Monte Carlo integration

Introduction

Monte Carlo integration is a quadrature (cubature) where the nodes are chosen randomly. Typically no assumptions are made about the smoothness of the integrand, not even that it is continuous.

Monte Carlo algorithms are particularly suited for multi-dimensional integrations where one of the problems is that the integration region, Ω, often has quite complicated boundary which can not be easily described by simple functions. However, it is usually much easier to find out whether a given point lies within the integration region or not. Therefore a popular strategy is to create an auxiliary rectangular volume, V, which encompasses the integration volume Ω, and an auxiliary function which coincides with the integrand inside the volume Ω and is equal zero outside. Then the integral of the auxiliary function over the auxiliary volume is equal the original integral.

However, the auxiliary function is generally non-continuous at the boundary; thus ordinary quadratures which assume continuity of the integrand are bound to have difficulties here. One the contrary the Monte-Carlo quadratures will do just as good (or as bad) as with continuous integrands.

A typical implementation of a Monte Carlo algorithm integrates the given function over a rectangular volume, specified by the coordinates of its "lower-left" and "upper-right" vertices, assuming the user has provided the encompassing volume with the auxiliary function.

Plain Monte Carlo algorithm distributes points uniformly throughout the integration region using uncorrelated pseudo-random sequences of points.

Adaptive algorithms, such as VEGAS and MISER, distribute points non-uniformly in an attempt to reduce integration error using correspondingly *importance* and *stratified* sampling.

Yet another strategy to reduce the error is to use correlated quasi-random sequences.

The GNU Scientific Library, GSL, implements a plain Monte Carlo integration algorithm; a stratified sampling algorithm, MISER; an importance sampling algorithm, VEGAS; and a number of quasi-random generators.

Plain Monte Carlo sampling

Plain Monte Carlo is a quadrature with random abscissas and equal weights,

$$\int_V f(\mathbf{x})dV \approx w\sum_{i=1}^{N} f(\mathbf{x}_i)\ , \tag{1}$$

where $\mathbf{x}$ is a point in the multi-dimensional integration space. One free parameter, w, allows one condition to be satisfied: the quadrature must integrate exactly a constant function. This gives $w = V/N$,

$$\int_V f(\mathbf{x})dV \approx \frac{V}{N}\sum_{i=1}^{N} f(\mathbf{x}_i) \doteq V\langle f\rangle\ . \tag{2}$$

Under the assumptions of the *central limit theorem* the error of the integration can be estimated as

$$\text{error} = V\frac{\sigma}{\sqrt{N}}\ , \tag{3}$$

where σ is the variance of the sample,

$$\sigma^2 = \langle f^2\rangle - \langle f\rangle^2\ . \tag{4}$$

The familiar $1/\sqrt{N}$ convergence of a random walk process is quite slow: to reduce the error by a factor 10 requires 100-fold increase in the number of sample points.

Expression (3) provides only a statistical estimate of the error, which is not a strict bound; random sampling may not uncover all the important features of the function, resulting in an underestimate of the error.

A simple implementation of the plain Monte Carlo algorithm is shown in Table 1.

Table 1: Plain Monte Carlo integrator

```
#include <math.h>
#include <stdlib.h>
#define RND ((double)rand()/RAND_MAX)
void randomx(int dim, double *a, double *b, double *x)
{ for(int i=0;i<dim;i++) x[i]=a[i]+RND*(b[i]-a[i]); }

void plainmc(int dim, double *a, double *b,
double f(double* x),int N, double*result, double*error)
{ double V=1; for(int i=0;i<dim;i++) V*=b[i]-a[i];
  double sum=0, sum2=0, fx, x[dim];
  for(int i=0;i<N;i++){ randomx(dim,a,b,x); fx=f(x);
                        sum+=fx; sum2+=fx*fx; }
  double avr = sum/N, var = sum2/N-avr*avr;
  *result = avr*V; *error = sqrt(var/N)*V;
}
```

Importance sampling

Suppose the points are distributed not uniformly but with some density $\rho(\mathbf{x})$. That is, the number of points Δn in the volume ΔV around point $\mathbf{x}$ is given as

$$\Delta n = \frac{N}{V}\rho(\mathbf{x})\Delta V, \tag{5}$$

where ρ is normalised such that $\int_V \rho dV = V$.

