2 Monte Carlo integration

In this chapter we review the basic algorithms for the calculation of integrals using random variables and define the general strategy based on the replacement of an integral by a sample mean

2.1 Hit and miss

The hit and miss method is the simplest of the integration methods that use ideas from probability theory. Although it is not very competitive in practical applications, it is very convenient in order to explain in a simple and comprehensive case some of the ideas of more general methods. Let $y = g(x)$ be a real function which takes only bounded positive values in a finite interval $[a, b]$, i.e. $0 \le g(x) \le c$. In Riemann's sense, the integral

$$I = \int_a^b g(x)\, dx \tag{2.1}$$

is nothing but the area of the region Ω between the X-axis and the curve given by $g(x)$. In the rectangle $S = \{(x, y),\ a \le x \le b,\ 0 \le y \le c\}$ we consider a pair or random variables $(\hat{\mathbf{x}}, \hat{\mathbf{y}})$ which are uniformly distributed in S, see figure 2.1. The joint probability density function of $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ is

$$f_{\hat{\mathbf{x}}\hat{\mathbf{y}}}(x, y) = \begin{cases} \dfrac{1}{c(b-a)} & (x, y) \in S, \\ 0 & (x, y) \notin S. \end{cases} \tag{2.2}$$

The probability that a point (x, y) distributed according to $f_{\hat{\mathbf{x}}\hat{\mathbf{y}}}(x, y)$ lies within Ω is, using (1.65):

$$p = \int_\Omega f_{\hat{\mathbf{x}}\hat{\mathbf{y}}}(x, y)\, dx\, dy = \frac{1}{c(b-a)} \int_\Omega dx\, dy = \frac{I}{c(b-a)} \tag{2.3}$$

as $\int_\Omega dx\, dy$ is the area of Ω, or the integral I. The idea of the Monte Carlo integration is to read this equation as: $I = c(b-a)p$. So, if we know the probability p then we know the integral I. But how can we obtain the probability p that a point distributed

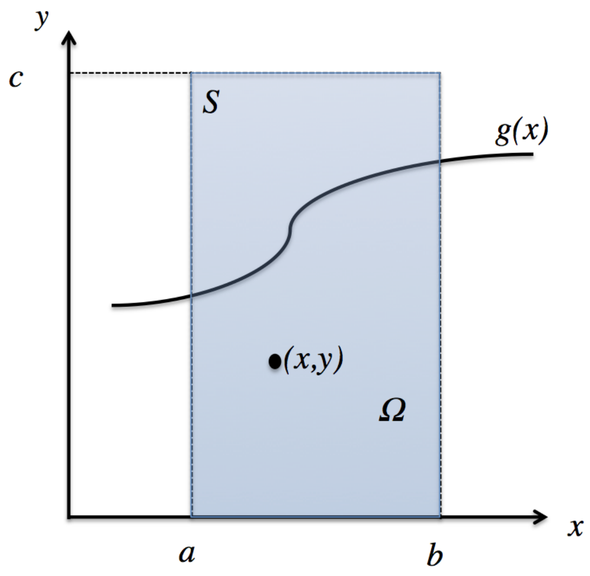

Figure 2.1 Schematics and definitions of the hit and miss method.

according to a uniform distribution lies in region Ω? Simple: perform an experiment whose outcome is the pair (x, y), distributed according to (2.2), and compute from that experiment the probability p. It might help to imagine an archer who is sending arrows uniformly to region S. If he sends 100 arrows and 78 of them are in region Ω then an estimation of the probability is $p = 0.78$. The hit and miss method is a sophisticated manner of replacing the archer giving us, hopefully, a precise value for p as well as the error in the calculation of the integral I.

Let us imagine, then, that we have an ideal (though very inefficient) archer that is able to send arrows that fall uniformly (and this is the key word) in points (x, y) of S. The point (x, y) can or can not fall within Ω and, hence, the event "the point (x, y) is in Ω" can take the values "yes" or "no", a binary variable that follows a Bernoulli distribution. We make M independent repetitions of this Bernoulli experiment, generate the points $(x_1, y_1), (x_2, y_2), ..., (x_M, y_M)$ uniformly distributed in S and define the random variable $\hat{\mathbf{N}}_B$ as the number of points that belong to Ω, i.e. those satisfying $y_i \le g(x_i)$. We introduce a random variable $\hat{\mathbf{I}}_1$, defined from $\hat{p} = \frac{\hat{\mathbf{N}}_B}{M}$, as

$$\hat{\mathbf{I}}_1 = c(b-a)\frac{\hat{\mathbf{N}}_B}{M} = c(b-a)\hat{p}. \tag{2.4}$$

As $\hat{\mathbf{N}}_B$ follows a binomial distribution of mean value pM, it results that $\langle \hat{\mathbf{I}}_1 \rangle = I$. Such a random variable whose average value is equals to I is called an *unbiased estimator* of the integral. Using the results of the binomial distribution we can compute also the root mean square of this estimator as:

$$\sigma[\hat{\mathbf{I}}_1] = \frac{c(b-a)}{M}\sigma[\hat{N}_B] = \frac{c(b-a)}{M}\sqrt{Mp(1-p)} = c(b-a)\sqrt{\frac{p(1-p)}{M}}. \tag{2.5}$$

We can now perform the experiment which consists in generating M points independently and uniformly distributed in the rectangle S and count how many times $\hat{\mathbf{N}}_B$ the points lie in Ω. This gives us a value for the random variable $\hat{p}$. Using the short-hand notation we derived from Chebyshev's theorem we can write:

$$I = \hat{\mathbf{I}}_1 \pm \hat{\sigma}[\hat{\mathbf{I}}_1] = c(b-a)\hat{p} \pm c(b-a)\sqrt{\frac{\hat{p}(1-\hat{p})}{M}}, \tag{2.6}$$

with the usual confidence intervals for the error. If we make a large number of repetitions M we can approximate the binomial by a Gaussian distribution and the confidence levels indicate that there is a probability of 68% that the integral is indeed in the interval $(\hat{\mathbf{I}}_1 - \hat{\sigma}[\hat{\mathbf{I}}_1], \hat{\mathbf{I}}_1 + \hat{\sigma}[\hat{\mathbf{I}}_1])$, 95% that is in the interval $(\hat{\mathbf{I}}_1 - 2\hat{\sigma}[\hat{\mathbf{I}}_1], \hat{\mathbf{I}}_1 + 2\hat{\sigma}[\hat{\mathbf{I}}_1])$, etc. Consequently we associate $\hat{\sigma}[\hat{\mathbf{I}}_1]$ to the error (in the statistical sense) of our estimator. The relative error is:

$$\frac{\sigma[\hat{\mathbf{I}}_1]}{\langle\hat{\mathbf{I}}_1\rangle} \approx \sqrt{\frac{1-p}{pM}}. \tag{2.7}$$

From this expression we obtain (i) the error decreases as the inverse square root $M^{-1/2}$ of the number of repetitions M and (ii) the error decreases with increasing p. Therefore it is convenient to chose the, otherwise arbitrary, rectangle S as small as possible, which is equivalent to taking $c = \max(g(x))$.

In a numerical algorithm, to replace our archer we need to generate random points (x, y) from a two-dimensional uniform distribution. This can be obtained by generating one random variable $\hat{\mathbf{x}}$ uniformly in the interval $[a, b]$ and another, independent, random variable $\hat{\mathbf{y}}$ uniformly in $(0, c)$. In practice, we use two independent random variables $\hat{\mathbf{u}}$, $\hat{\mathbf{v}}$ uniformly distributed in the interval $[0, 1]$ and the lineal transformations $\hat{\mathbf{x}} = a + (b-a)\hat{\mathbf{u}}$, $\hat{\mathbf{y}} = c\hat{\mathbf{v}}$.

So let us assume that we have a way, a black box for the moment being, an algorithm that enables us to generate independent $\hat{\mathbf{U}}(0, 1)$ random variables. We explain in appendix A how to construct such an algorithm. It suffices to say that almost every scientific programming language or library incorporates a built-in function to do the job. The name of the function depends on the programing language and may even vary from compiler to compiler. Popular names are `rand()`, `random()`, `randu()`, etc. Here we will use here the generic name `ran_u()`. In the following, every time we see a call to the function `ran_u()` the return is an independent $\hat{\mathbf{U}}(0, 1)$ random number uniformly distributed in the interval $(0, 1)$.

We implement the hit and miss method as a subroutine. The subroutine returns the estimator of the integral in the variable r and the error in s. In this and other examples, the important part of the code, the one that the reader should read in detail is framed in a box.

```
subroutine mc1(g,a,b,c,M,r,s)
  implicit none
  double precision, intent (in) :: a,b,c
  integer, intent (in) :: M
  double precision, intent (out) :: r,s
  double precision :: g,p,u,v,ran_u
  integer :: na,i
  external g
```

```
    na=0
    do i=1,M
      u=ran_u()
      v=ran_u()
      if (g(a+(b-a)*u).gt.c*v) na=na+1
    enddo
    p=dble(na)/M
    r=(b-a)*c*p
    s=sqrt(p*(1.d0-p)/M)*c*(b-a)
```

```
end subroutine mc1
```

For the sake of clarity we include an example of a program calculating the area of the function $g(x) = x^2$ in the interval $[0, 1]$:

```
program area
  implicit none
  interface
    double precision function g(x)
      double precision, intent (in) :: x
    end function g
  end interface
  double precision :: a,b,c,r,s
  integer :: M
  a=0.d0
  b=1.d0
  c=1.d0
  M=100000
  call mc1(g,a,b,c,M,r,s)
  write (*,*) "The estimated value of the integral is", r
  write (*,*) "The estimated error is", s
end program area

double precision function g(x)
  implicit none
  double precision, intent (in) :: x
  g=x*x
end function g
```

The reader is encouraged to run this simple program and check that the result is compatible, within errors, with the exact value $I = 1/3$.

