Laboration: Monte Carlo Integration

Simon Sigurdhsson

October 5, 2010

Abstract

The Monte Carlo method, a general method based on generating a large number of samples to approximate expected values and errors, can be applied when approximating integrals. While this is most effective in higher dimensions, it can also be useful in the one-dimensional case.

1 Approximating π

Calculating π can be done in a number of different ways; one way is to use the integral below, derived from the derivative of the inverse tangent function, $\tan^{-1}$:

$$\int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x = 4\left[\tan^{-1}(x)\right]_{x=0}^{1} = 4\tan^{-1}(1) = 4\frac{\pi}{4} = \pi$$

To obtain an approximative value of π using this integral, we must compute it numerically. There are many methods to compute integrals numerically — some of them approximate the integral with Riemann sums (or variants of them, as done in the trapezoid method), others use polynomial interpolation. These methods can be very computationally expensive if high accuracy is required.

Another method of approximating integrals is using stochastic methods, approximating the integral with a sum of random variables. This method, often referred to as the Monte Carlo integration method, is derived from the expected value of a function g, given that its argument is a random variable from a continuous distribution f:

$$\mathbf{E}\{g(x)\} = \int g(x)f(x)\mathrm{d}x$$

Now, suppose we have a function f that we want to integrate over the interval $[0, 1]$. We have the following integral:

$$\int_0^1 f(x)\,\mathrm{d}x$$

This integral can be interpreted as the expectation $\mathbf{E}\{f(X)\}$ of the random variable $f(X)$, where X is uniformly distributed over $[0, 1]$. Hence, we can approximate the expected value by the sample mean of a number of simulated samples of $f(X)$:

$$\int_0^1 f(x)\,\mathrm{d}x \approx \frac{1}{n}\sum_{i=1}^{n} f(x_i)$$

Our computationally expensive integration has now been reduced to generating a sample of n uniformly distributed variables and calculating the sample mean, which is comparatively easy.

By the law of large numbers, our sample mean must converge toward the true value of the integral as $n \to \infty$. In real-world applications, generating an infinite number of random variables is of course impossible. Thus, it is of interest to be able to calculate the expected error of our estimation (or indeed the *variance*).

After some fiddling around with the Central Limit Theorem, and the variance of normally distributed random variables, we arrive at the conclusion that the error can be estimated by σ (here, E is the approximated value given by the sum above):

$$\sigma^2(f) = \frac{1}{n}\sum_{i=1}^{n} f(x_i)^2 - E^2 \sim n^{-1}$$

The advantage of Monte Carlo integration is both its relatively inexpensive computation costs, and its relatively constant error: the error when using the Monte Carlo method is approximately $O(n^{-1/2})$, where n is the number of samples.

1.1 A MATLAB simulation

Now, using the Monte Carlo method to actually approximate π will be child's play — in MATLAB, we can do it in just a few lines (along with an error estimation):

```
n = 10000;
x_i = rand(1, n);
fx_i = 4./(1+x_i.^2);
approx_pi = sum(fx_i)/n
error = sqrt((sum(fx_i.^2)/n-approx_pi^2)/n)
```

This yields the fairly accurate result $\pi = 3.14 \pm 0.0064$ — the estimated error of 0.0064 is fairly large compared to the actual error of 0.0016. For higher accuracy, we can increase n; performing the same computation with $n = 10^8$ yields a more accurate result of $\pi = 3.1417 \pm 6 \cdot 10^{-5}$ (rounded). Compared to the real error of $8 \cdot 10^{-5}$, this is fairly spot-on. Hence, we can conclude that the error estimate is *very* accurate.

1.2 Reducing variance

To reduce the variance (and hence improve the accuracy of our approximation), several methods may be applied; among those are *stratified sampling*, *importance sampling*, *control variates* and *antithetic variates*. Theoretically, some of these could be combined to produce even better results.