The estimate of the integral is then given as

$$\int_V f(\mathbf{x})dV \approx \sum_{i=1}^{N} f(\mathbf{x}_i)\Delta V_i = \sum_{i=1}^{N} f(\mathbf{x}_i)\frac{V}{N\rho(\mathbf{x}_i)} = V\left\langle \frac{f}{\rho}\right\rangle , \tag{6}$$

where

$$\Delta V_i = \frac{V}{N\rho(x_i)} \tag{7}$$

is the volume-per-point at the point x_i.

The corresponding variance is now given by

$$\sigma^2 = \left\langle \left(\frac{f}{\rho}\right)^2 \right\rangle - \left\langle \frac{f}{\rho}\right\rangle^2 . \tag{8}$$

Apparently if the ratio f/ρ is close to a constant, the variance is reduced.

It is tempting to take $\rho = |f|$ and sample directly from the integrand. However in practice evaluations of the integrand are typically expensive. Therefore a better strategy is to build an approximate density in the product form, $\rho(x, y, \dots, z) = \rho_x(x)\rho_y(y)\dots\rho_z(z)$, and then sample from this approximate density. A popular routine of this sort is called VEGAS.

Stratified sampling

Stratified sampling is a generalisation of the recursive adaptive integration algorithm to random quadratures in multi-dimensional spaces.

The ordinary "dividing by two" strategy does not work for multi-dimensional integrations as the number of sub-volumes grows way too fast to keep track of. Instead one estimates along which dimension a subdivision should bring the most dividends and only subdivides along this dimension. Such strategy is called *recursive stratified sampling.* A simple variant of this algorithm is presented in Table 2.

In a stratified sample the points are concentrated in the regions where the variance of the function is largest, see an illustration in Figure 1.

Table 2: Recursive stratified sampling algorithm

```
sample $N$ random points with plain Monte Carlo;
estimate the average and the error;
IF the error is acceptable :
  RETURN the average and the error;
ELSE :
  FOR EACH dimension :
    subdivide the volume in two along the dimension;
    estimate the sub-variances in the two sub-volumes;
  pick the dimension with the largest sub-variance;
  subdivide the volume in two along this dimension;
  dispatch two recursive calls to each of the sub-volumes;
  estimate the grand average and grand error;
  RETURN the grand average and grand error;
```

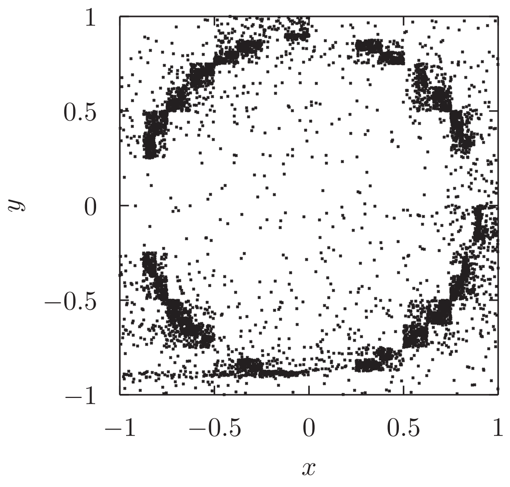

Figure 1: Stratified sample of a discontinuous function
$$f(x,y) = \begin{cases} 1 \text{ if } x^2+y^2<0.9^2 \\ 0 \text{ otherwise} \end{cases}$$
built with the algorithm in Table 2.

Quasi-random (low-discrepancy) sampling

Pseudo-random sampling has high discrepancy[1]: it typically creates regions with hight density of points and other regions with low density of points, see an illustration on Figure 2 (left). With pseudo-random sampling there is a finite probability that all the N points would fall into one half of the region and none into the other half.