2.2 Uniform sampling

Let us know explain a second, more sophisticated method to perform integrals using random numbers and ideas from probability theory. We consider again the integral

$$I = \int_a^b g(x)\,dx \tag{2.8}$$

but now $g(x)$ does not need to be necessarily limited to take non-negative values. We write this integral as

$$I = \int G(x) f_{\hat{\mathbf{x}}}(x)\,dx, \tag{2.9}$$

with $G(x) = (b-a)g(x)$ and

$$f_{\hat{\mathbf{x}}}(x) = \begin{cases} \dfrac{1}{b-a} & a \le x \le b, \\ 0 & \text{otherwise.} \end{cases} \tag{2.10}$$

This integral can be understood as the average value of the function $G(x)$ with respect to a random variable $\hat{\mathbf{x}}$ whose probability density function is $f_{\hat{\mathbf{x}}}(x)$, i.e. a uniform random variable in the interval (a,b). If we now devise an experiment whose outcome is the random variable $\hat{\mathbf{x}}$ and repeat that experiment M times to obtain the values $x_1,\dots,x_M$ we can compute from those values the sample mean and variance:

$$\hat{\mu}_M[G] \quad = \quad \frac{1}{M}\sum_{i=1}^{M} G(x_i), \tag{2.11}$$

$$\hat{\sigma}_M^2[G] \quad = \quad \frac{1}{M}\sum_{i=1}^{M} G(x_i)^2 - \left(\frac{1}{M}\sum_{i=1}^{M} G(x_i)\right)^2 . \tag{2.12}$$

note that we have not included the term $\frac{M}{M-1}$ in the definition of $\hat{\sigma}_M^2[G]$ as we will be using these formulas with a large value for M (or the order of thousands or millions) and it has a minimal numerical importance. Using (1.100), the integral can be estimated as:

$$I = \hat{\mu}_M[G] \pm \frac{\hat{\sigma}_M[G]}{\sqrt{M}}, \tag{2.13}$$

where the error has to be understood in the statistical sense of Chebyshev's theorem as explained in section 1.6. If we write, for convenience, everything in terms of the original function $g(x)$, we obtain

$$I = (b-a)\hat{\mu}_M[g] \pm (b-a)\frac{\hat{\sigma}_M[g]}{\sqrt{M}}, \tag{2.14}$$

where $\hat{\mu}_M[g]$ and $\hat{\sigma}_M[g]$ are the sample mean and root-mean-square of the function $g(x)$.

It is very easy to implement this method as we already know how to generate values of a random variable $\hat{\mathbf{x}}$ which is uniformly distributed in an interval (a,b): take a $\hat{\mathbf{U}}(0,1)$ variable $\hat{\mathbf{u}}$ and use the linear transformation $\hat{\mathbf{x}} = (b-a)\hat{\mathbf{u}} + a$. Let us give a simple computer program:

```
subroutine mc2(g,a,b,M,r,s)
  implicit none
  double precision, intent (in) :: a,b
  integer, intent (in) :: M
  double precision, intent (out) :: r,s
  double precision :: g,g0,ran_u
  integer :: i
  external g
    r=0.d0
    s=0.d0
    do i=1,M
      g0=g(a+(b-a)*ran_u())
      r=r+g0
      s=s+g0*g0
    enddo
    r=r/M
    s=sqrt((s/M-r*r)/M)
    r=(b-a)*r
    s=(b-a)*s
end subroutine mc2
```

This simple algorithm is the uniform sampling method. If we think of the Riemann integral as the limiting sum of the function $g(x)$ over an infinite number of points in the interval (a, b), the uniform sampling method replaces that infinte sum by a finite sum (2.11) over points randomly distributed in the same interval.

2.3 General sampling methods

Similar ideas can be used for the integral

$$I = \int G(x) f_{\hat{\mathbf{x}}}(x)\, dx, \qquad (2.15)$$

where again there are no restrictions about the function $G(x)$, but we demand that $f_{\hat{\mathbf{x}}}(x)$ has the required properties of a probability density function: non-negative and normalized. The key point is to understand the integral as the average value of the function $G(x)$ with respect to a random variable $\hat{\mathbf{x}}$ whose probability density function is $f_{\hat{\mathbf{x}}}(x)$:

$$I = E[G]. \qquad (2.16)$$

If we devise now an experiment whose outcome is the random variable $\hat{\mathbf{x}}$ with probability density function $f_{\hat{\mathbf{x}}}(x)$ and repeat that experiment M times to obtain the values $x_1, \ldots, x_M$ we can compute from those values the sample mean and variance using Eqs. (2.11) and (2.12). Then, as before, the integral can be approximated by Eq. (2.13). This simple, but deep, result is the basis of the general sampling method: To evaluate integral (2.15) we consider it as the average of $G(x)$ with respect to a random variable $\hat{\mathbf{x}}$ whose probability distribution function is $f_{\hat{\mathbf{x}}}(x)$ and use the approximation given by the sample mean and the error given by the sample root mean square: formulas (2.11)-(2.12) and (2.13), but using for $x_i, i = 1, \ldots, M$ values of the random variable $\hat{\mathbf{x}}$ whose pdf is $f_{\hat{\mathbf{x}}}(x)$. It is straightforward to write a computer program to implement this algorithm.

```
subroutine mc3(g,ran_f,M,r,s)
  implicit none
  integer, intent (in) :: M
  double precision, intent (out) :: r,s
  double precision :: g,ran_f,g0
  integer :: i
  external g,ran_f
    r=0.d0
    s=0.d0
    do i=1,M
      g0=g(ran_f())
      r=r+g0
      s=s+g0*g0
    enddo
    r=r/M
    s=sqrt((s/M-r*r)/M)
end subroutine mc3
```

The main difference with respect to the uniform sampling is the substitution of the values uniform distribution $(\mathtt{b}-\mathtt{a})\mathtt{ran\_u}()+\mathtt{a}$ distributed according to an uniform distribution in (a, b) by the values $\mathtt{ran\_f}()$ distributed according to the distribution $f_{\hat{\mathbf{x}}}(x)$. And how do we implement this function? Although we will devote next chapter entirely to this question, let us give now some basic results.

2.4 Generation of non-uniform random numbers: basic concepts

The basic method to generate values of a random variable $\hat{\mathbf{x}}$ distributed according to the probability distribution function $f_{\hat{\mathbf{x}}}(x)$ uses the theorem proven in section 1.3:

Theorem.- If $\hat{\mathbf{x}}$ is a random variable whose cumulative probability function is $F_{\hat{\mathbf{x}}}(x)$, then the change of variables $\hat{\mathbf{u}} = F_{\hat{\mathbf{x}}}(\hat{\mathbf{x}})$ yields a random variable $\hat{\mathbf{u}}$ uniformly distributed in the interval $(0, 1)$, a $\hat{\mathbf{U}}(0, 1)$ variable.

We read this theorem backwards: if $\hat{\mathbf{u}}$ is a $\hat{\mathbf{U}}(0, 1)$ random variable, then the probability distribution function of $\hat{\mathbf{x}} = F_{\hat{\mathbf{x}}}^{-1}(\hat{\mathbf{u}})$ is $f_{\hat{\mathbf{x}}}(x)$, see figure 2.2. Let us see an example. We want to generate random numbers distributed according to an **exponential distribution**:

$$f_{\hat{\mathbf{x}}}(x) = \begin{cases} 0, & x < 0, \\ a\exp(-ax), & x \geq 0, \end{cases} \tag{2.17}$$

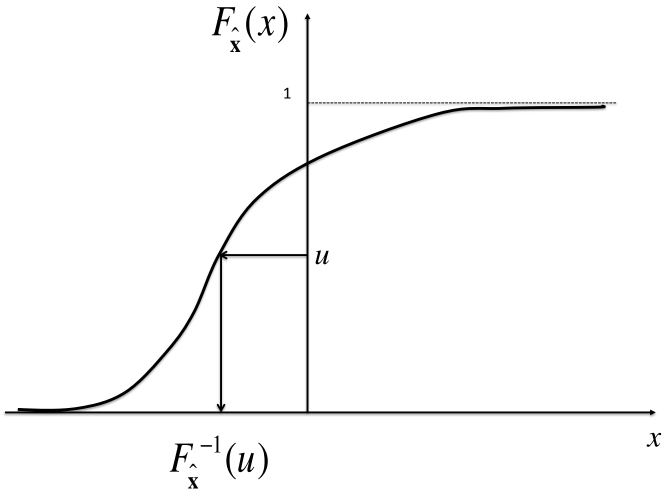

Figure 2.2 Inversion of the cdf $F_{\hat{\mathbf{x}}}(x)$. If u is a value of a random variable $\hat{\mathbf{u}}$ uniformly distributed in the $(0,1)$ interval, then $x = F_{\hat{\mathbf{x}}}^{-1}(u)$ follows the corresponding pdf $f_{\hat{\mathbf{x}}}(x) = \dfrac{F_{\hat{\mathbf{x}}}^{-1}(x)}{dx}$.

We first compute the cumulative distribution

$$F_{\hat{\mathbf{x}}}(x) = \int_{-\infty}^{x} f_{\hat{\mathbf{x}}}(x')\,dx' = 1 - \exp(-ax), \tag{2.18}$$

and then invert $x = F_{\hat{\mathbf{x}}}^{-1}(u)$ by solving $u = F_{\hat{\mathbf{x}}}(x)$ or $x = \frac{-1}{a}\log(1-u)$. This means that to generate random numbers x distributed according to an exponential distribution, we need to generate numbers u uniformly distributed in the $(0,1)$ and then apply $x = \frac{-1}{a}\log(1-u)$. As $1-\hat{\mathbf{u}}$ is obviously also a $\hat{\mathbf{U}}(0,1)$ variable, we can simply use the formula:

$$x = \frac{-1}{a}\log(u). \tag{2.19}$$

The algorithm can be implemented in the following simple program[1]:

```
double precision function ran_exp(a)
  implicit none
  double precision, intent (in) :: a
  double precision :: ran_u
```

1) A possible problem with this algorithm one needs to be aware of is that it might occur that the uniform random number u takes exactly the value $u = 0$. This can be avoided generating the uniform numbers in a suitable way as discussed in the appendix 12.

```
    ran_exp=-log(ran_u())/a
end function ran_exp
```

Let us see some more examples:

Cauchy distribution: The probability distribution function is:

$$f_{\hat{\mathbf{x}}}(x) = \frac{1}{\pi}\frac{1}{1+x^2}. \tag{2.20}$$

The cumulative function is

$$F_{\hat{\mathbf{x}}}(x) = \int_{-\infty}^{x} f_{\hat{\mathbf{x}}}(x)\,dx = \int_{-\infty}^{x} \frac{1}{\pi}\frac{1}{1+x^2} = \frac{1}{2} + \frac{1}{\pi}\text{arctan}(x), \tag{2.21}$$

and its inverse function:

$$x = \tan\left(\pi(u - \frac{1}{2})\right). \tag{2.22}$$

To generate random values distributed according to the more general distribution:

$$f_{\hat{\mathbf{x}}}(x) = \frac{a}{\pi}\frac{1}{1+a^2(x-x_0)^2}, \tag{2.23}$$

we do not need to repeat the whole process. We simply note that the variable $\hat{\mathbf{z}}$ defined by $\hat{\mathbf{z}} = a(\hat{\mathbf{x}} - x_0)$ follows the Cauchy distribution $f_{\hat{\mathbf{z}}}(z) = \frac{1}{\pi}\frac{1}{1+z^2}$ and use $x = x_0 + a^{-1}\tan\left(\pi(u - \frac{1}{2})\right)$.