1.2.1 Stratified sampling

Stratified sampling is in effect a division of the integral; integrating over several different intervals and summing these to obtain the actual value of the original integral. This way, we can concentrate on more difficult areas of the integral and approximate these more closely. In our case, we may divide the interval $[0, 1]$ into smaller sub-intervals $M_1, \ldots, M_k$. For each of these smaller intervals M_j, we generate n_j random variables with a uniform distribution over that interval. We can then calculate the expected value as

$$\sum_{j=1}^{k} \frac{\mathrm{vol}(M_j)}{n_j} \sum_{i=1}^{n_j} f(x_{ij}),$$

where $\mathrm{vol}(M_j)$ is the *volume* of the interval (or set) M_j. The error estimation is somewhat more involved:

$$\sigma = \sqrt{\sum_{j=1}^{k} \frac{\mathrm{vol}(M_j)^2}{n_j} \sigma_{M_j}^2(f)}$$

Here, $\sigma_{M_j}^2(f)$ is the approximation error on the sub-interval M_j, given by

$$\sigma_{M_j}^2(f) = \left(\frac{1}{\mathrm{vol}(M_j)} \int_{M_j} f(x)^2 \, \mathrm{d}x - \left(\frac{1}{\mathrm{vol}(M_j)} \int_{M_j} f(x) \, \mathrm{d}x \right)^2 \right).$$

Stratified sampling performs best when n_j is selected so that it is proportional to the total error on the corresponding interval, given by $\mathrm{vol}(M_j)\sigma_{M_j}(f)$. Thus, if we run our simulation once with $n_j = 1$, $\forall j$, we can calculate more suitable values for n_j.

All this may look terribly involved, but even with arbitrarily chosen intervals and varying values, the MATLAB code is fairly simple:

```
n_j = [3.9 3.9 3.8 3.8 3.7 3.6 3.4 2.9 2.4 2.9] * 10000;
volM_j = [0.05 0.05 0.05 0.05 0.05 0.05 0.1 0.2 0.2 0.2];
x_ij = rand(length(n_j), max(n_j)).*repmat(volM_j, max(n_j), 1)' ...
       + repmat(cumsum([0 volM_j(1:(end-1))]), max(n_j), 1)';
for i=1:length(n_j)
  x_ij(i, n_j(i):end) = -Inf;
end
fx_ij = 4./(1+x_ij.^2).*(x_ij ~= Inf);
sigma2_Mj = 1./volM_j.*(1./n_j.*(sum(fx_ij'.^2))) ...
            - 1./volM_j.*(1./n_j.*(sum(fx_ij'))).^2;
approx_pi = sum(volM_j./n_j.*sum(fx_ij'));
error = sqrt(sum(volM_j.^2./n_j.*sigma2_Mj));
good_nj = volM_j.*sqrt(sigma2_Mj);
```

Emphasis here is put on the *first* half of the interval. The values for n_j have been obtained by first doing a run with $n_j = 1$, and simply looking at the `good_nj` variable. Note that we pad the matrix `x_ij` with $-\infty$ where the matrix is wider than the number

of random samples; this does not affect the final result (since we filter out those values from `fx_ij`).

Using this method, an approximation of $\pi = 3.1412 \pm 0.0006$ was obtained. A few similar interval divisions repeatedly return an error estimate around 0.0005. This is clearly an improvement from before, and given that the actual error (0.0004) is much smaller than that of the original approximation (0.0016) this method is something you should consider. It is fairly easy to implement in MATLAB as well (although gets more complex in a low-level language such as C).

1.2.2 Importance sampling

The next method, importance sampling, relies on the fact that

$$\int f(d)\ \mathrm{d}x = \int \frac{f(x)}{p(x)} p(x)\ \mathrm{d}x.$$

Hence, if $p(x)$ is a probability density function and our random samples x_i are generated from the corresponding distribution, we can approximate the integral using the following, modified mean:

$$\frac{1}{n} \sum_{i=1}^{n} \frac{f(x_i)}{p(x_i)}$$

The optimal selection of $p(x)$ is a distribution as close to f as possible — if $p \equiv f$, this sum will be equal to 1, and thus the error (again estimated by σ) will sum to zero. This is to be expected, since we are in fact calculating the expected value of the (non-)random variable 1, given a probability distribution function $p(x)$. Since 1 isn't a random variable, it's expectation value will be 1 and its variance will be 0.

Given our function, the Cauchy$(0, 1)$ distribution (basically only differing from our function by a factor 4π) would be a good candidate. However, since its support is $(-\infty, \infty)$, we'd in effect be integrating on that interval. We need a probability distribution with support over $[0, 1]$. The only other requirement of this probability distribution is that

$$\int_0^1 p(x)\ \mathrm{d}x = 1$$

Given our function (seen as the blue curve in Figure 1) and the form of our function f, the Cauchy$(0, 1)$ distribution (seen as the red curve in Figure 1) seem like a good choice — but its support is (as said above) the whole real line. We do however note that

$$\int_0^1 \frac{1}{\pi(1 + x^2)} \mathrm{d}x = \frac{1}{4},$$

which implies that we can *construct* a probability distribution by multiplying the Cauchy distribution. We will then have a probability distribution with support on $[0, 1]$:

$$p(x) = \frac{4}{\pi(1 + x^2)}$$

This is pretty similar to our function f. In fact, the two differ only by a factor of π (as expected). This is further illustrated by the green curve in Figure 1, which is the Cauchy(nction f.

