2

Random Variable Generation

"It has long been an axiom of mine that the little things are infinitely the most important."

Arthur Conan Doyle
A Case of Identity

Reader's guide

In this chapter, we present practical techniques that can produce random variables from both standard and nonstandard distributions by using a computer program. Given the availability of a uniform generator in `R`, as explained in Section 2.1.1, we do not deal with the specific production of uniform random variables. The most basic techniques relate the distribution to be simulated to a uniform variate by a transform or a particular probabilistic property, as in Section 2.2, while the most generic one is a simulation version of the trial-and-error method, described in Section 2.3 under the name of the Accept–Reject method. In all cases, the methods rely on the availability of sequences of independent uniform generations that are provided by the resident `R` generator, `runif`.

C.P. Robert, G. Casella, *Introducing Monte Carlo Methods with R*, Use R,
DOI 10.1007/978-1-4419-1576-4_2, © Springer Science+Business Media, LLC 2010

2.1 Introduction

The methods developed in this book and summarized under the denomination of *Monte Carlo methods* mostly rely on the possibility of producing (with a computer) a supposedly endless flow of random variables for well-known or new distributions. Such a simulation is, in turn, based on the production of uniform random variables on the interval $(0, 1)$. Although we are not directly concerned with the *mechanics* of producing such uniform random variables, because existing uniform generators can be considered as "perfect", we will completely rely on those generators to produce other random variables. In a sense, the uniform distribution $\mathcal{U}_{[0,1]}$ provides the basic probabilistic representation of randomness on a computer and the generators for all other distributions do require a sequence of uniform variables to be simulated.

As already pointed out in Section 1.4 of Chapter 1, `R` has a large number of built-in functions that will generate the standard random variables listed in Table 1.1. For instance,

```
> rgamma(3,2.5,4.5)
```

produces three independent generations from a $\mathcal{G}(5/2, 9/2)$ distribution with all due guarantees of representing this distribution. It is therefore counterproductive, inefficient, and even dangerous to generate from those standard distributions using anything but the resident `R` generators. The principles developed in the following sections are, however, essential to deal with less standard distributions that are not built into `R`.

2.1.1 Uniform simulation

The basic uniform generator in `R` is the function `runif`, whose only required entry is the number of values to be generated. The other optional parameters are `min` and `max`, which characterize the bounds of the interval supporting the uniform. (The default is `min=0` and `max=1`.) For instance,

```
> runif(100, min=2, max=5)
```

will produce 100 random variables distributed uniformly between 2 and 5.

Strictly speaking, all the methods we will see (and this includes `runif`) produce *pseudo-random numbers* in that there is no randomness involved—based on an initial value u_0 of a uniform $\mathcal{U}(0, 1)$ sequence and a transformation D, the uniform generator produces a sequence $(u_i) = (D^i(u_0))$ of values in $(0, 1)$—but the outcome has the same *statistical properties* as an iid sequence. Further details on the random generator of `R` are provided in the on-line help on `RNG`.

While extensive testing of this function has been undertaken to make sure it does produce uniform variates for all purposes (see, e.g., Robert and Casella,

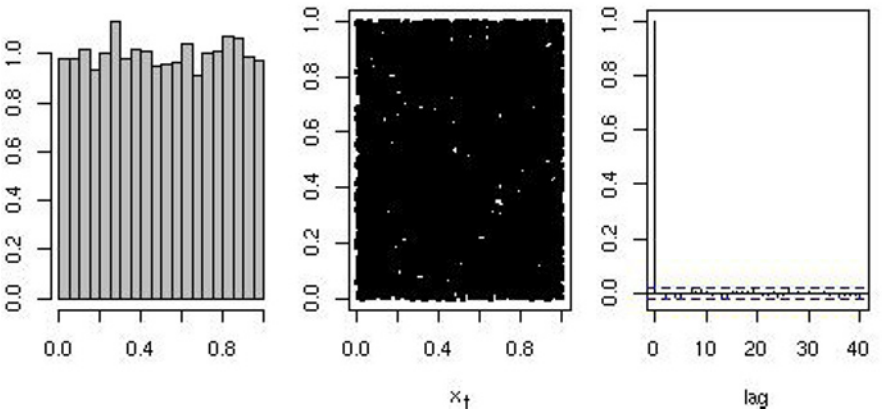

Fig. 2.1. Histogram *(left)*, pairwise plot *(center)*, and estimated autocorrelation function *(right)* of a sequence of 10^4 uniform random numbers generated by runif.

2004, Chapter 2), a quick check on the properties of this uniform generator is to look at an histogram of the X_i's, a plot of the pairs (X_i, X_{i+1}), and the estimated autocorrelation function, as any random variable generator does suffer from a residual autocorrelation and good algorithms will reduce this to a negligible value. The R code used to produce the output in Figure 2.1 is

```
> Nsim=10^4              #number of random numbers
> x=runif(Nsim)
> x1=x[-Nsim]            #vectors to plot
> x2=x[-1]               #adjacent pairs
> par(mfrow=c(1,3))
> hist(x)
> plot(x1,x2)
> acf(x)
```

and shows that `runif` is apparently acceptable for this casual evaluation.