A power-law distribution in a bounded domain: Consider the random variable $\hat{\mathbf{x}}$ distributed according to:

$$f_{\hat{\mathbf{x}}}(x) = \begin{cases} (1+a)x^a, & x \in [0,1], \\ 0, & x \notin [0,1], \end{cases} \tag{2.24}$$

with $a > -1$ (otherwise, it is not normalizable). The cumulative distribution is $F_{\hat{\mathbf{x}}}(x) = x^{a+1}$ if $x \in [0,1]$ and the inversion is $x = u^{1/(1+a)}$.

A power-law distribution in an infinite domain. This kind of random variables appear in many modern applications of the field of complex systems. Roughly speaking, a random variable defined in the infinite domain $(0,\infty)$ is said to have a power-law distribution $f_{\hat{\mathbf{x}}}(x)$ with exponent $a > 0$ if $f_{\hat{\mathbf{x}}}(x) \sim x^{-a}$ for large x. The problem is that this definition is not very precise. There are many functions $f_{\hat{\mathbf{x}}}(x)$ which behave as $f_{\hat{\mathbf{x}}}(x) \sim x^{-a}$ for large x and one needs to specify which one is needed in a particular problem. Note that one can not use $f_{\hat{\mathbf{x}}}(x) = Cx^{-a}$ for all $x \in (0,\infty)$ as this function is not normalizable for any value of the exponent a. Many functions $f_{\hat{\mathbf{x}}}(x)$ have been proposed. We will consider here the specific example:

$$f_{\hat{\mathbf{x}}}(x) = \begin{cases} 0, & x < 0, \\ \frac{a-1}{x_0}\left(1 + \frac{x}{x_0}\right)^{-a}, & x \in [0,\infty), \end{cases} \tag{2.25}$$

which is well defined if $a > 1$ and $x_0 > 0$. The cumulative function for $x \geq 0$ is

$$F_{\hat{\mathbf{x}}}(x) = 1 - \left(1 + \frac{x}{x_0}\right)^{1-a}, \tag{2.26}$$

and the inversion $x = F_{\hat{\mathbf{x}}}^{-1}(u)$ leads to

$$x = (u^{\frac{1}{1-a}} - 1)\, x_0, \tag{2.27}$$

where we have replaced u by $1-u$ as they are statistically equivalent.

Other possibilities for a *bona fide* power-law distribution imply the use of cut-offs. For example, a distribution cut-off at small values of x:

$$f_{\hat{\mathbf{x}}}(x) = \begin{cases} 0, & x < x_0, \\ \frac{a-1}{x_0}\left(\frac{x}{x_0}\right)^{-a}, & x \in [x_0, \infty), \end{cases} \tag{2.28}$$

valid for $x_0 > 0$ and $a > 1$. It is easy to show that values of this random variable can be generated using $x = u^{\frac{1}{1-a}}\, x_0$.

Rayleigh distribution: A random variable $\hat{\mathbf{r}}$ that follows the Rayleigh distribution has as probability density function:

$$f_{\hat{\mathbf{r}}}(r) = \begin{cases} 0, & r < 0, \\ re^{-\frac{1}{2}r^2}, & 0 \le r \le \infty. \end{cases} \tag{2.29}$$

The cumulative function is:

$$F_{\hat{\mathbf{r}}}(r) = \int_{-\infty}^{r} f_{\hat{\mathbf{r}}}(r)\, dr = 1 - e^{-\frac{1}{2}r^2}, \tag{2.30}$$

and the inverse function

$$r = \sqrt{-2\log(1-u)} \equiv \sqrt{-2\log(u)} \tag{2.31}$$

where again we can replace u by $1-u$ as they are statistically equivalent. We will be using this formula later when discussing the Box-Muller-Wiener algorithm for the Gaussian distribution.

Gaussian distribution. If $\hat{\mathbf{x}}$ is a Gaussian random variable of mean μ and variance σ^2, with probability density function

$$f_{\hat{\mathbf{x}}}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \tag{2.32}$$

then $\hat{\mathbf{z}} = \frac{\hat{\mathbf{x}}-\mu}{\sigma}$ is a Gaussian random variable of zero mean and variance 1 with probability density function:

$$f_{\hat{\mathbf{z}}}(z) = \frac{1}{\sqrt{2\pi}} e^{-\frac{z^2}{2}}. \tag{2.33}$$

Then, in order to generate $\hat{\mathbf{x}}$ all we need to do is generate $\hat{\mathbf{z}}$ and apply the linear change $\hat{\mathbf{x}} = \sigma\hat{\mathbf{z}} + \mu$. The cumulative function of $\hat{\mathbf{z}}$ is:

$$F_{\hat{\mathbf{z}}}(z) = \int_{-\infty}^{z} f_{\hat{\mathbf{z}}}(z)\, dz = \int_{-\infty}^{z} \frac{1}{\sqrt{2\pi}} e^{-\frac{z^2}{2}}\, dz = \frac{1}{2}\left(1 + \operatorname{erf}\left(z/\sqrt{2}\right)\right), \tag{2.34}$$

and values of the random variable $\hat{\mathbf{z}}$ are obtained from:

$$z = \sqrt{2}\operatorname{erf}^{-1}(2u-1) \tag{2.35}$$

A possible problem with this expression is that the inverse error function $\mathrm{erf}^{-1}(x)$ is not standard in most programming languages and you will need to search for it in an appropriate scientific library. For instance, in Intel's Math Kernel Library it is called `erfinv(x)`. One could use, alternatively, some of the good approximations to the inverse error function $z = F_{\hat{\mathbf{z}}}^{-1}(u)$,

$$z \approx t - \frac{c_0 + c_1 t + c_2 t^2}{1 + d_1 t + d_2 t^2 + d_3 t^3}, \qquad t = \sqrt{-2\log(1-u)} \tag{2.36}$$

and the numerical values c_0=2.515517, c_1=0.802853, c_2=0.010328, $d1$=1.432788, d_2=0.189269, d_3=0.001308. The error is less than 4.5×10^{-4} if $0.5 \le u \le 1.0$. For $0 < u < 0.5$ we can use the symmetry property of the Gaussian distribution around $x = 0$ to write up the following computer program:

```
double precision function ran_g()
  implicit none
  double precision, parameter :: c0=2.515517d0
  double precision, parameter :: c1=0.802853d0
  double precision, parameter :: c2=0.010328d0
  double precision, parameter :: d1=1.432788d0
  double precision, parameter :: d2=0.189269d0
  double precision, parameter :: d3=0.001308d0
  double precision :: u,t,ran_u
  u=ran_u()
   if (u.gt.0.5d0) then
     t=sqrt(-2.d0*log(1.-u))
     ran_g=t-(c0+t*(c1+c2*t))/(1.d0+t*(d1+t*(d2+t*d3)))
   else
     t=sqrt(-2.d0*log(u))
     ran_g=-t+(c0+t*(c1+c2*t))/(1.d0+t*(d1+t*(d2+t*d3)))
   endif
end function ran_g
```

Besides the intrinsic error which might or might not be important in a particular application, this approximation to the inverse error function is relatively slow as it involves the calculation of a square root, a logarithm and a ratio of polynomials. In the next chapter we will develop alternative algorithms for the Gaussian distribution that use faster functions.

The problem of the lack of a good algorithm for the calculation of the inverse cumulative distribution function is very general and, sometimes, it is a bottleneck for the implementation of the method described in this section. For example, consider a variable distributed according to the distribution:

$$f_{\hat{\mathbf{x}}}(x) = \begin{cases} 0, & x < 0, \\ xe^{-x}, & x \ge 0, \end{cases} \tag{2.37}$$

(it belongs to the Gamma-distribution family, concretely it is the $\hat{\mathbf{\Gamma}}(2,1)$ distribution, see more in the next chapter). The cumulative distribution is:

$$F_{\hat{\mathbf{x}}}(x) = \begin{cases} 0, & x < 0, \\ 1-(1+x)e^{-x}, & x > 0. \end{cases} \tag{2.38}$$

Now, given a value u uniformly distributed in the interval $(0,1)$ all we need to do is to find the positive solution x of the non-algebraic equation $u = F_{\hat{\mathbf{x}}}(x) = 1-(1+x)e^{-x}$. This could be solved, for example, using Newton-Raphson method with the recursion relation[2)]:

$$x_{n+1} = x_n - \frac{F_{\hat{\mathbf{x}}}(x_n)-u}{f_{\hat{\mathbf{x}}}(x_n)}, \tag{2.39}$$

where we have replace $F'_{\hat{\mathbf{x}}}(x) = f_{\hat{\mathbf{x}}}(x)$. As a initial condition we can expand $F_{\hat{\mathbf{x}}}(x) = x^2/2 + O(x^3) = u$ and use $x_0 = \sqrt{2u}$. A possible program could be:

```
double precision function ran_gamma2()
  implicit none
  double precision :: u,x,xn,fx,fxc,ran_u

    fx(x)=x*exp(-x)
    fxc(x)=1.d0-exp(-x)*(1.d0+x)
    u=ran_u()
    x=sqrt(2.d0*u)
    xn=x-(fxc(x)-u)/fx(x)
    do while(abs(xn-x).gt.1.d-8)
      xn=x
      x=x-(fxc(x)-u)/fx(x)
    enddo
    ran_gamma2=x

end function ran_gamma2
```