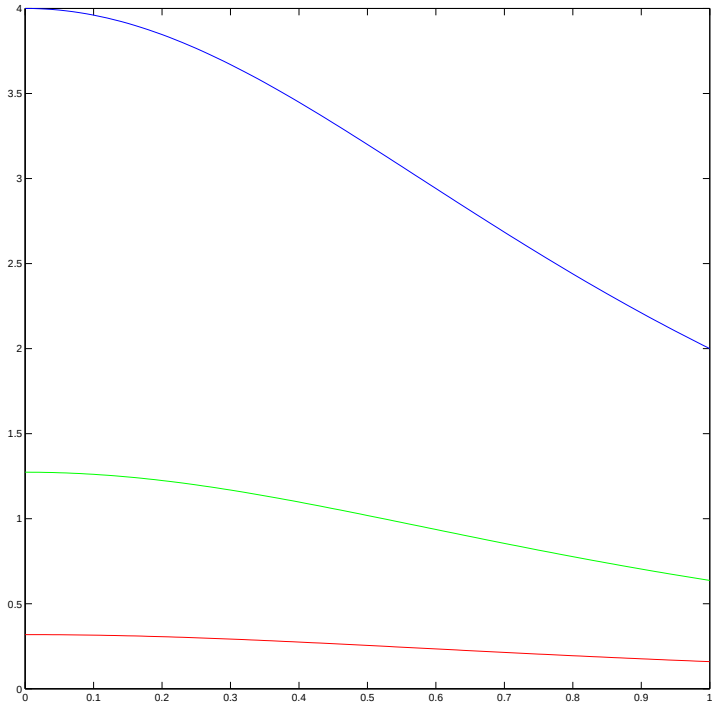

Figure 1: The integrand (blue), Cauchy pdf (red), and the m-Cauchy pdf (green).

Let us then define a probability distribution, based on $p(x)$ given above, and let's call it the *m-Cauchy* distibution:

$$C_m(x) = \begin{cases} \frac{4}{\pi(1+x^2)}, & 0 \leq x \leq 1 \\ 0, & \text{otherwise} \end{cases}$$

Our concern now is generating random samples with the m-Cauchy distribution. There is (understandably) no standard function to do this in MATLAB, so we have to resort to a simulation of random variables — we will have to feed uniformly distributed samples to the generalized right-inverse $F^{\leftarrow}$. The m-Cauchy distribution has a well-defined inverse (easily calculated analytically), so we'll use it:

$$F^{-1}(x) = \tan\left(\frac{\pi}{4}\left(x - \frac{1}{2}\right)\right)$$

Of course, we have to change our sample mean to match our new sum, given by $f(x)/p(x)$. We'll see that it reduces quite nicely to π:

$$\frac{1}{n}\sum_{i=1}^{n} \frac{\frac{4}{1+x^2}}{\frac{4}{\pi(1+x^2)}} = \frac{1}{n}\sum_{i=1}^{n} \frac{\pi(1+x^2)}{1+x^2} = \frac{1}{n}\sum_{i=1}^{n} \pi = \pi$$

In this case we've obviously reached a dead-end: we now need to use the actual value of π to calculate the value of π, which of course is quite bad. However, we've

reached a point where the variance is exactly zero for any r.v. input. There are of course distributions for which this method, along with this integral, doesn't end with a constant value — but those are far worse fiting and may very well result in a *larger* error.

1.2.3 Control variates

Introducing a function g similar to f, with a known integral value G, we can rewrite our initial integral as follows:

$$\int f(x)\,\mathrm{d}x = \int (f(x) - g(x))\,\mathrm{d}x + \int g(x)\mathrm{d}x = \int (f(x) - g(x))\,\mathrm{d}x + G$$