Quasi-random sequences avoid this phenomenon by distributing points in a highly correlated manner with a specific requirement of low discrepancy, see Figure 2 for an example. Quasi-random sampling is like a computation on a grid where the grid constant must not be known in advance as the grid is ever gradually refined and the points are always distributed uniformly over the region. The computation can be stopped at any time.

By placing points more evenly than at random, the quasi-random sequences try to improve on the $1/\sqrt{N}$ convergence rate of pseudo-random sampling.

The central limit theorem does not apply in this case as the points are not statistically independent. Therefore the variance can not be used as an estimate of the error. The error estimation is actually not trivial. In practice one can employ two different sequences and use the difference in the resulting integrals as an error estimate.

[1]discrepancy is a measure of how unevenly the points are distributed over the region.

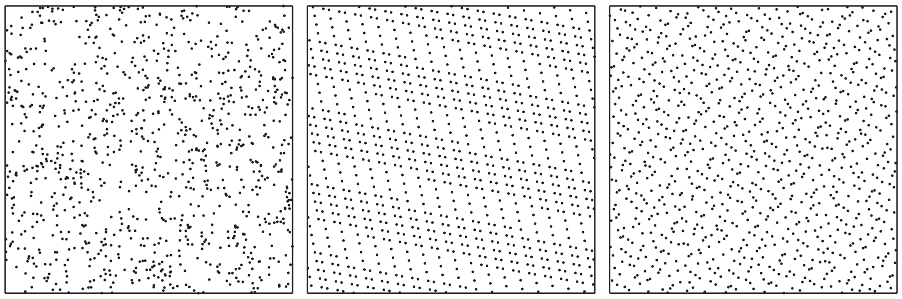

Figure 2: Typical distributions of pseudo-random points (left), and quasi-random low-discrepancy points: lattice (center) and base-2/3 Halton (right) sequences. The first thousand points are plotted in each case.

Van der Corput and Halton sequences

A *van der Corput sequence* is a low-discrepancy sequence over the unit interval. It is constructed by reversing the base-b representation of the sequence of natural numbers $(1, 2, 3, \dots)$. For example, the decimal van der Corput sequence begins as

$$0.1, 0.2, 0.3, \dots, 0.8, 0.9, 0.01, 0.11, 0.21, 0.31, \dots, 0.91, 0.02, 0.12, \dots . \tag{9}$$

In a base-b representation a natural number n with s digits $\{d_i \mid i = 1 \dots s,\ 0 \le d_i < b\}$ is given as

$$n = \sum_{k=1}^{s} d_k b^{k-1} . \tag{10}$$

The corresponding base-b van der Corput number $q_b(n)$ is then given as

$$q_b(n) = \sum_{k=1}^{s} d_k b^{-k} . \tag{11}$$

Here is a C implementation of this algorithm,

```c
double corput(int n, int base){
  double q=0, bk=(double)1/base;
  while(n>0){ q += (n % base)*bk; n /= base; bk /= base; }
  return q; }
```

The van der Corput numbers of any base are uniformly distributed over the unit interval. They also form a dense set in the unit interval: there exists a subsequence of the van der Corput sequence which converges to any given real number in $[0, 1]$.

The Halton sequence is a generalization of the van der Corput sequence to d-dimensional spaces. One chooses a set of coprime bases $b_1, \dots, b_d$ and then for each dimension i generates a van der Corput sequence with its own base b_i. The n-th Halton d-dimentional point $\mathbf{x}$ in the unit volume is then given as

$$\mathbf{x}_{b_1,\dots,b_d}(n) = \{q_{b_1}(n), \dots, q_{b_d}(n)\} . \tag{12}$$

Here is a C implementation which calls the `corput` function listed above,

```c
#include<assert.h>
void halton(int n, int d, double *x){
  int base[]={2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67};
  int maxd=19; assert(d <= maxd);
  for(int i=0;i<d;i++) x[i]=corput(n,base[i]); }
```

Lattice rules

In the simplest incarnation a lattice rule can be defined as follows. Let the *generating vector* $\mathbf{z} = \{\alpha_i \mid i = 1, \dots, d\}$—where d is the dimension of the integration space—be a set of cleverly chosen irrational numbers. Then the n-th point (in the unit volume) of the sequence is given as

$$\mathbf{x}(n) = \mathrm{frac}(n\mathbf{z}) \equiv \{\mathrm{frac}(n\alpha_1), \dots, \mathrm{frac}(n\alpha_d)\} , \tag{13}$$

where frac(x) is the fractional part of x.