The same procedure can be used, for instance, to generate the distribution (the $\hat{\mathbf{\Gamma}}(3,1)$ member of the gamma family):

$$f_{\hat{\mathbf{x}}}(x) = \begin{cases} 0, & x<0, \\ \frac{1}{2}x^2e^{-x}, & x>0. \end{cases} \tag{2.40}$$

with a cumulative distribution:

$$F_{\hat{\mathbf{x}}}(x) = \begin{cases} 0, & x<0, \\ 1-\left(1+x+\frac{1}{2}x^2\right)e^{-x}, & x>0. \end{cases} \tag{2.41}$$

The Newton-Raphson algorithm can be implemented in the following program using the initial value $x_0 = (6u)^{1/3}$ obtained by expanding $F_{\hat{\mathbf{x}}}(x) = \frac{1}{6}x^3 + O(x^4)$:

```
double precision function ran_gamma3()
  implicit none
  double precision :: u,x,xn,fx,fxc,ran_u
```

2) Other methods of solution will also work. For instance, write it in the form $x = -\log\left(\frac{u}{1+x}\right)$ and use the recursion relation $x_{n+1} = -\log\left(\frac{u}{1+x_n}\right)$ with $x_0 = 1$. The reader can check that this procedure always converges to the appropriate, positive, solution for x.

```
  fx(x)=0.5d0*x*x*exp(-x)
  fxc(x)=1.d0-exp(-x)*(1.d0+x*(1.d0+0.5d0*x))
  u=ran_u()
  x=(6.d0*u)**(1.d0/3.d0)
  xn=x-(fxc(x)-u)/fx(x)
  do while(abs(xn-x).gt.1.d-8)
    xn=x
    x=x-(fxc(x)-u)/fx(x)
  enddo
  ran_gamma3=x
end function ran_gamma3
```

In the next chapter, however, we will see more efficient methods to generate values of random variables distributed according to gamma-type distributions.

The same method of the inversion of the cumulative distribution function can be used in the case of discrete random variables, those taking values from a numerable (maybe infinite) set $(x_1, x_2, \dots)$ ordered as $x_1 < x_2 < x_3 \dots$. If p_i is the probability that the random variable $\hat{\mathbf{x}}$ takes the value x_i, the pdf is:

$$f_{\hat{\mathbf{x}}}(x) = \sum_{i=1} p_i \delta(x - x_i), \tag{2.42}$$

and the corresponding cdf

$$F_{\hat{\mathbf{x}}}(x) = \int_{-\infty}^{x} f_{\hat{\mathbf{x}}}(x)\, dx = \sum_{i=1}^{m} p_i, \tag{2.43}$$

here m stands for the largest integer number for which it is $x_m \le x$. The cdf is a step function. To invert $F_{\hat{\mathbf{x}}}(x)$ we need to determine to which interval $[x_m, x_{m+1})$ does $F_{\hat{\mathbf{x}}}^{-1}(u)$ belong to, see figure 2.3 In general, it is not easy to find the interval $[x_m, x_{m+1})$ and it is better to consider the equivalent problem of finding the maximum value of m for which

$$F_{\hat{\mathbf{x}}}(x_m) = \sum_{i=1}^{m} p_i \le u. \tag{2.44}$$

If the number of values that the random variable $\hat{\mathbf{x}}$ can take is not too big, it might be convenient to check directly to which interval does the uniform random number belongs. For instance, let us consider the distribution

$$f_{\hat{\mathbf{x}}}(x) = \frac{1}{8}\delta(x) + \frac{3}{8}\delta(x - \frac{1}{2}) + \frac{1}{6}\delta(x-2) + \frac{1}{3}\delta(x-4), \tag{2.45}$$

for which we could use the following program

```
double precision function ran_discrete_f()
  implicit none
  double precision :: u,p1,p2,p3,ran_u
```

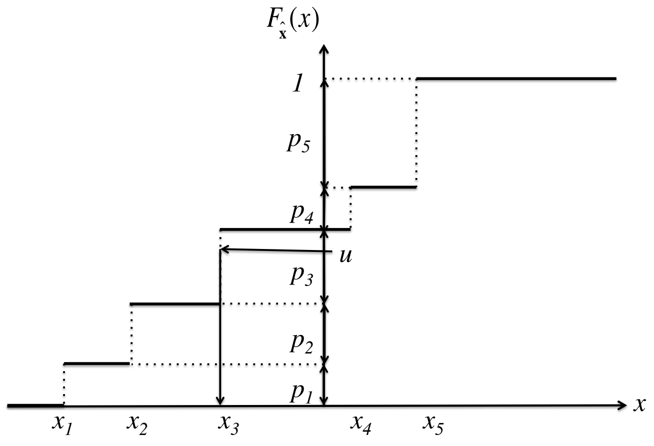

Figure 2.3 Inversion of the cdf for a discrete distribution.

```
  u=ran_u()
  s1=1.d0/8.d0
  s2=s1+3.d0/8.d0
  s3=s2+1.d0/6.d0
  if (u.lt.s1) then
    ran_f=0.d0
  elseif(u.lt.s2) then
    ran_f=0.5d0
  elseif(u.lt.s3) then
    ran_f=2.d0
  else
    ran_f=4.d0
  endif
end function ran_discrete_f
```

A particular case is that of the Bernoulli distribution for which the cumulative distribution function is (1.29). The inverse function takes only two values:

$$F_{\hat{\mathbf{x}}}^{-1}(u) = \begin{cases} 0 & u < 1-p, \\ 1 & u \geq 1-p, \end{cases} \tag{2.46}$$

see figure 1.2. Hence, a program to generate a Bernoulli random variable with parameter p could be:

```
integer function iran_bern(p)
  implicit none
  double precision, intent (in) :: p
  double precision :: ran_u
    if (ran_u().lt.1.0d0-p) then
      iran_bern=0
    else
      iran_bern=1
    endif
end function iran_bern
```

Replacing u by $1-u$ (as they are statistically equivalent), the relevant lines of this program can also be written as:

```
    if (ran_u().gt.p) then
      iran_bern=0
    else
      iran_bern=1
    endif
```

If the random variable $\hat{\mathbf{x}}$ takes N values $x_1,\dots,x_N$ with the same probability, i.e. $p_i=\frac{1}{N}$, it is possible to find the desired interval $[x_m,x_{m+1})$ as[3] $m=[Nu]+1$. Hence, to sample

$$f_{\hat{\mathbf{x}}}(x)=\sum_{i=1}^{N}\frac{1}{N}\delta(x-x_i) \tag{2.47}$$

we take u from a $\hat{\mathbf{U}}(0,1)$ distribution and take[4] $x=x_{[Nu]+1}$.

If the discrete random variable can take an infinite numerable number of values, it is usually difficult to invert efficiently the cdf $F_{\hat{\mathbf{x}}}(x)$. A notable exception is the **geometric distribution** which takes integer values $x=i$ with probability $p_i=pq^i$ (with $q=1-p$). The cdf is

$$F_{\hat{\mathbf{x}}}(m)=\sum_{i=0}^{m}p_i=\sum_{i=0}^{m}pq^i=1-q^{m+1}. \tag{2.48}$$

We need to find the maximum integer value m for which

$$1-q^{m+1}\le u\to q^{m+1}\ge 1-u\equiv u, \tag{2.49}$$

or

$$m=\left[\frac{\log(u)}{\log(q)}\right]. \tag{2.50}$$

The program could look like[5]:

3) Recall that $[z]$ denotes the integer part of the real variable z.
4) This algorithm assumes that the value $u=1$ can never appear exactly. This is true for the random numbers of the form $u=m/M$ with $0\le m\le M-1$ discussed in the appendix 12.
5) Now we need to make sure that the random number generator can not return exactly the value $u=0$.

```
integer function iran_geo(p)
  implicit none
  double precision, intent (in) :: p
  double precision :: ran_u

    iran_geo=int(log(ran_u())/log(1.d0-p))

end function iran_geo
```

In other cases of discrete variables, the cdf $F_{\hat{\mathbf{x}}}(x)$ might be difficult to invert. Consider the discrete distribution: $p_i = (1+i)p^2q^i$, with $q = 1-p$, a modified geometric distribution. The cumulative distribution is:

$$F_{\hat{\mathbf{x}}}(m) = \sum_{i=0}^{m} p_i = 1 - q^{m+1}(1 + p(m+1)), \tag{2.51}$$

and we need to find the maximum integer value m for which $F_{\hat{\mathbf{x}}}(m) \le u$, or

$$q^{m+1}(1 + p(m+1)) \ge 1 - u \equiv u. \tag{2.52}$$

This equation needs to be solved numerically. For instance, we can set $x = m+1$ and set the iteration scheme:

$$\begin{aligned} x_0 &= \frac{\log(u)}{\log(q)}, \\ x_{n+1} &= \frac{\log u - \log(1 + px_n)}{\log q}, \end{aligned} \tag{2.53}$$

and iterate until $|x_n - x_{n+1}|$ is less than, say 10^{-6}. Next, we set $m = [x]$, the integer part of x. The program would be:

```
integer function iran_modgeo(p)
  implicit none
  double precision, intent (in) :: p
  double precision :: a,v,x0,x,ran_u

    a=log(1.d0-p)
    v=log(ran_u())
    x0=v/a
    x=x+1.d0
    do while(abs(x0-x).gt.1.d-6)
      x=x0
      x0=(v-dlog(1.d0+p*x))/a
    enddo
    iran_modgeo=int(x)

end function iran_modgeo
```