The idea is that if we have a smaller integral to approximate, we will have a smaller error. The Monte Carlo approximation can now be written as:

$$\frac{1}{n}\sum_{i=1}^{n}(f(x_i) - g(x_i)) + G$$

The error is still estimated as in the unmodified Monte Carlo method, with the exception that G is not included anywhere in the calculations. Our first task when using control variates is thus to find a function g that is both close to f and has a known value. In our case, we should also make sure that it does not contain π itself. We find that the following function may be a good candidate:

$$g(x) = \frac{4}{(1+x)^2)},\ \ G = 2$$

Based on this, and the MATLAB code from our first attempt, we can construct a Monte Carlo integration with control variates like this:

```
n = 10000; G = 2;
x_i = rand(1, n);
fx_i = 4./(1+x_i.^2)-4./((1+x_i).^2);
approx_pi = sum(fx_i)/n + G
error = sqrt((sum(fx_i.^2)/n-(approx_pi-G)^2)/n)
```

As expected, this variant produces much better results than the approximation before control variates; our approximation is now $\pi = 3.1415 \pm 0.0032$, with an actual error of only 0.0001! Increasing n to 10^8 as before produces even better results; the estimated error and the real error are both now approximately $3 \cdot 10^{-5}$ — which is very good.

1.2.4 Antithetic variates

Antithetic variates rely on the fact that the variance of the sum of two negatively correlated random variables should be small, since the covariance effectively reduces the variance sum:

$$\mathbf{Var}\{f_1 + f_2\} = \mathbf{Var}\{f_1\} + \mathbf{Var}\{f_2\} + 2\mathbf{Cov}\{f_1, f_2\}$$

Hence, if we use the random variables x_i and $1 - x_i$, which are both random and negatively correlated, we can instead approximate our integral with the mean of the sample mean of these two correlated variables:

$$\frac{1}{2n}\left(\sum_{i=1}^{n} f(x_i) + \sum_{i=1}^{n} f(1 - x_i)\right)$$

The error is calculated as before. This translates very well to our MATLAB code:

```
n = 10000;
x_i = rand(1, n);
fx_i = 4./(1+x_i.^2);
fx_i2 = 4./(1+(1-x_i).^2);
approx_pi = (sum(fx_i)+sum(fx_i2))/(2*n)
error = sqrt(((sum(fx_i.^2)+sum(fx_i2.^2))/(2*n)-approx_pi^2)/(2*n))
```

Unfortunately, this does not improve the result significantly; the approximation using this method is $\pi = 3.1416 \pm 0.0045$, with a real error of $6 \cdot 10^{-6}$. Compared to the original result, the actual error has decreased, while the estimated error has increased.

A disadvantage of this method is that it can only be used on *monotone* functions f. This makes it fairly useless in a lot of cases.

1.3 Better performance using C

Using C, a lower-level language with no matrix manipulation capabilities, we're forced to use a classic for-loop. The relevant parts of the C code are as follows (note that it has to be compiled in C99 mode with GNU extenstions, i.e. `-std=gnu99`):

```
int n = pow(10, 8);
double approx_pi, error_term, error, x, y;
double exact_pi = 3.14159265;

srand48(time(NULL));
for (int i = 0; i <= n; i++) {
  x = drand48();
  y = 4 / (1 + x * x);
  approx_pi += y;
  error_term += y * y;
}
approx_pi = approx_pi / n;
error = sqrt((error_term / n - pow(approx_pi, 2)) / n);
```

Those versed in both C and MATLAB will see that this is basically the same thing, except it has been converted into a for-loop. This code (expectedly) yields pretty much the same results as the MATLAB version: an approximation of $\pi = 3.141569 \pm 0.000064$; the real error is now 0.000034. This is of course without any variance-reducing methods.

One should comment on the efficiency of these solutions; the MATLAB solution (which is pretty much the fastest you'll get in MATLAB, given that it uses matrix manipulation exclusively) runs in around 3.5 seconds. The C version however runs in roughly 2.2 seconds — a significant improvement of almost 40%! This showcases the strength of low-level languages: they almost always perform better.

2 A quite irregular function

One strength of Monte Carlo integration is that it isn't too bothered by very irregular functions. Take for example the integral of the (almost) Brownian motion:

$$\int_0^1 \left(\sum_{k=0}^{n} \frac{\sqrt{8}}{\pi} \frac{\sin(\frac{1}{2}(2k+1)\pi t)}{2k+1} n_k \right) \, \mathrm{d}t$$

Where $\{n_k\}_{k=1}^n$ is independent and $N(0,1)$-distributed. When $n \to \infty$, this is Brownian motion; known to be continuous but not differentiable in any single point on the interval $[0,1]$.