As pointed out in the previous remark, `runif` does not involve randomness per se. Producing `runif(Nsim)` is better described as a deterministic sequence based on a random starting point. An extreme illustration of this fact is obtained through the R function `set.seed`, which uses its single integer argument to set as many seeds as required. For instance,

```
> set.seed(1)
> runif(5)
[1] 0.2655087 0.3721239 0.5728534 0.9082078 0.2016819
> set.seed(1)
> runif(5)
```

```
[1] 0.2655087 0.3721239 0.5728534 0.9082078 0.2016819
> set.seed(2)
> runif(5)
[1] 0.0693609 0.8177752 0.9426217 0.2693818 0.1693481
```

shows that setting the seed determines all the subsequent values produced by the random generator. In the overwhelming majority of cases, we do not set the seed, which is then chosen according to the current time. But in settings where we need to reproduce the exact same sequence of random simulations, for example to compare two procedures or two speeds, setting a fixed value of the seed is reasonable.

2.1.2 The inverse transform

There is a simple, sometimes useful transformation, known as the *probability integral transform,* that allows us to transform any random variable into a uniform random variable and, more importantly, vice versa. For example, if X has density f and cdf F, then we have the relation

$$F(x) = \int_{-\infty}^{x} f(t)\,\mathrm{d}t,$$

and if we set $U = F(X)$, then U is a random variable distributed from a uniform $\mathcal{U}(0,1)$. This is because

$$P(U \le u) = P[F(X) \le F(x)] = P[F^{-1}(F(X)) \le F^{-1}(F(x))] = P(X \le x),$$

where we have assumed that F has an inverse. This assumption can be relaxed (see Robert and Casella, 2004, Section 2.1) but holds for most continuous distributions.

Exercise 2.1 For an arbitrary random variable X with cdf F, define the generalized inverse of F by

$$F^{-}(u) = \inf\{x;\, F(x) \ge u\}.$$

Show that if $U \sim \mathcal{U}(0,1)$, then $F^{-}(U)$ is distributed like X.

Example 2.1. If $X \sim \mathcal{E}xp(1)$, then $F(x) = 1 - e^{-x}$. Solving for x in $u = 1 - e^{-x}$ gives $x = -\log(1-u)$. Therefore, if $U \sim \mathcal{U}_{[0,1]}$, then

$$X = -\log U \sim \mathcal{E}xp(1)$$

(as U and $1 - U$ are both uniform). The corresponding `R` code

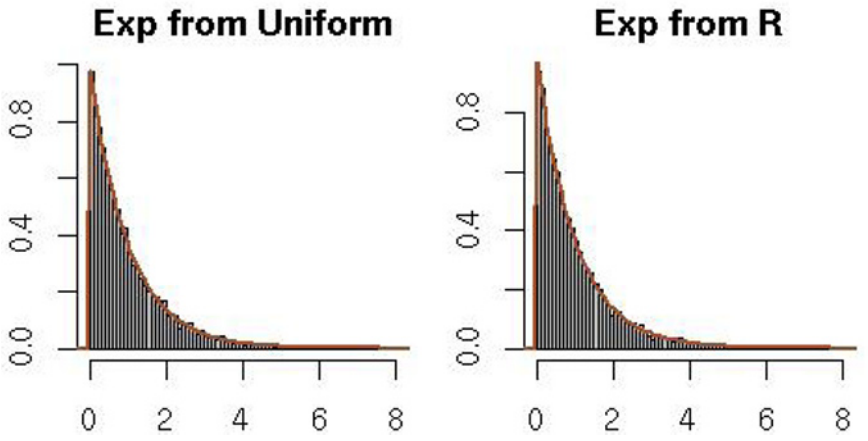

Fig. 2.2. Histograms of exponential random variables using the inverse transform *(right)* and using the `R` command `rexp` *(left)*, with the $\mathcal{E}xp(1)$ density on top.

```
> Nsim=10^4              #number of random variables
> U=runif(Nsim)
> X=-log(U)              #transforms of uniforms
> Y=rexp(Nsim)           #exponentials from R
> par(mfrow=c(1,2))      #plots
> hist(X,freq=F,main="Exp from Uniform")
> hist(Y,freq=F,main="Exp from R")
```

compares the output from the probability inverse transform with the output from `rexp`. The fits of both histograms to their exponential limit are not distinguishable in Figure 2.2. ◄

The generation of uniform random variables is therefore a key determinant of the behavior of simulation methods for other probability distributions since those distributions can be represented as a deterministic transformation of uniform random variables.