An implementation of this algorithm in C is given in Table 3 and an illustration of such sequence is shown on Figure 2 (center).

Table 3: Lattice low-disrepancy quasi-random sequence in C.

```
#include <math.h>
#include <assert.h>
#include <gc.h> // garbage collector
#define fracl(x) ((x)-floorl(x))
#define real long double
void lattice(int d, double *x){
  static int dim=0, n=-1; static real *alpha; int i;
  if(d<0){ // d<0 is the signal to (re-)initialize the lattice
    dim=-d; n=0; alpha=(real*)GC_MALLOC(dim*sizeof(real));
    for(i=0;i<dim;i++) alpha[i]=fracl(sqrtl((real)13*(i+1))); }
  else{
    assert(d==dim && n>=0);
    n++;for(i=0;i<dim;i++)x[i]=fracl(n*alpha[i]); }
}
```

Implementations

Table 4: Javascript implementation of the stratified sampling algorithm.

```
Array.prototype.__iterator__=function(){
  for(var i=0;i<this.length;i++) yield i;}

function strata(f,a,b,acc,eps,N,oldstat,V)
{
var randomx=function(a,b)[a[i]+Math.random()*(b[i]-a[i])for(i in a)]
var range  =function(n) {for(var i=0;i<n;i++) yield i}
var stats  =function(xs){
  var n=xs.length;
  var average=xs.reduce(function(a,b)a+b,0)/n;
  var variance=xs.reduce(function(a,b)a+b*b,0)/n-average*average;
  return [average,variance,n];
   }
if(typeof(N)=="undefined")N=42;
if(typeof(oldstat)=="undefined"){// first call: setting up old stats
  var V=1; for(let k in a) V*=(b[k]-a[k]);
  var xs=[randomx(a,b) for(i in range(N))];
  var ys=[f(xs[i]) for (i in xs)];
  var oldstat=stats(ys);
  }
var xs=[randomx(a,b) for(i in range(N))] // new points
var ys=[f(xs[i]) for(i in xs)]            // new function values
var [average,variance,]=stats(ys)  // average and variance
var [oaverage,ovariance,oN]=oldstat
var integ=V*(average*N+oaverage*oN)/(N+oN)    // integral and error
var error=V*Math.sqrt( (variance*N+ovariance*oN)/(N+oN)/(N+oN) )
if(error<acc+eps*Math.abs(integ)) return [integ,error]; // done
else{ // not done: need to dispatch a recursive call
  var vmax=-1, kmax=0
  for(let k in a){ // look in all dimensions with is best to bisect
    var [al,vl,nl]=
      stats([ys[i] for(i in xs)if(xs[i][k]< (a[k]+b[k])/2)])
    var [ar,vr,nr]=
      stats([ys[i] for(i in xs)if(xs[i][k]>=(a[k]+b[k])/2)])
    var v=Math.abs(al-ar) // take the one with largest variation
    if(v>vmax){ // remember the values
      vmax=v;kmax=k;
      var oldstatl=[al,vl,nl], oldstatr=[ar,vr,nr]
      }
  } // now dispatch two recursive calls
  var a2=a.slice(); a2[kmax]=(a[kmax]+b[kmax])/2
  var b2=b.slice(); b2[kmax]=(a[kmax]+b[kmax])/2
  var [i1,e1]=strata(f,a,b2,acc/1.414,eps,N,oldstatl,V/2)
  var [i2,e2]=strata(f,a2,b,acc/1.414,eps,N,oldstatr,V/2)
  return [i1+i2,Math.sqrt(e1*e1+e2*e2)] // return results
  }
}
```