Sometimes, it is not even possible to find a suitable analytical expression for $F_{\hat{\mathbf{x}}}(m) = \sum_{i=0}^{m} p_i$. In this case, what we can do is to find directly the solution of $F_{\hat{\mathbf{x}}}(m) \le u$. This can be generically implemented as follows:

```
integer function iran_generic(p)
  implicit none
  double precision :: p
  double precision :: F,v,ran_u
  integer :: i
  external p
    v=ran_u()
    F=p(0)
    i=0
    do while(F.lt.v)
      i=i+1
      F=F+p(i)
    enddo
    iran_generic=i
end function iran_generic
```

Where p(i) is a external function which returns the value of p_i. For example, for the distribution $p_i = \frac{6}{\pi^2}(i+1)^{-2}$ the function p(i) can be written as:

```
double precision function p(i)
  implicit none
  integer, intent (in) :: i
  double precision, parameter :: coeff=0.6079271018540266261d0
  p=coeff/(1+i)**2
end function p
```

Since in this case, $F_{\hat{\mathbf{x}}}(m) \le u$ corresponds to $\sum_{i=0}^{m}(i+1)^{-2} \le \frac{\pi^2}{6}u$. Therefore one can also implement directly:

```
integer function iran_pot2()
  implicit none
  double precision, parameter :: pi26=1.644934066848226d0
  double precision :: v,F,ran_u
  integer :: m
    v=pi26*ran_u()
    F=1.d0
    m=0
    do while(F.lt.v)
      m=m+1
      F=F+1.d0/(m+1)**2
    enddo
    iran_pot2=m
end function iran_pot2
```

The same ideas can be applied to the Poisson distribution of parameter λ, $\hat{\mathbf{P}}(\lambda)$, which takes the integer value $i = 0, 1, 2, \dots$ with probability $p_i = e^{-\lambda}\lambda^i/i!$. Given a number u extracted from a uniform $\hat{\mathbf{U}}(0,1)$ distribution, we need to find the maximum value m which satisfies:

$$\sum_{i=0}^{m} e^{-\lambda}\frac{\lambda^i}{i!} \le u. \tag{2.54}$$

Although this sum can be written out in terms of known functions, namely

$$\sum_{i=0}^{m} e^{-\lambda}\frac{\lambda^i}{i!} = \frac{\Gamma(m+1,\lambda)}{\Gamma(m+1)}, \tag{2.55}$$

being $\Gamma(x,\lambda) = \int_\lambda^\infty dt\, t^{x-1}e^{-t}$ the incomplete Gamma function, this does not help us in the solution of (2.54), as both $\Gamma(x)$ and $\Gamma(x,\lambda)$ are not part of the standard set of most compiling languages. A better alternative is to solve (2.54) directly. We first write it as

$$\sum_{i=0}^{m} \frac{\lambda^i}{i!} \le ue^{\lambda}. \tag{2.56}$$

We find the value of m by adding terms to the left-hand-side of this expression until the inequality is not satisfied. In the calculation of the terms of the sum it is helpful to use the relation:

$$\frac{\lambda^{i+1}}{(i+1)!} = \frac{\lambda}{i+1}\frac{\lambda^i}{i!}, \tag{2.57}$$

leading to the algorithm:

```
integer function iran_poisson(lambda)
  implicit none
  double precision, intent (in) :: lambda
  double precision :: v,F,a,ran_u
  integer :: m
```

```
    F=1.d0
    v=dexp(lambda)*ran_u()
    a=F
    m=0
    do while(F.lt.v)
     a=a*lambda/(m+1)
     F=F+a
     m=m+1
    enddo
    iran_poisson=m
```

```
end function iran_poisson
```

This algorithm works fine for small values of λ but its efficiency worsens for large λ. The reason is simple to understand. We know that the Poisson distribution has mean value and variance equal to λ. This means that most of the times the generated values belong to the interval $\lambda - \sqrt{\lambda}, \lambda + \sqrt{\lambda}$. As we begin searching from $m = 0$, it takes, on average, λ sums to solve (2.54). Why not then start by searching the solution of (2.54) using a value of m close to λ, namely $m = [\lambda]$, instead of starting from $m = 0$? All we need to do is to compute first the partial sum

$$F = \sum_{i=0}^{[\lambda]} \frac{\lambda^i}{i!}, \tag{2.58}$$

as well as the latest term $a = \dfrac{\lambda^{[\lambda]}}{[\lambda]!}$. If $F < ue^{\lambda}$, this means that the required value of m that satisfies (2.56) is larger that $[\lambda]$ and we keep on increasing m until we find its largest value for which (2.56) holds. If, on the other hand, $F > ue^{\lambda}$, it is $m < [\lambda]$, and we decrease m and subtract terms from the sum until the condition (2.56) is satisfied. In this way, we need to sum of the order of $\sqrt{\lambda}$ terms to find the solution of (2.54). A big improvement if λ is large.

As a practical issue, in this case, to find the corresponding terms of the sum we use

$$\frac{\lambda^{i-1}}{(i-1)!} = \frac{i}{\lambda}\frac{\lambda^i}{i!}. \tag{2.59}$$

Here comes a program implementing these ideas.

```
integer function iran_poisson(lambda,a0,F0)
  implicit none
  double precision, intent (in) :: lambda
  double precision :: v,F,F0,a,a0,ran_u
  integer :: m
```

```
    v=dexp(lambda)*dran_u()
    m=lambda
    a=a0
    F=F0
    if (F.lt.v) then
      do  while(F.lt.v)
         m=m+1
         a=a*lambda/m
         F=F+a
      enddo
    else
      do  while(F.gt.v)
         m=m-1
         F=F-a
         a=a*(m+1)/lambda
      enddo
      m=m+1
    endif
    iran_poisson=m
```

```
end function iran_poisson
```

where, in the main program, we need to add the lines:

```
    F0=1.0d0
    a0=F0
    do i=1,lambda
       a0=a0*lambda/i
       F0=F0+a0
    enddo
```

In any event, neither of these programs is useful for very large λ as the calculation of e^λ can easily yield an overflow. There are alternative ways to generate a Poisson distribution. We will review them in a later chapter.

The final example is that of the binomial distribution, where $p_i = \binom{N}{i} p^i (1-p)^{N-i}$. According to the general procedure, we need to find the maximum value of m for which:

$$\sum_{i=0}^{m} \binom{N}{i} p^i (1-p)^{N-i} \le u, \tag{2.60}$$

where u is a random number taken from a $\hat{\mathbf{U}}(0,1)$ distribution. When programming this algorithm it is useful to use the relation between two consecutive terms in the sum:

$$\binom{N}{i+1} p^{i+1} (1-p)^{N-i-1} = \frac{p}{1-p}\frac{N-i}{i+1}\binom{N}{i} p^i (1-p)^{N-i}, \tag{2.61}$$

valid for $i = 0, 1, \ldots, N-1$, and that the first term is $(1-p)^N$. A possible implementation is:

```
integer function iran_binomial(N,p)
  implicit none
  integer, intent (in) :: N
  double precision, intent (in) :: p
  double precision :: u,a,F,ran_u
  integer :: m
    a=(1.d0-p)**N
    F=a
    m=0
    u=ran_u()
    do while(F.lt.u)
      a=a*p/(1.d0-p)*(N-m)/(m+1.d0)
      F=F+a
      m=m+1
    enddo
    iran_binomial=m
end function iran_binomial
```

As before, we could start by checking first the value of m equal to the average value of the distribution pN and then increase or decrease m as needed. In the next chapter, however, we will see alternative methods for the generation of the binomial distribution.

This ends our short introduction to the basic methods for the generation of random variables distributed according to a given probability distribution. We will devote more time to this important topic in next chapters, but now let us return to the explanation of different integration techniques.

2.5 Importance sampling

Let us consider again the general integral

$$I = \int g(x)\, dx, \qquad (2.62)$$

that we write in the form:

$$I = \int G(x) f_{\hat{\mathbf{x}}}(x)\, dx, \qquad (2.63)$$

with $G(x) = \dfrac{g(x)}{f_{\hat{\mathbf{x}}}(x)}$ and $f_{\hat{\mathbf{x}}}(x)$ has to be a probability density function (non-negative and normalized). In the sampling method we considering this to be equal to the average $E[G]$ and use the approximation used by the sample mean.

There are infinite ways in which (2.62) can be decomposed as in (2.63), although some might be more "natural" than others. For instance, take the integral,

$$I = \int_0^\infty dx\, \cos(x) x^2 \mathrm{e}^{-x}, \qquad (2.64)$$

which is $I = -1/2$. If we would like to use a sampling method for its calculation, possible "natural" choices would be

$$\begin{aligned} G^{(1)}(x) &= \cos(x)x^2, \;\; x \ge 0, & (2.65)\\ f^{(1)}_{\hat{\mathbf{x}}}(x) &= \begin{cases} 0, & x < 0, \\ e^{-x}, & x \ge 0, \end{cases} & (2.66) \end{aligned}$$

or

$$\begin{aligned} G^{(2)} &= \cos(x)x, \;\; x \ge 0, & (2.67)\\ f^{(2)}_{\hat{\mathbf{x}}}(x) &= \begin{cases} 0, & x < 0, \\ xe^{-x}, & x \ge 0, \end{cases} & (2.68) \end{aligned}$$

or

$$\begin{aligned} G^{(3)} &= 2\cos(x), \quad x \ge 0, & (2.69)\\ f^{(3)}_{\hat{\mathbf{x}}}(x) &= \begin{cases} 0, & x < 0, \\ \frac{1}{2}x^2e^{-x}, & x \ge 0, \end{cases} & (2.70) \end{aligned}$$

but we could have used, for example, a not so obvious splitting:

$$\begin{aligned} G^{(4)}(x) &= \begin{cases} 0, & x < 0, \\ \pi(x^2+1)\cos(x)x\mathrm{e}^{-x}, & x \ge 0. \end{cases} & (2.71)\\ f^{(4)}_{\hat{\mathbf{x}}}(x) &= \frac{1}{\pi}\frac{1}{1+x^2}, \qquad\qquad \forall x. & (2.72) \end{aligned}$$