Exercise 2.2 Two distributions that have explicit forms of the cdf are the logistic and Cauchy distributions. Thus, they are well-suited to the inverse transform method. For each of the following, verify the form of the cdf and then generate 10,000 random variables using the inverse transform. Compare your program with the built-in `R` functions `rlogis` and `rcauchy`, respectively:

a. Logistic pdf: $f(x) = \frac{1}{\beta} \frac{e^{-(x-\mu)/\beta}}{[1+e^{-(x-\mu)/\beta}]^2}$, cdf: $F(x) = \frac{1}{1+e^{-(x-\mu)/\beta}}$.
b. Cauchy pdf: $f(x) = \frac{1}{\pi\sigma} \frac{1}{1+\left(\frac{x-\mu}{\sigma}\right)^2}$, cdf: $F(x) = \frac{1}{2} + \frac{1}{\pi}\arctan((x-\mu)/\sigma)$.

2.2 General transformation methods

When a distribution with density f is linked in a relatively simple way to another distribution that is easy to simulate, this relationship can often be exploited to construct an algorithm to simulate variables from f.

Example 2.2. In Example 2.1, we saw how to generate an exponential random variable starting from a uniform. Now we illustrate some of the random variables that can be generated starting from an exponential distribution. If the X_i's are iid $\mathcal{E}xp(1)$ random variables, then three standard distributions can be derived as

$$
\begin{aligned}
Y &= 2\sum_{j=1}^{\nu} X_j \sim \chi^2_{2\nu}\,, \qquad \nu \in \mathbb{N}^*\,,\\
Y &= \beta\sum_{j=1}^{a} X_j \sim \mathcal{G}(a,\beta)\,, \qquad a \in \mathbb{N}^*\,,\\
Y &= \frac{\sum_{j=1}^{a} X_j}{\sum_{j=1}^{a+b} X_j} \sim \mathcal{B}e(a,b)\,, \qquad a,b \in \mathbb{N}^*\,,
\end{aligned}
\tag{2.1}
$$

where $\mathbb{N}^* = \{1,2,\ldots\}$. For example, to generate χ^2_6 random variables, we could use the `R` code

```
> U=runif(3*10^4)
> U=matrix(data=U,nrow=3) #matrix for sums
> X=-log(U)               #uniform to exponential
> X=2* apply(X,2,sum)     #sum up to get chi squares
```

Obviously, this is not nearly as efficient as calling `rchisq`, as can be checked by the `R` code

```
> system.time(test1());system.time(test2())
   user  system elapsed
  0.104   0.000   0.107
   user  system elapsed
  0.004   0.000   0.004
```

where `test1` corresponds to the `R` code above and `test2` to its substitution by `X=rchisq(10^4,df=6)`. ◀

Many other derivations of standard distributions are possible when taking advantage of existing probabilistic properties, as shown in Exercise 2.12.

ϟ These transformations are quite simple to use and hence will often be a favorite in our illustrations. However, there are limits to their usefulness, both in the scope of variables that can be generated that way (think, for instance, of a chi-squared distribution with a noneven number of degrees of freedom) and efficiency of generation. For any specific distribution,

efficient algorithms have been developed. Thus, if R has a distribution built in, it is almost always worth using, as shown by Example 2.2. Moreover, the transformation method described above cannot reach all distributions; for example, we cannot get a standard normal.

2.2.1 A normal generator

One way to achieve normal random variable simulation using a transform is with the Box–Muller algorithm, devised for the generation of $\mathcal{N}(0,1)$ variables.

Example 2.3. If U_1 and U_2 are iid $\mathcal{U}_{[0,1]}$, the variables X_1 and X_2 defined by

$$X_1 = \sqrt{-2\log(U_1)}\,\cos(2\pi U_2)\,, \qquad X_2 = \sqrt{-2\log(U_1)}\,\sin(2\pi U_2)\,,$$

are then iid $\mathcal{N}(0,1)$ by virtue of a simple change of variable argument. Note that this is *not* the generator implemented in R, which uses by default the probability inverse transform, based on a very accurate representation of the normal cdf inverse qnorm (up to 16 digits!). (It is, however, possible, if not recommended, to switch the normal generator to the Box–Muller (or even to the Kinderman-Ramage) version via the RNG function.) ◀

In comparison with (crudely) approximative algorithms based on the Central Limit Theorem (CLT), the Box–Muller algorithm is exact, producing two normal random variables from two uniform random variables, the only drawback (in speed) being the necessity of calculating transcendental functions such as log, cos, and sin.

Exercise 2.3 An antiquated generator for the normal distribution is:
Generate $U_1, \ldots, U_{12} \sim \mathcal{U}[-1/2, 1/2]$
Set $Z = \sum_{i=1}^{12} U_i$
the argument being that the CLT normality is sufficiently accurate with 12 terms.

a. Show that $\mathbb{E}[Z] = 0$ and $\text{var}(Z) = 1$.
b. Using histograms, compare this CLT-normal generator with the Box–Muller algorithm. Pay particular attention to tail probabilities.
c. Compare both of the generators in part a. with rnorm.

