1 Monte Carlo integration

1.1 Introduction

Monte Carlo integration is a quadrature (cubature) where the nodes are chosen randomly [**?**]. Typically no assumption is 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, might have a quite complicated boundary which can not be easily described by simple functions. On the other hand, 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—that 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.

1.2 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

```
def plainmc(f, a, b, N):
    dim=len(a)
    V=1; for i in range(dim): V*=a[i]-b[i]
    sum=0; sum2=0
    for i in range(N):
        x=[random.uniform(a[i],b[i]) for i in range(dim)]
        fx=f(x)
        sum+=fx; sum2+=fx*fx
    mean = sum/N, variance2 = sum2/N-mean*mean
    return (mean*V, sqrt(variance2/N)*V;
```

1.3   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, $\rho(x, y, \ldots, z) = \rho_x(x)\rho_y(y)\ldots\rho_z(z)$, and then sample from this approximate density. A popular routine of this sort is called VEGAS.

1.4 Stratified sampling

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

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;
```

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 the illustration in Figure 1.

The naive implementation in Table 2 keeps throwing points until the given tolerance is achieved. This might be problematic for multidimensional integrals
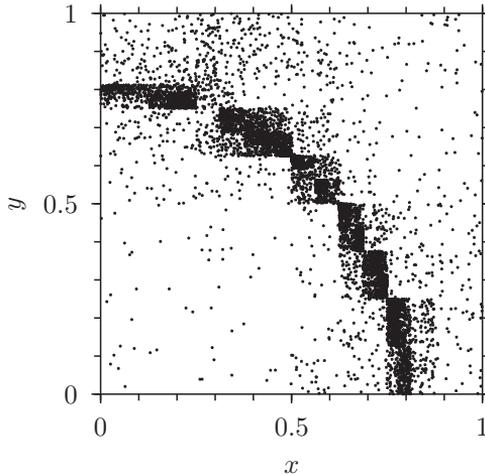
Figure 1: Stratified sample of a discontinuous function, $f(x, y) = 1$ if $x^2 + y^2 < 0.8^2$ otherwise $f(x, y) = 0$, built with the algorithm in Table 2.

where for the given tolerance one might need excessively many points. The GSL's implementation instead takes the number of points as the argument and then returns the estimate of the integral and the estimate of the error.

1.5 Quasi-random (low-discrepancy) sampling

Pseudo-random sampling has high discrepancy[1]: it typically creates regions with high 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.
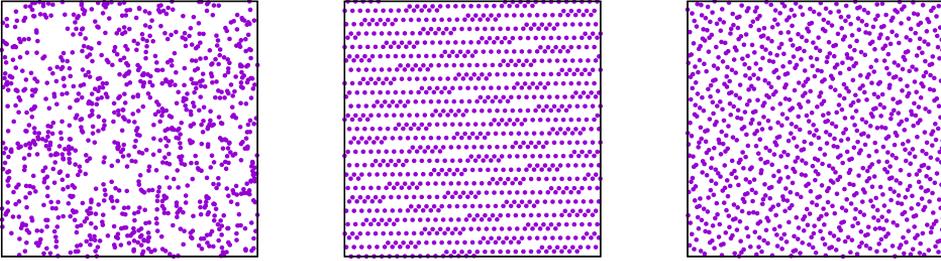
Figure 2: Typical distributions of pseudo-random points (left), and quasi-random low-discrepancy points: additive (center) and base-2/3 Halton (right) sequences. The first thousand points are plotted in each case.

1.5.1 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

$$0.1, 0.2, 0.3, \dots, 0.8, 0.9, 0.01, 0.11, 0.21, 0.31, \dots, 0.91, 0.02, 0.12, \dots . \qquad (9)$$

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

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

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

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

Here is a Python implementation of this algorithm,

```python
def corput(n,b=2):
    q=0; bk=1/b
    while(n>0): q += n % b * bk; n //= b; bk /= b;
    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