Which is the best way to split $g(x) = G(x) f_{\hat{\mathbf{x}}}(x)$? Of course, a possible criterion is that the numerical generation of random numbers according to $f_{\hat{\mathbf{x}}}(x)$ turns out to be easy from a practical point of view. But leaving aside this, otherwise very reasonable, requirement, a sensible condition to chose the splitting is to obtain an algorithm with the minimum statistical error. If we look at (2.13), the smallest error is obtained when the sample variance $\hat{\sigma}_M^2[G]$ is minimum, or, equivalently, when the variance as given by:

$$\sigma^2[G] = \int G[x]^2 f_{\hat{\mathbf{x}}}(x)dx - \left(\int G(x) f_{\hat{\mathbf{x}}}(x)dx\right)^2 = \int \frac{g(x)^2}{f_{\hat{\mathbf{x}}}(x)}\, dx - I^2 \qquad (2.73)$$

is a minimum. As I is independent of $f_{\hat{\mathbf{x}}}(x)$, the minimization of $\sigma^2[G]$ is equivalent to the minimization of $\int \frac{g(x)^2}{f_{\hat{\mathbf{x}}}(x)}\, dx$. The minimization has to be achieved in the space of functions $f_{\hat{\mathbf{x}}}(x)$ which can be considered as probability density functions, i.e. those functions satisfying:

$$f_{\hat{\mathbf{x}}}(x) \geq 0, \qquad (2.74)$$

$$\int f_{\hat{\mathbf{x}}}(x)\, dx = 1. \qquad (2.75)$$

The optimal solution $f_{\hat{\mathbf{x}}}^{\text{opt}}(x)$ can be found using the method of the Lagrange multipliers. Let us introduce the functional $\mathcal{L}\left[f_{\hat{\mathbf{x}}}\right]$:

$$\mathcal{L}\left[f_{\hat{\mathbf{x}}}\right] = \int \frac{g(x)^2}{f_{\hat{\mathbf{x}}}(x)}\, dx + \lambda \int f_{\hat{\mathbf{x}}}(x)\, dx, \qquad (2.76)$$

being λ the Lagrange multiplier needed to take into account the normalization condition (2.75). The minimization of $\mathcal{L}\left[f_{\hat{\mathbf{x}}}\right]$ leads to:

$$\left.\frac{\delta \mathcal{L}}{\delta f_{\hat{\mathbf{x}}}}\right|_{f_{\hat{\mathbf{x}}}(x) = f_{\hat{\mathbf{x}}}^{\text{opt}}(x)} = 0 \Longrightarrow -\frac{g(x)^2}{f_{\hat{\mathbf{x}}}^{\text{opt}}(x)^2} + \lambda = 0, \qquad (2.77)$$

or

$$f_{\hat{\mathbf{x}}}^{\text{opt}}(x) = +\lambda^{-1/2}\, |g(x)|\,. \qquad (2.78)$$

According to (2.74) we have taken the $+$ sign for the square root. Now λ is found from the normalization condition (2.75):

$$f_{\hat{\mathbf{x}}}^{\text{opt}}(x) = \frac{|g(x)|}{\int |g(x)|\, dx}. \qquad (2.79)$$

The corresponding optimal function $G^{\text{opt}}(x)$ is

$$G^{\text{opt}}(x) = \frac{g(x)}{f_{\hat{\mathbf{x}}}^{\text{opt}}(x)}, \qquad (2.80)$$

and the associated minimum variance is

$$\sigma^2[G^{\text{opt}}] = \left(\int |g(x)|\, dx\right)^2 - I^2. \qquad (2.81)$$

This result indicates that if $g(x)$ is a non-negative function, such that $|g(x)| = g(x)$, then the variance of the optimal estimator is 0. In other words, the statistical error is 0 and the sample average is exact (independently of the number of points M used). Where is the trick? If we look at the optimal choice, in the denominator of (2.79) appears precisely the integral I. But knowing the value of the integral was the initial goal. If we know it, why should we need a numerical algorithm whose goal is the calculation of the integral?

However, the idea of importance sampling is to replace the optimal value $f_{\hat{\mathbf{x}}}^{\text{opt}}(x)$ by some other function $f_{\hat{\mathbf{x}}}(x)$ close to it (while still keeping the generation of the random variable easy enough). For example, let us look at the calculation of the integral in (2.64). The optimal choice would be the splitting

$$G^{\text{opt}}(x) = \lambda^{1/2}\frac{\cos(x)}{|\cos(x)|}, \quad x \ge 0, \tag{2.82}$$

$$f_{\hat{\mathbf{x}}}^{\text{opt}}(x) = \begin{cases} 0, & x < 0, \\ \lambda^{-1/2}|\cos(x)|x^2 e^{-x}, & x \ge 0, \end{cases} \tag{2.83}$$

being $\lambda^{1/2} = \int_0^\infty |\cos(x)|x^2 e^{-x}dx$ the normalization constant[6]. The minimal variance of the optimal choice is $\sigma^2[G^{\text{opt}}] = \lambda - I^2 \approx 1.190009$. However, there is no simple way to solve $\int_0^x f_{\hat{\mathbf{x}}}^{\text{opt}}(x')dx' = u$ and the optimal choice is useless. We could use instead, for example, any of the four splittings given in (2.65)-(2.71). Which one would be the most efficient? The one that uses a probability density function $f_{\hat{\mathbf{x}}}(x)$ closer to $f_{\hat{\mathbf{x}}}^{\text{opt}}$. Intuitively, it is option number 3, as all it does is to replace $|\cos(x)|$ by an average value $1/2$, We can check this out, by computing (analytically) the variance in the four cases using the integrals of (2.73), with the result:

$$\begin{aligned}
\sigma^2[G^{(1)}] &= \frac{148843}{12500} \approx 11.907\\
\sigma^2[G^{(2)}] &= \frac{6791}{2500} \approx 2.716\\
\sigma^2[G^{(3)}] &= \frac{787}{500} \approx 1.574\\
\sigma^2[G^{(4)}] &= -\frac{1}{4} + \frac{27\pi}{16} \approx 5.051
\end{aligned}$$

Hence, the splitting number 3, given by (2.69) is indeed the most efficient, at least from the point of view of the associated variance. Of course, we need to check that the generation of the random numbers distributed according to $f_{\hat{\mathbf{x}}}^{(3)}(x)$, a $\hat{\mathbf{\Gamma}}(3,1)$ distribution is not so slow as to render the method more inefficient than the others. We will return to this point later in the chapter.

Once we have determined which is the optimal splitting, all we need to do is to use subroutine `mc3` with a function $G^{(3)}(x) = 2\cos(x)$ and `ran_gamma3` for the generation of random numbers distributed according to the $\hat{\mathbf{\Gamma}}(3,1)$ distribution. For the sake of clarity we include below a driver program as well as an implementation of the function $G^{(3)}(x)$ which together with subroutine `mc3` and function `ran_gamma3` leads to a full program.

6) The integral can be performed analytically to obtain
$\lambda^{1/2} = \frac{2-6e^{\pi}+6e^{2\pi}-2e^{3\pi}+e^{\pi/2}(-2+\pi)^2+e^{5\pi/2}(2+\pi)^2+e^{3\pi/2}(-8+6\pi^2)}{4(-1+e^{\pi})^3} \approx 1.20003565\ldots.$

```
program area2
  implicit none
  interface
    double precision function g3(x)
      double precision, intent (in) :: x
    end function g3
  end interface
  interface
    double precision function ran_gamma3()
    end function ran_gamma3
  end interface
  double precision :: r,s
  integer :: M
  M=100000
  call mc3(g3,ran_gamma3,M,r,s)
  write (*,*) "The estimated value of the integral is", r
  write (*,*) "The estimated error is", s
end program area2

double precision function g3(x)
  implicit none
  double precision, intent (in) ::x
  g3=2.d0*cos(x)
end function g3
```

The reader should run this program with a reasonable large number, e.g. $M = 10^5$, and check that the numerical result agrees, within errors, with the known value $I = -1/2$.

Summing up, the basic idea of the importance sampling method is to split the integrand as $g(x) = G(x)f_{\hat{\mathbf{x}}}(x)$ using a suitable random variable $\hat{\mathbf{x}}$ with pdf $f_{\hat{\mathbf{x}}}(x)$. It is desirable to chose $f_{\hat{\mathbf{x}}}(x)$ in such a way that (i) it is reasonably simple to generate, (ii) it gives a small variance. The optimal $f_{\hat{\mathbf{x}}}(x)$ most often does not satisfy condition (i), but we might then look for functions $f_{\hat{\mathbf{x}}}(x)$ which are close enough to the optimal one. For example, let us consider the integral of $g(x) = J_0(x_0 x)$,

$$I = \int_0^1 dx J_0(x_0 x) \tag{2.84}$$

being J_0 the Bessel function and $x_0 = 2.404825557695773\ldots$ the first zero of $J_0(x)$. Looking at the shape of J_0 we propose the family of pdf's:

$$f_{\hat{\mathbf{x}}}(x) = \frac{6}{3+a}\left(-ax^2 + (a-1)x + 1\right), \qquad x \in (0,1) \tag{2.85}$$

i.e. a properly normalized family of parabolic functions that have a maximum at $x = 0$ if $a < 1$, and vanish at $x = 1$. Positivity is ensured if $a \geq -1$. Therefore a is restricted to the interval $(-1, 1)$. We define, hence:

$$G(x) = \frac{g(x)}{f_{\hat{\mathbf{x}}}(x)} = \frac{3+a}{6}\frac{J_0(x_0 x)}{(-ax^2 + (a-1)x + 1)} \tag{2.86}$$

and use the sample mean of $G(x)$ as an unbiased estimator for the integral. To generate random numbers distributed according to $f_{\hat{\mathbf{x}}}(x)$ we need to invert the cumulative distribution function:

$$F_{\hat{\mathbf{x}}}(x) = \frac{6}{3+a}\left(-\frac{a}{3}x^3 + \frac{a-1}{2}x^2 + x\right), \qquad x \in (0,1) \tag{2.87}$$