Note that this exercise does not suggest *using* the CLT for normal generations! This is a very poor approximation indeed.

The simulation of a multivariate normal distribution $\mathcal{N}_p(\mu, \Sigma)$, where Σ is a $p \times p$ symmetric and positive-definite matrix, can be derived from the generic rnorm generator in that using a Cholesky decomposition of Σ (that is, $\Sigma = AA^{\mathrm{T}}$) and taking the transform by A of an iid normal vector of dimension p leads to a $\mathcal{N}_p(0, \Sigma)$ normal vector. There is, however, an R package

that replicates those steps, called `rmnorm` and available from the `mnormt` library (Genz and Azzalini, 2009). This library also allows computation of the probability of hypercubes via the function `sadmvn`, as in

```
> sadmvn(low=c(1,2,3),upp=c(10,11,12),mean=rep(0,3),var=B)
[1] 9.012408e-05
attr(,"error")
[1] 1.729111e-08
```

where `B` is a positive-definite matrix. This is quite useful since the analytic derivation of this probability is almost always impossible.

Exercise 2.4 Given a 3×3 matrix `Sigma`:

a. Show that `Sigma=cov(matrix(rnorm(30),nrow=10))` defines a proper covariance matrix.
b. Show that setting `A=t(chol(Sigma))` leads to a simulation from $\mathcal{N}_p(0, \Sigma)$ by using the command `x=A%*%rnorm(3)`.
c. Compare the execution times of this approach and `rmnorm` when simulating one vector and 100 vectors.

2.2.2 Discrete distributions

We next turn to the generation of discrete random variables, where we have an "all-purpose" algorithm. Again using the inverse transform principle of Section 2.1.2, we can indeed construct a generic algorithm that will formally work for any discrete distribution.

To generate $X \sim P_\theta$, where P_θ is supported by the integers, we can calculate—once for all, assuming we can store them—the probabilities

$$p_0 = P_\theta(X \leq 0), \quad p_1 = P_\theta(X \leq 1), \quad p_2 = P_\theta(X \leq 2), \quad \dots,$$

and then generate $U \sim \mathcal{U}_{[0,1]}$ and take

$$X = k \text{ if } p_{k-1} < U < p_k.$$

Example 2.4. To generate $X \sim \mathcal{B}in(10, .3)$, the probability values are obtained by `pbinom(k,10,.3)` as

$$p_0 = 0.028, \quad p_1 = 0.149, \quad p_2 = 0.382, \dots, p_{10} = 1\,,$$

and to generate $X \sim \mathcal{P}(7)$, take

$$p_0 = 0.0009, \quad p_1 = 0.0073, \quad p_2 = 0.0296, \dots\,,$$

the sequence being stopped when it reaches 1 with a given number of decimals. (For instance, $p_{20} = 0.999985$.) ◀

Specific algorithms are usually more efficient (as shown in Example 2.5), but it is mostly because of the storage problem. We can often improve on the algorithm above by a judicious choice of what probabilities we compute first. For example, if we want to generate random variables from a Poisson distribution with mean $\lambda = 100$, the algorithm above is woefully inefficient. This is because we expect most of our observations to be in the interval $\lambda \pm 3\sqrt{\lambda}$ (recall that λ is both the mean and the variance for the Poisson distribution), and for $\lambda = 100$ this interval is $(70, 130)$. Thus, starting at 0 will almost always produce 70 tests of whether or not $p_{k-1} < U < p_k$ that are useless because they will almost certainly be rejected. A first remedy is to "ignore" what is outside of a highly likely interval such as $(70, 130)$ in the current example, as

$$P(X < 70) + P(X > 130) = 0.00268.$$

Formally, we should find a lower and an upper bound to make this probability small enough, but informally $\pm 3\sigma$ works fine.

Example 2.5. Here is an `R` code that can be used to generate Poisson random variables for large values of lambda. The sequence `t` contains the integer values in the range around the mean.

```
> Nsim=10^4; lambda=100
> spread=3*sqrt(lambda)
> t=round(seq(max(0,lambda-spread),lambda+spread,1))
> prob=ppois(t, lambda)
> X=rep(0,Nsim)
> for (i in 1:Nsim){
+    u=runif(1)
+    X[i]=t[1]+sum(prob<u) }
```

The last line of the program checks to see what interval the uniform random variable fell in and assigns the correct Poisson value to X. See Exercise 2.14 for other distributions. ◀

A more formal remedy to the inefficiency of starting the cumulative probabilities at p_0 is to start instead from the mode of the discrete distribution P_θ and to explore the neighboring values until the cumulative probability is 1 up to an approximation error. The p_k's are then indexed by the visited values rather than by the integers, but the validity of the method remains complete.