Translating this to C to approximate it using Monte Carlo integration is not difficult; we begin by generating a suitable number of normally distributed values n_k, since these should be constant during integration:

```
double nk[N];

srand48(time(NULL));
for (int i = 0; i < N; i++)
  nk[i] = sqrt(-2*log(drand48()))*cos(2*M_PI*drand48());
```

The normally distributed variable is generated from two uniformly distributed variables η and ξ using the relation $Z = \sqrt{-2\ln(\eta)}\cos(2\pi\xi)$. After this, we simply perform the Monte carlo approximation (with 10^8 samples):

```
int n = pow(10, 5);
double approx_B, error_term, error, x, y;

for (int i = 0; i < n; i++) {
    x = drand48();
    y = 0;
    for (int k = 0; k < N; k++) {
        y = y + sin((M_PI * x)*(2*k + 1)/2);
y = y * nk[k] / (2*k + 1);
    }
    y = y * sqrt(8)/M_PI;
    approx_B += y;
    error_term += y * y;
}
approx_B = approx_B / n;
error = sqrt((error_term / n - pow(approx_B, 2)) / n);
```

With $N = 1000$, the approximated value is -0.6997. The error is 347, which is *very* large. Applying one of the variance-reducing methods is clearly of interest — but which one?

We can't apply anithetic variates, since the function isn't monotone, and we don't want to apply importance sampling, since it's *very* hard to find a distribution close to our function. Stratified sampling won't improve the situation significantly, since the function looks pretty much the same over the whole interval (i.e. there is no sub-interval

that would benefit from a smalle interval). Left now is *control variates*. Using those would require us to find a function g that is both close to f and has a known integral value. This is also very difficult, although we recognize the integral as (at least being similar to) a fourier series with stochastic coefficients n_k.

The large error is to be expected of such an irregular and stochastic function.

3 Expected shortfalls

The expected shortfall $\mathbf{E}\{S_X(u)\}$, given by $\mathbf{E}\{S_X(u)\} = \mathbf{E}\{X|X > u\}$, can be approximated using the Monte Carlo method. In the case given, where we have $X = YZ$ and $Y = \text{Bernoulli}(p)$, $Z = \exp(\lambda)$ we can simply generate a large number of samples from Y and Z, multiply these and calculate the sample mean. To include the constraint $X > u$, we simply discard all samples less than u. So, if we have n samples, and m of those satisfy $x_i > u$, (m), we can approximate the expected shortfall:

$$\mathbf{E}\{S_X(u)\} \approx \frac{1}{m}\sum_{i=0}^{m} f(x_i), \quad f(x_i) = \begin{cases} x_i, & x_i > u \\ 0, & x_i \leq u \end{cases}$$

The error is approximated in a similar way by calculating the square root of the sample variance:

$$\sigma = \sqrt{\mathbf{V}\{S_X(u)\}} = \sqrt{\mathbf{E}\{X^2\} - \mathbf{E}\{X\}^2} \approx \sqrt{\frac{1}{m}\sum_{i=0}^{m} f(x_i)^2 - \left(\frac{1}{m}\sum_{i=0}^{m} f(x_i)\right)^2}$$

The issue now is generating the random samples y_i and z_i. Fortunately, the exponential distribution has a well-defined inverse, and the Bernoulli distribution has a simple generalized right-inverse. We can thus sample y_i and z_i like this, assuming ξ is a random variable with uniform distribution over $[0, 1]$:

$$y_i(\xi) = \begin{cases} 0, & \xi < p \\ 1, & \xi \geq p \end{cases}$$

$$z_i(\xi) = \frac{-\ln(1-\xi)}{\lambda}$$

Keep in mind that these should be independently generated. Thus, if we want to generate x_i directly we must use two random variables ξ and η, both uniformly distributed over $[0, 1]$, before finally calculating the sample $x_i = y_i(\xi)z_i(\eta)$. Doing all this in C is fairly straight-forward:

```
int n = pow(10, 8);
int m = 0;
double shortfall, error, error_term, x;
double p = 0.1;
double invlambda = 3.4;
double u = 10;

srand48(time(NULL));
```

```
for (int i = 0; i < n; i++) {
    x = -log(1-drand48())*invlambda;
    if(drand48() >= p && x > u) {
        shortfall += x;
error_term += x * x;
m++;
    }
}
shortfall = shortfall / m;
error = sqrt((error_term / m - pow(shortfall, 2)) / m);
```

Here, we are using the given values $p = 0.1$, $1/\lambda = 3.4$ and $u = 10$. Running this yields an approximated shortfall of 13.397956 ± 0.001558, which is (if the error estimate is to be trusted) a very good approximation. Variance reduction methods could be applied if a more accurate result is required; antithetic variates would possibly be a good candidate here, but given the small error, this is not necessary.