Although in this case Cardano's formula gives us the solution of the cubic equation $F_{\hat{\mathbf{x}}}(x) = u$, it is actually easier to implement again the Newton-Raphson algorithm. The following program returns random numbers distributed according to $f_{\hat{\mathbf{x}}}(x)$.

```
double precision function ran_poli2()
  implicit none
  double precision :: a,x,xn,fx,fxc,u,ran_u
  common /a/a
```

```
    fx(x)=6.0/(3.0+a)*(1.0+x*(a-1.0-a*x))
    fxc(x)=6.0/(3.0+a)*x*(1.0+x*((a-1.0)/2-a/3.0*x))
    u=ran_u()
    x=u
    xn=x-(fxc(x)-u)/fx(x)
    do while(abs(xn-x).gt.1.0d-8)
      xn=x
      x=x-(fxc(x)-u)/fx(x)
    enddo
    ran_poli2=x
```

```
end function ran_poli2
```

We give here an example of a driver and an implementation of $G(x)$:

```
program area3
  implicit none
  interface
    double precision function gb(x)
      double precision, intent (in) :: x
    end function gb
  end interface
  interface
    double precision function ran_poli2()
    end function ran_poli2
  end interface
  double precision :: a,r,s
  integer :: M
  common /a/a
  M=1000000
  a=0.5d0
  call mc3(gb,ran_poli2,M,r,s)
  write (*,*) "The estimated value of the integral is", r
  write (*,*) "The estimated error is", s
end program area3

double precision function gb(x)
  implicit none
  double precision, intent (in) :: x
  double precision, parameter :: x0=2.404825557695773d0
  double precision :: a,fx,dbesj0
```

```
    common /a/a
    fx(x)=6.d0/(3.d0+a)*(1.d0+x*(a-1.d0-a*x))
    gb=dbesj0(x*x0)/fx(x)
  end function gb
```

This has to be complemented with the function `ran_poli2()` and the subroutine `mc3`. Here `dbesj0` is the Bessel function $J_0(x)$ as implemented in gfortran and also in the Intel fortran compiler. Otherwise a routine that returns the value of the Bessel function $J_0(x)$ must be provided. Running this program for different values of a it is found that the minimal error is obtained for $a \approx 0.4$. From a particular run, we have obtained [7] $I = 0.611411 \pm 0.000024$. We do not have an exact value to compare with now, but a numerical integration using more traditional techniques[8] yields $I = 0.6113957$. The difference is 1.5×10^{-5}, in perfect agreement with the importance sampling estimate, including the error.

2.6 Advantages of Monte Carlo integration

The reader might be somehow disappointed as the examples we have given so far could be done either analytically, e.g. (2.64), or by other numerical methods with more precision, case of (2.84). Although there might be examples of one-dimensional integrals in which the Monte Carlo integration could be competitive compared to more traditional methods (like Simpson integration), the truth is that the real power of Monte Carlo integration lies in the calculation of N-dimensional integrals. Let us consider, for example, the integral:

$$I(N) = \int_{-\infty}^{\infty} dx_1 \cdots \int_{-\infty}^{\infty} dx_N e^{-(x_1+\cdots+x_N)} J_0\left(x_1^2+\cdots+x_N^2\right). \tag{2.88}$$

For large N it would be difficult to write an efficient code implementing, for instance, a Simpson-like algorithm. Let us consider it from the point of view of importance sampling. We split the integrand $g(x_1,\dots,x_N) = G(x_1,\dots,x_N) f_{\hat{\mathbf{x}}_1,\dots,\hat{\mathbf{x}}_n}(x_1,\dots,x_N)$ with:

$$\begin{aligned} G(x_1,\dots,x_N) &= J_0\left(x_1^2+\cdots+x_N^2\right), & (2.89)\\ f_{\hat{\mathbf{x}}_1,\dots,\hat{\mathbf{x}}_n}(x_1,\dots,x_N) &= e^{-(x_1+\cdots+x_N)} = e^{-x_1}\cdots e^{-x_N}. & (2.90)\end{aligned}$$

The key point is that $f_{\hat{\mathbf{x}}_1,\dots,\hat{\mathbf{x}}_n}(x_1,\dots,x_N)$ can be considered as the probability density function of an N-dimensional random variable $\hat{\mathbf{x}} = (\hat{\mathbf{x}}_1,\dots,\hat{\mathbf{x}}_N)$. In fact, as it can be factorized as $f_{\hat{\mathbf{x}}_1,\dots,\hat{\mathbf{x}}_n}(x_1,\dots,x_N) = f_{\hat{\mathbf{x}}}(x_1)\cdots f_{\hat{\mathbf{x}}}(x_N)$ with $f_{\hat{\mathbf{x}}}(x) = e^{-x}$, the N random variables $\hat{\mathbf{x}}_1,\dots,\hat{\mathbf{x}}_N$ are independent of each other

7) Needless to say, running the program with different random number generators will produce different results. This value is just an example of one output of this program, not the one the reader will obtain when running the algorithm by himself. This comment applies to all the cases in which we give a numerical estimate of an integration using these any of these Monte Carlo methods.

8) Actually, it is the result given by the Mathematica program.

and follow the same exponential distribution. It is the very easy to use the method of importance sampling to compute (2.88): Generate M values $\mathbf{x}_{(1)}, \dots, \mathbf{x}_{(M)}$ of the N-dimensional vector $\mathbf{x} = (x_1, \dots, x_N)$, and use them to compute the sample mean and variance of $G(x)$, using the known formulas (2.11-2.12) and approximate the integral by (2.13).

Let us present a simple computer program:

```
program n_dimensional_integral
  implicit none
  interface
    double precision function ran_exp(a)
      double precision, intent (in) :: a
    end function ran_exp
  end interface
  double precision :: r,s,x,dbesj0,sx,s2,g0
  integer, parameter :: N=4
  integer ::M,i,k
  M=1000000
    r=0.d0
    s=0.d0
    do k=1,M
      sx=0.d0
      s2=0.d0
      do i=1,N
        x=ran_exp(1.d0)
        sx=sx+x
        s2=s2+x*x
      enddo
      g0=dbesj0(s2)
      r=r+g0
      s=s+g0*g0
    enddo
    r=r/M
    s=sqrt((s/M-r*r)/M)
  write (*,*) "The estimated value of the integral is", r
  write (*,*) "The estimated error is", s
end program n_dimensional_integral
```

As an example, we have ran this program with $M = 10^6$ samples and it took us a fraction of a second in a desktop computer to obtain the values $I(2) = 0.38596 \pm 0.00049$, $I(3) = 0.20028 \pm 0.00044$, $I(4) = 0.08920 \pm 0.00036$. These are in agreement with what you can get with other numerical methods, but are much faster to obtain. The complexity of the Monte Carlo algorithm does not increase with N, the number of integration variables. The program runs perfectly for $N = 10$ giving, using $M = 10^8$, $I(10) = -0.002728 \pm 0.000016$ in less than one minute, whereas it would extremely costly to get the same accuracy with a deterministic integration algorithm.

2.7 Monte Carlo importance sampling for sums

The same ideas of the different Monte Carlo integration techniques can be used in the case of sums. Imagine we want to compute a sum $\sum_i g_i$ and that it is possible to split $g_i = G_i p_i$, in such a way that $p_i \geq 0$ and $\sum_i p_i = 1$. Then the sum can be considered as the average value of a random variable $\hat{\mathbf{x}}$ that takes only integer values, such that its pdf is:

$$f_{\hat{\mathbf{x}}}(x) = \sum_i p_i \delta(x - i). \tag{2.91}$$

We can then consider this sum as the average value $E[G]$ and evaluate it using the sample mean. In this case this means to generate integer values $i_1, \dots, i_M$ distributed with the set of probabilities p_i and then compute the sample mean and variance:

$$\hat{\mu}_M[G] = \frac{1}{M}\sum_{k=1}^{M} G_{i_k} \tag{2.92}$$

$$\hat{\sigma}^2_M[G] = \frac{1}{M}\sum_{k=1}^{M} G^2_{i_k} - \left(\frac{1}{M}\sum_{k=1}^{M} G_{i_k}\right)^2. \tag{2.93}$$

For example, if we want to compute the sum

$$S = \sum_{i=0}^{\infty} i^{1/2} 2^{-i} \tag{2.94}$$

(the exact value is PolyLog $\left[-\frac{1}{2}, \frac{1}{2}\right] = 1.347253753\dots$), we split $g_i = G_i p_i$ with $G_i = 2i^{1/2}$ and $p_i = \frac{1}{2}2^{-i}$. Then the p_i's are the probabilities of a discrete geometric distribution of parameter $q = 1/2$ and we already learned how to generate those. A full program is:

```
program monte_carlo_sum
  implicit none
  interface
    integer function iran_geo(a)
       double precision, intent (in) :: a
    end function ran_exp
  end interface
  double precision :: q,r,s,a,g0
  integer :: M,i,k
  p=0.5d0
  M=1000000
```

```
    r=0.d0
    s=0.d0
    a=-log(q)
    do k=1,M
      i=iran_geo(p)
      g0=2.d0*sqrt(dble(i))
      r=r+g0
      s=s+g0*g0
    enddo
    r=r/M
    s=sqrt((s/M-r*r)/M)
  write (*,*) "The estimated value of the sum is", r
  write (*,*) "The estimated error is", s
end program monte_carlo_sum
```

A particular run of this program with $M = 10^8$ gave us $S = 1.34714 \pm 0.00015$, in perfect agreement, within statistical errors, with the exact result. Again, the real advantage of the Monte Carlo integration lies in the numerical calculations of high-dimensional sums. For example, the sum:

$$S(N) = \sum_{i_1=0}^{\infty} \cdots \sum_{i_N=0}^{\infty} \frac{2^{-(i_1+\cdots+i_N)}}{1+i_1^2+\cdots+i_N^2} \tag{2.95}$$

can be evaluated splitting $g_{i_1\cdots i_N} = G_{i_1\cdots i_N} p_{i_1\cdots i_N}$, with $G_{i_1\cdots i_N} = \frac{2^N}{1+i_1^2+\cdots+i_N^2}$ and $p_{i_1\cdots i_N} = \left(\frac{1}{2}2^{-i_1}\right)\cdots\left(\frac{1}{2}2^{-i_N}\right)$, the product of N independent geometric distributions. The program is:

```
program monte_carlo_multidimensional_sum
  implicit none
  interface
    integer function iran_geo(a)
       double precision, intent (in) :: a
    end function ran_exp
  end interface
  integer, parameter :: N=4
  double precision :: q,r,s,sx,a,g0
  integer :: M,i,k,j
  p=0.5d0
  M=100000000
  r=0.d0
  s=0.d0
  a=-log(q)
  do k=1,M
    sx=1.0d0
    do i=1,N
      j=iran_geo(p)
      sx=sx+j*j
    enddo
    g0=2.0d0**N/sx
    r=r+g0
    s=s+g0*g0
  enddo
  r=r/M
  s=sqrt((s/M-r*r)/M)
  write (*,*) "The estimated value of the sum is", r
  write (*,*) "The estimated error is", s
end program monte_carlo_multidimensional_sum
```

A particular run with $M = 10^8$ gave the following results $S(1) = 1.318107 \pm 0.000073$, $S(2) = 1.79352 \pm 0.00014$, $S(4) = 3.67076 \pm 0.00040$, $S(10) = 63.5854 \pm 0.0071$. While the first three are in agreement with other numerical methods to evaluate sums, the last result, for $N = 10$, which requires less than one minute in a desktop computer, would be difficult to evaluate using any other method.