Specific algorithms exist for almost any distribution and are often quite fast. Thus, we once again stress that, if `R` has the distribution that you are interested in, the wisest course is to use it. Once again, the comparison of the code of Example 2.5 with the resident `rpois` shows how inefficient this simple implementation can be. If `test3` corresponds to the above and `test4` to `rpois`, the execution times are given by

```
> system.time(test3()); system.time(test4())
   user  system elapsed
  0.436   0.000   0.435
   user  system elapsed
  0.008   0.000   0.006
```

However, R does not handle every distribution that we will need, so approaches such as the above can be useful. See Exercise 2.15 for some specific algorithms.

2.2.3 Mixture representations

It is sometimes the case that a probability distribution can be naturally represented as a *mixture distribution*; that is, we can write it in the form

$$f(x) = \int_{\mathcal{Y}} g(x|y)p(y)\ \mathrm{d}y \qquad \text{or} \qquad f(x) = \sum_{i\in\mathcal{Y}} p_i\ f_i(x)\ , \tag{2.2}$$

depending on whether the auxiliary space $\mathcal{Y}$ is continuous or discrete, where g and p are standard distributions that can be easily simulated. To generate a random variable X using such a representation, we can first generate a variable Y from the mixing distribution and then generate X from the selected conditional distribution. That is,

if $y \sim p(y)$ and $X \sim f(x|y)$, then $X \sim f(x)$ (if continuous);
if $\gamma \sim P(\gamma = i) = p_i$ and $X \sim f_\gamma(x)$, then $X \sim f(x)$ (if discrete).

For instance, we can write Student's t density with ν degrees of freedom $\mathcal{T}_\nu$ as a mixture, where

$$X|y \sim \mathcal{N}(0, \nu/y) \quad \text{and} \quad Y \sim \chi^2_\nu.$$

Generating from a $\mathcal{T}_\nu$ distribution could then amount to generating from a χ^2_ν distribution and then from the corresponding normal distribution. (Obviously, using `rt` is slightly more efficient, as you can check via `system.time`.)

Example 2.6. If X is a negative binomial random variable, $X \sim \mathcal{N}eg(n, p)$, then X has the mixture representation

$$X|y \sim \mathcal{P}(y) \quad \text{and} \quad Y \sim \mathcal{G}(n, \beta),$$

where $\beta = (1-p)/p$. The following R code generates from this mixture

```
> Nsim=10^4
> n=6;p=.3
> y=rgamma(Nsim,n,rate=p/(1-p))
> x=rpois(Nsim,y)
> hist(x,main="",freq=F,col="grey",breaks=40)
> lines(1:50,dnbinom(1:50,n,p),lwd=2,col="sienna")
```

and produces Figure 2.3, where the fit to the negative binomial pdf is shown as well. ◀

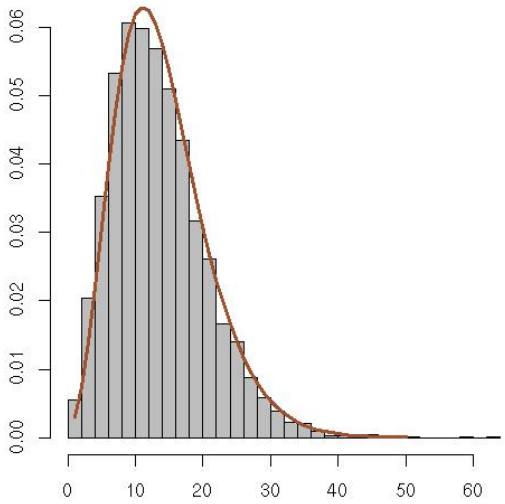

Fig. 2.3. Histogram of 10^4 negative binomial $\mathcal{N}eg(6, .3)$ random variables generated from the mixture representation along with the probability function.

2.3 Accept–reject methods

There are many distributions for which the inverse transform method and even general transformations will fail to be able to generate the required random variables. For these cases, we must turn to *indirect* methods; that is, methods in which we generate a candidate random variable and only accept it subject to passing a test. As we will see, this class of methods is extremely powerful and will allow us to simulate from virtually any distribution.

These so-called *Accept–Reject methods* only require us to know the functional form of the density f of interest (called the *target density*) up to a multiplicative constant. We use a simpler (to simulate) density g, called the *instrumental or candidate density*, to generate the random variable for which the simulation is actually done. The only constraints we impose on this candidate density g are that

(i). f and g have compatible supports (i.e., $g(x) > 0$ when $f(x) > 0$).
(ii). There is a constant M with $f(x)/g(x) \leq M$ for all x.

