

The distribution function for the exponential distribution is easily inverted:

$$\frac{dP}{dx} = e^{-x}, \quad P_x(x) = 1 - e^{-x}$$

$$P_x^{-1}(u) = -\log(1-u)$$

Thus, $x = -\log(1-u)$ is an exponential random number with the desired probability density. This can be verified by calculating $dP/dx = (dP/du) \cdot (du/dx)$.

Rejection method

Another way to obtain $dP/dx = f(x)$, less efficient, is to generate x uniformly in $x_{\min} < x < x_{\max}$, then generate a second random number p in $[0, 1)$. If

* Also called the "probability distribution function" in some texts, although this terminology is easily confused with the probability density function (p.d.f.), especially since the density function when drawn actually has the shape of the distribution of the probability.

$p < f(x)/f_{max}$, where f_{max} is the largest value f can have, then accept x, otherwise generate a new x and p and try again.

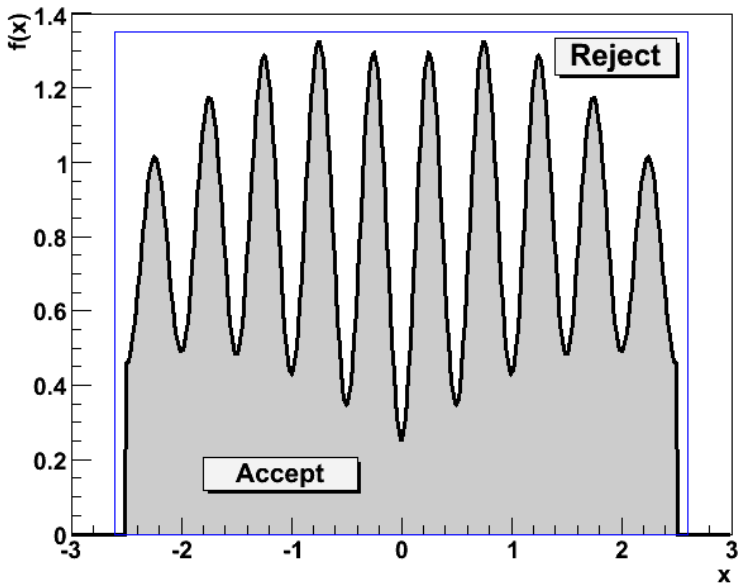


Figure 2: Pairs of (x, p) random variables are rejected if outside the shaded region.

Some pitfalls in using RNGs

- RNGs are really pseudo random numbers. They have a fixed sequence for a given starting seed.
  - Use a different seed if you want different random numbers each time you run your program.
  - Use the same seed if you want the same random numbers when you re-run your program (e.g., for debugging a problem).
- RNGs have some number of bits of precision. Be careful not to ask for more than that.
  - E.g., if you have a lousy 16-bit RNG, don't expect it to find events with probability $< 2^{-16}$ no matter how long you run it.
  - Good RNGs have 48 or more bits, but don't try to "take apart" the random number, like `if (u<p) { u2= u/p; if (u2<p2) { ... } else { ... } } else { ... }`.

Assignment: Random variable resistor network

Suppose I set up 3 variable 1-kohm resistors connected to big knobs in the hallway with a big sign saying "DO NOT ADJUST". Every time a student passes, s/he adjusts the knobs to a random place, uniformly between minimum and maximum. Make the histogram of the total resistor network resistance if one knob is in parallel with two knobs in series, as shown in the figure.

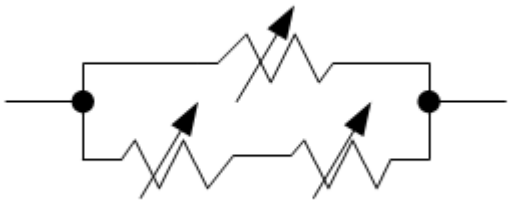


Figure 3: Schematic diagram of the "random knobs".