2.8 Efficiency of an integration method

This is a concept very easy to understand. The efficiency of an integration method is measured by the time it takes a computer to provide an estimate of the integral with some given error ϵ. In the Monte Carlo methods explained so far, the error is always given by

$$\epsilon = \frac{\sigma}{\sqrt{M}}, \tag{2.96}$$

being σ, the root-mean-square, an intrinsic value independent of the number of repetitions M. If we fix the error ϵ, it turns out that $M = \sigma^2/\epsilon^2$ is proportional to the variance of the estimator, σ^2. At first sight, then, it seems natural to use an estimator with the smallest possible variance (i.e. the importance sampling), but there is another factor to take into consideration: the actual time (in seconds) t that it takes to generate each contribution to the estimator. This consists in the time spent in every call to the function `ran_f()` plus the time needed to compute the function $G(x)$ and do the sums contributing to the average and the variance. The total needed time is then Mt which is proportional to $t\sigma^2$. Hence, it might pay off to use a method with a larger variance, if the time needed to generate each contribution to the sample mean compensates the larger variance. The relative efficiency between integration methods 1 and 2 is nothing but the ratio

$$e_{12} = \frac{t_1\sigma_1^2}{t_2\sigma_2^2}, \tag{2.97}$$

being t_1 and σ_1^2 the time and variance of method 1 and similarly for method 2. If $e_{12} > 1$ we conclude that method 2 is more efficient than method 1.

We can prove, for instance, that the method of uniform sampling is more efficient than the hit and miss method. For the hit and miss method, recalling (2.4)-(2.5), we get that the variance of this method can be written as:

$$\sigma_1^2 = c(b-a)I - I^2. \tag{2.98}$$

While for the uniform sampling method, the variance is:

$$\sigma_2^2 = (b-a)\int g(x)^2\,dx - I^2. \tag{2.99}$$

So that

$$\sigma_1^2 - \sigma_2^2 = (b-a)\left[cI - \int g(x)^2\,dx\right]. \tag{2.100}$$

Since we had the condition $0 \le g(x) \le c$ we obtain

$$\int g(x)^2\,dx \le c\int g(x)\,dx = cI \tag{2.101}$$

and $\sigma_1 \ge \sigma_2$. Furthermore, given that the hit and miss method requires the calculation of two random numbers, whereas the uniform sampling requires only one, it turns out that it takes more time, $t_1 > t_2$. As $e_{12} > 1$, we conclude that uniform sampling is more efficient than hit and miss.

2.9 Summary

In this chapter we have argued that numerical integration based on the sampling algorithm:

$$\int G(x)f_{\hat{\mathbf{x}}}(x)\,dx = \hat{\mu}_M[G] \pm \frac{\hat{\sigma}_M[G]}{\sqrt{M}} \tag{2.102}$$

with

$$\hat{\mu}_M[G] = \frac{1}{M}\sum_{k=1}^{M} G(\mathbf{x}_k), \tag{2.103}$$

$$\hat{\sigma}_M^2[G] = \frac{1}{M}\sum_{k=1}^{M} G(\mathbf{x}_k)^2 - \left(\frac{1}{M}\sum_{k=1}^{M} G(\mathbf{x}_k)\right)^2, \tag{2.104}$$

being $\mathbf{x}_k, k = 1,\dots,M$ values of the random variable $\hat{\mathbf{x}}$ whose pdf is $f_{\hat{\mathbf{x}}}(x)$, can be very competitive if $x = (x_1,\dots,x_N)$ is high-dimensional. How high is high? How large must N be for this method to be competitive? The exact answer might depend on the specific details of the function $G(x)$ and the difficulty in the generation of the random variables. The answer might be $N > 10$ or $N > 5$, but one thing is sure: when N is very large, of the order of thousands or millions of variables involved in the integral, then **there is no alternative to the Monte Carlo sampling**. There are many problems that need of the calculation of such high dimensional integrals, but the typical applications are to the field of statistical mechanics, where ideally one would like to deal with N of the order of the Avogadro number, $N \sim 10^{23}$, although we are very far away from being able to get close to this number with today's (or tomorrow's, for that matter) computer capabilities. We have reached a stage, however, where it is not unusual to consider that N is in the range of 10^6, although much larger values can be dealt with satisfactorily in some specific cases.

It is clear that, to proceed, we need efficient ways of generating the values of the N-dimensional random variable $\hat{\mathbf{x}} = (\hat{\mathbf{x}}_1, \dots, \hat{\mathbf{x}}_N)$ whose pdf is $f_{\hat{\mathbf{x}}_1,\dots,\hat{\mathbf{x}}_n}(x_1, \dots, x_N)$. In the few examples of this chapter in which we have used the sampling method for N-dimensional integrals or sums, we have managed to split $f_{\hat{\mathbf{x}}_1,\dots,\hat{\mathbf{x}}_n}(x_1, \dots, x_N) = f_{\hat{\mathbf{x}}}(x_1) \dots f_{\hat{\mathbf{x}}}(x_N)$ as the product of N identical and independent random variables. Unfortunately, this it not the case of the majority of applications of interest and we must develop methods to generate those high-dimensional random variables. Before we dwell into this, we will explain further algorithms valid for one variable pdf's and then move on to high-dimensional variables.

Further reading

The basic Monte Carlo integration techniques, including hit and miss, sampling methods and other not explained here can be found in the classic books by Kalos and Whitlock[3] and Rubinstein[4].

Exercises

1) Compute numerically using the hit and miss Monte Carlo method, the integral

$$\int_0^1 dx\sqrt{1-x^2}$$

Compute the dependence of the root–mean-square σ of the Monte Carlo estimator with the number of points used (take M=10,100,1000, etc.). Calculate the integral analytically and plot in a log-log scale the real error (absolute difference between in exact value and the Monte Carlo estimator) versus N. Which is the slope of the curve as $M \to \infty$?

2) Repeat the previous problem using the uniform sampling method. Which is the relation between the rms of both methods? Measure the time it takes to run both algorithms and find which method is more efficient. How many CPU seconds will it take to compute the integral with an error less that 10^{-6}?

3) Repeat the two previous problems 10^4 times with $M = 100$ using each time a different sequence of random numbers. Compute, for each method, the percentage of cases for which the numerical result differs from the exact one in less that (a) σ, (b) 2σ, (c) 3σ.

4) Compute the integral of the previous exercises using the method of uniform sampling with $M = 10^4$. Compute the rms of the result. Let μ_1 and σ_1 be, respectively, the estimator and the rms obtained. Repeat 10 times using each a different sequence of random numbers and obtain 10 different estimators $\mu_1, \dots, \mu_{10}$ and rms $\sigma_1, \dots, \sigma_{10}$. Compute a new estimator μ and rms σ from the average of the 10 values I_i and their rms. Which relation do you expect between I_i, σ_i, I and σ? Check that relation.

5) Using the pdf $f_{\hat{\mathbf{x}}}(x) = \exp(-x)$, $x \geq 0$ compute using the sampling method the integral:

$$\int_0^\infty dx\sqrt{x}\cos(x)\exp(-x)$$

6) Compute the integral

$$I = \int_0^1 dx \cos\left(\frac{\pi x}{2}\right)$$

using uniform sampling and a general sampling method with the pdf $f_{\hat{\mathbf{x}}}(x) = \frac{3}{2}(1-x^2)$. Which is the relation between the rms of both methods? Which is their relative efficiency?

7) Compute the integral of the previous problems using the sampling method with the pdf $f_{\hat{\mathbf{x}}}(x) = \frac{6}{3+a}[1 + x(a-1) - ax^2]$. Compute numerically the rms of the result as a function of a and determine the optimal value of a. Compare the efficiency for $a = 1$ and the optimal value.

8) Compute using the sampling method the integral of problem 1 using the pdf $f_{\hat{\mathbf{x}}}(x) = \frac{1-ax^2}{1-a/3}$ depending on the parameter a and compute, numerically, the optimal value for a.

9) For problem 1 check that when using the optimal pdf $f_{\hat{\mathbf{x}}}^{\text{opt}}(x)$ as given by the importance sampling method, then the sampling error is zero whenever the number of points used for the numerical integration.
10) Compute

$$\int_0^1 dx_1 \ldots \int_0^1 dx_n \exp\left[-\frac{1}{2}(x_1^2 + x_2^2 + \cdots + x_n^2)\right] \cos^2(x_1x_2+x_2x_3+\cdots+x_nx_1)$$

for $n = 1, 2, 3, 5, 10$ using Simpson's, uniform sampling and hit and miss methods.
11) Prove, without using variational methods, that the function $f_{\hat{\mathbf{x}}}^{\text{opt}}$ is the one that minimizes the rms of the numerical estimator to the integral I. Show that this equivalent to prove:

$$\left(\int |g(x)|\, dx\right)^2 \leq \int \frac{g(x)^2}{f_{\hat{\mathbf{x}}}(x)}\, dx$$

and this follows from Schwartz's inequality

$$\left(\int f_1(x) f_2(x)\, dx\right)^2 \leq \int f_1(x)^2\, dx \int f_2(x)^2\, dx$$