In this case, X can be simulated as follows. First, we generate $Y \sim g$ and, independently, we generate $U \sim \mathcal{U}_{[0,1]}$. If

$$U \leq \frac{1}{M} \frac{f(Y)}{g(Y)},$$

then we set $X = Y$. If the inequality is not satisfied, we then discard Y and U and start again. Succinctly, the algorithmic representation of the Accept–Reject method is as follows:

Algorithm 1 Accept–Reject Method

1. Generate $Y \sim g$, $U \sim \mathcal{U}_{[0,1]}$;
2. Accept $X = Y$ if $U \leq f(Y)/Mg(Y)$;
3. Return to 1 otherwise.

The `R` implementation of this algorithm is straightforward: If `randg` is a function that delivers generations from the density g, in the same spirit as `rnorm` or `rt`, a simple `R` version of Algorithm 1 is

```
> u=runif(1)*M
> y=randg(1)
> while (u>f(y)/g(y)){
+   u=runif(1)*M
+   y=randg(1)}
```

which produces a single generation y from f.

Why does this method work? A straightforward probability calculation shows that the cdf of the accepted random variable, $P(Y \leq x | U \leq f(Y)/\{Mg(Y)\})$, is exactly the cdf of X. That is,

$$\begin{aligned} P(Y \leq x | U \leq f(Y)/\{Mg(Y)\}) &= \frac{P(Y \leq x, U \leq f(Y)/\{Mg(Y)\})}{P(U \leq f(Y)/\{Mg(Y)\})} \\ &= \frac{\int_{-\infty}^{x} \int_{0}^{f(y)/\{Mg(y)\}} \mathrm{d}u \, g(y) \, \mathrm{d}y}{\int_{-\infty}^{\infty} \int_{0}^{f(y)/\{Mg(y)\}} \mathrm{d}u \, g(y) \, \mathrm{d}y} \\ &= \frac{\int_{-\infty}^{x} [f(y)/\{Mg(y)\}] \, g(y) \, \mathrm{d}y}{\int_{-\infty}^{\infty} [f(y)/\{Mg(y)\}] \, g(y) \, \mathrm{d}y} \\ &= \frac{\int_{-\infty}^{x} f(y) \, \mathrm{d}y}{\int_{-\infty}^{\infty} f(y) \, \mathrm{d}y} = P(X \leq x), \end{aligned}$$

where we use the fact that the uniform integral is equal to its upper limit. Despite simulating only from g, the output of this algorithm is thus exactly distributed from f.

ϟ The Accept–Reject method is applicable in any dimension, provided g is a density over the same space as f.

Note the cancellation of the $g(y)$'s and the M's in the integrals above. It also follows from this representation that we do not need to be concerned about normalizing constants. As long as we know f/g up to a constant, $f/g \propto \tilde{f}/\tilde{g}$, the algorithm can be implemented if an upper bound $\widetilde{M}$ can be found on $\tilde{f}/\tilde{g}$. (The missing constants actually get absorbed into M.)

Exercise 2.5 Show that the probability of acceptance in an Accept–Reject algorithm with upper bound M on the density ratio f/g is $1/M$. Show that the expected value of the acceptance rate, $\mathbb{E}[I(U < \tilde{f}/\tilde{M}\tilde{g})]$, can be used to compute the missing constant in f/g.

As stressed by this exercise, the probability of acceptance is $1/M$ *only if* the normalizing constants are known. Otherwise, since the missing constants do get absorbed into $\tilde{M}$, $1/\tilde{M}$ is not the probability of acceptance.

Example 2.7. Example 2.2 did not provide a general algorithm to simulate beta $\mathcal{B}e(\alpha, \beta)$ random variables. We can, however, construct a toy algorithm based on the Accept–Reject method, using as the instrumental distribution the uniform $\mathcal{U}_{[0,1]}$ distribution when both α and β are larger than 1. (The generic `rbeta` function does not impose this restriction.)

The upper bound M is then the maximum of the beta density, obtained for instance by `optimize` (or its alias `optimise`):

```
> optimize(f=function(x){dbeta(x,2.7,6.3)},
+ interval=c(0,1),max=T)$objective
[1] 2.669744
```

Since the candidate density g is equal to one, the proposed value Y is accepted if $M \times U < f(Y)$, that is, if $M \times U$ is under the beta density f at that realization. Note that generating $U \sim \mathcal{U}_{[0,1]}$ and multiplying by M is equivalent to generating $U \sim \mathcal{U}_{[0,M]}$. For $\alpha = 2.7$ and $\beta = 6.3$, an alternative `R` implementation of the Accept–Reject algorithm is

```
> Nsim=2500
> a=2.7;b=6.3
> M=2.67
> u=runif(Nsim,max=M)     #uniform over (0,M)
> y=runif(Nsim)           #generation from g
> x=y[u<dbeta(y,a,b)]     #accepted subsample
```

and the left panel in Figure 2.4 shows the results of generating 2500 pairs (Y, U) from $\mathcal{U}_{[0,1]} \times \mathcal{U}_{[0,M]}$. The black dots $(Y, Ug(Y))$ that fall under the density f are those for which we accept $X = Y$, and we reject the grey dots $(Y, Ug(Y))$ that fall outside. It is again clear from this graphical representation that the black dots are uniformly distributed over the area under the density f. Since the probability of acceptance of a given simulation is $1/M$ (Exercise 2.5), with $M = 2.67$ we accept approximately $1/2.67 = 37\%$ of the values. ◀