5

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

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

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

```
class Halton:
    def __init__(self,dim) : self.bases=prime_numbers(dim)
    def x(self,n) : return [corput(n,b) for b in self.bases]
```

where a (short) list of n prime numbers can be generated as

```
def prime_numbers(n):
    primes=[]; candidate=2
    while len(primes)<n:
        candidate_is_prime=True
        for p in primes:
            if p*p>candidate: break
            if candidate%p==0 : candidate_is_prime=False; break
        if candidate_is_prime: primes.append(candidate)
        candidate+=1
    return primes
```

1.5.2 Additive recurrences (lattice rules)

For any irrational number α the sequence

$$s_n(\alpha) \doteq \text{frac}(n\alpha) \tag{13}$$

(where $\text{frac}(x)$ is the fractional part of x) has discrepancy approaching $1/N$. This sequence can be also defined as an additive recurrence relation,

$$s_{n+1}(\alpha) = \text{frac}(s_n + \alpha) \equiv (s_n + \alpha) \mod 1 \, . \tag{14}$$

In a d-dimensional space one chooses for each dimension a separate sequence with its own irrational number $\alpha_i|_{i=1,\ldots,d}$ (one possibility is to take the square roots of prime numbers modulo 1). The corresponding quasi-random vector $\mathbf{x}$ is then given as

$$\mathbf{x}(n) = \{\text{frac}(n\alpha_1),\ldots,\text{frac}(n\alpha_d)\} \, . \tag{15}$$

This formula is sometimes also referred to as *lattice rule*.

Here is an implementation of this algorithm in Python

```
class Lattice:
    def __init__(self,dim):
        self.alphas=[math.sqrt(p) % 1 for p in prime_numbers(dim)]
    def x(self,n): return [n*alpha % 1 for alpha in self.alphas]
```

and an illustration of such sequence is shown on Figure 2 (center).

1.6 Implementations

Table 3: C implementation of the stratified sampling algorithm

```c
#include<math.h>
#include<stdlib.h>
#include<stdio.h>
#define FOR(k) for(int k=0;k<dim;k++)
#define RND (double)rand()/RAND_MAX

double strata(
    int dim, double f(int dim,double*x),
    double*a,double*b,
    double acc,double eps,
    int n_reuse,double mean_reuse)
{
int N=16*dim;
double V=1; FOR(k) V*=b[k]-a[k];
int n_left[dim], n_right[dim];
double x[dim], mean_left[dim], mean_right[dim],mean=0;
FOR(k){ mean_left[k]=0; mean_right[k]=0; n_left[k]=0; n_right[k]=0; }
for(int i=0;i<N;i++){
    FOR(k) x[k]=a[k]+RND*(b[k]-a[k]);
    double fx=f(dim,x);
    mean+=fx;
    FOR(k){
        if(x[k]>(a[k]+b[k])/2){ n_right[k]++; mean_right[k]+=fx; }
        else                  { n_left[k]++; mean_left[k]+=fx; }
        }
    }
mean/=N;
FOR(k){ mean_left[k]/=n_left[k]; mean_right[k]/=n_right[k]; }

int kdiv=0; double maxvar=0;
FOR(k){
    double var=fabs(mean_right[k]-mean_left[k]);
    if(var>maxvar){ maxvar=var; kdiv=k; }
    }

double integ=(mean*N+mean_reuse*n_reuse)/(N+n_reuse)*V;
double error=fabs(mean_reuse-mean)*V;
double toler=acc+fabs(integ)*eps;
if(error<toler)return integ;

double a2[dim],b2[dim]; FOR(k)a2[k]=a[k]; FOR(k)b2[k]=b[k];
a2[kdiv]=(a[kdiv]+b[kdiv])/2; b2[kdiv]=(a[kdiv]+b[kdiv])/2;
double integ_left=
    strata(dim,f,a,b2,acc/sqrt(2),eps,n_left[kdiv],mean_left[kdiv]);
double integ_right=
    strata(dim,f,a2,b,acc/sqrt(2),eps,n_right[kdiv],mean_right[kdiv]);
return integ_left+integ_right;
}
```