In the implementation of the Accept–Reject algorithm above, the total number of attempts `Nsim` is fixed, which means that the number of accepted values is a binomial random variable with probability $1/M$. Instead, in most cases, the number of accepted values is fixed, but this implementation can nonetheless be exploited as in

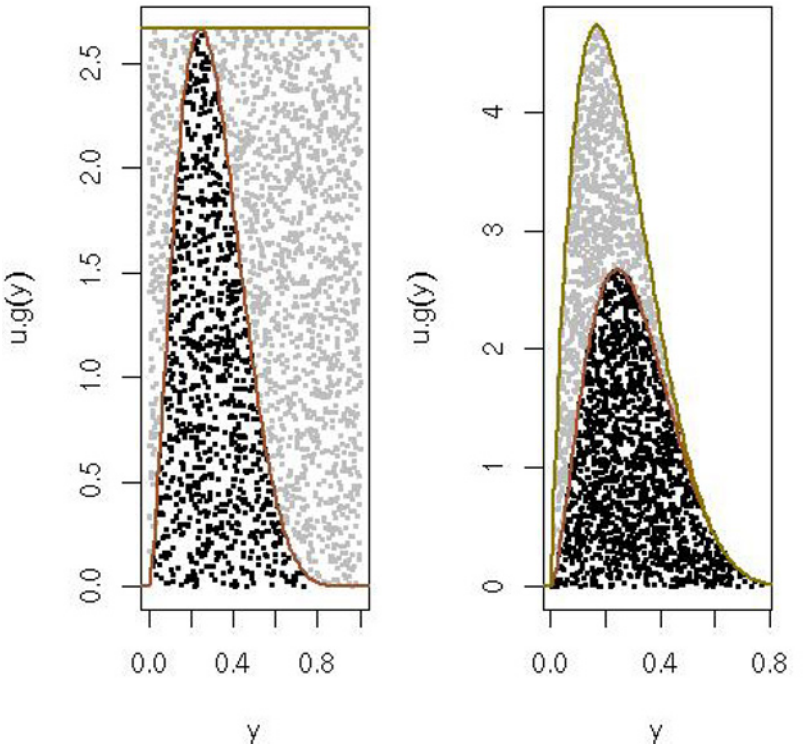

Fig. 2.4. Generation of beta random variables $X \sim \mathcal{B}e(2.7, 6.3)$: Using the Accept–Reject algorithm, 2500 (Y, U) proposals were generated from g and $\mathcal{U}_{[0,M]}$, respectively, and the points $(Y, Ug(Y))$ were represented with grey dots. In the left panel, $Y \sim \mathcal{U}_{[0,1]}$, and 36% of the candidate random variables were accepted and represented with black dots. In the right panel, $Y \sim \mathcal{B}e(2, 6)$ and 58% of the simulated values were accepted (and similarly represented with black dots). In both panels, f and Mg are also plotted.

```
> x=NULL
> while (length(x)<Nsim){
+    y=runif(Nsim*M)
+    x=c(x,y[runif(Nsim*M)*M<dbeta(y,a,b)])}
> x=x[1:Nsim]
```

(Note that using `y=u=runif(Nsim*M)` in the program would produce a bias, as `y` and `u` would then take the same values.) Simulating `Nsim*M` proposals from the start reduces the number of calls to `while` since this is the expected number of proposals (Exercise 2.5).

Exercise 2.6 Compare the execution times of the two proposed implementations of the Accept–Reject algorithm, as well as alternatives simulating `Nsim*Nprop` proposals at once when `Nprop` varies.

ϟ Some key properties of the Accept–Reject algorithm, which should always be considered when using it, are the following:

1. Only the ratio f/M is needed, so the algorithm does not depend on the normalizing constant.
2. The bound $f \leq Mg$ need not be tight; the algorithm remains valid (if less efficient) when M is replaced with any larger constant.
3. The probability of acceptance is $1/M$, so M should be as small as possible for a given computational effort.

The efficiency of a given Accept–Reject algorithm can be measured in terms of its acceptance probability since the higher this probability, the fewer wasted simulations from g. (In absolute terms, this must be weighted down by the computational cost of producing a value from g since otherwise the best choice for g would be f!)

Example 2.8. (Continuation of Example 2.7) Consider instead simulating $Y \sim \mathcal{B}e(2,6)$ as a proposal distribution. This choice of g is acceptable since

```
> optimize(f=function(x){dbeta(x,2.7,6.3)/dbeta(x,2,6)},
+ max=T,interval=c(0,1))$objective
1.671808
```

This modification of the proposal thus leads to a smaller value of M and a correspondingly higher acceptance rate of 58% than with the uniform proposal. The right panel of Figure 2.4 shows the outcome of the corresponding Accept–Reject algorithm and illustrates the gain in efficiency brought by simulating points in a smaller set. ◀

Exercise 2.7 Show formally that, for the ratio f/g to be bounded when f is a $\mathcal{B}e(\alpha,\beta)$ density and g is a $\mathcal{B}e(a,b)$ density, we must have both $a \leq \alpha$ and $b \leq \beta$. Deduce that the best choice for a and b among the integer values is $a = \lfloor \alpha \rfloor$ and $b = \lfloor \beta \rfloor$.

As shown by Example 2.8, some optimization of the Accept–Reject algorithm is possible by choosing the candidate density g in a parametric family and by then determining the value of the parameter that minimizes the bound M.

Exercise 2.8 Consider using the Accept–Reject algorithm to generate a $\mathcal{N}(0,1)$ random variable from a double-exponential distribution $\mathcal{L}(\alpha)$, with density $g(x|\alpha) = (\alpha/2)\exp(-\alpha|x|)$ as a candidate.

a. Show that
$$\frac{f(x)}{g(x|\alpha)} \leq \sqrt{\frac{2}{\pi}}\,\alpha^{-1}e^{\alpha^2/2}$$
and that the minimum of this bound (in α) is attained for $\alpha = 1$.

b. Show that the probability of acceptance is then $\sqrt{\pi/2e} = .76$ and deduce that, to produce one normal random variable, this Accept–Reject algorithm requires on average $1/.76 \approx 1.3$ uniform variables.

c. Show that $\mathcal{L}(\alpha)$ can be generated by the probability inverse transform, and compare this algorithm with the Box–Muller algorithm of Example 2.3 in terms of execution time.

It may sometimes happen that the complexity of the optimization is very expensive in terms of analysis or computing time. In the first case, the construction of the optimal algorithm should still be undertaken when the algorithm is to be subjected to intensive use. This is, for instance, the case for most random generators in `R`, as can be checked by `help`. In the second case, it is most often preferable to explore the use of another family of instrumental distributions g. (See Exercise 2.22.)

One particular application of the Accept–Reject algorithm has found a niche in population genetics and is called ABC, following the denomination proposed by Pritchard et al. (1999). The core version of this algorithm is an Accept–Reject algorithm fitted for Bayesian problems, where a posterior distribution $\pi(\theta|x_0) \propto \pi(\theta)f(x_0|\theta)$ is to be simulated for a likelihood function $f(x|\theta)$ that is not available but can be simulated. The ABC algorithm then generates values from the prior and from the likelihood until the simulated observation is equal to the original observation x_0:

Repeat

Generate $\theta \sim \pi(\theta)$ and $X \sim f(x|\theta)$

until $X = x_0$

Exercise 2.9 Prove that the conditional probability of acceptance in the loop above is $f(x_0|\theta)$, and deduce that the distribution of the accepted θ is $\pi(\theta|x_0)$.

This algorithm is thus valid, but it only applies in settings where $\pi(\theta)$ is a proper prior and where $P_\theta(X = x_0)$ has a positive probability of occurring. Even in population genetics where X is a discrete random variable, the size of the state-space is often such that this algorithm cannot be implemented. The proposal of Pritchard et al. (1999) is then to replace the *exact* acceptance condition $X = x_0$ with an *approximate* condition $d(X, x_0) < \epsilon$, where d is a distance and ϵ a tolerance level. While unavoidable, this approximation step

makes the ABC method difficult to recommend on a general basis, even though more recent works rephrase it in a non-parametric framework that aims at approximating the likelihood function $f(x|\theta)$ (Beaumont et al., 2002).

One criticism of the Accept–Reject algorithm is that it generates "useless" simulations from the proposal g when rejecting, even those necessary to validate the output as being generated from the target f. We will see in Chapter 3 how the method of importance sampling (Section 3.3) can be used to bypass this problem.

2.4 Additional exercises

Exercise 2.10 The vector `randu` is a historical reminder of how wrong a random generator can get. It consists of 400 rows of three consecutive values produced by a former VAX random generator called RANDU.

a. Produce a random sample by taking all columns of `randu`, and reproduce Figure 2.1.
b. Show that the triplets `randu[i,]` lie on one of 15 parallel hyperplanes.

Exercise 2.11 In both questions, the comparison between generators is understood in terms of efficiency via the `system.time` function.

a. Generate a binomial $\mathcal{B}in(n,p)$ random variable with $n = 25$ and $p = .2$. Plot a histogram for a simulated sample and compare it with the binomial mass function. Compare your generator with the `R` binomial generator.
b. For $\alpha \in [0,1]$, show that the `R` code

```
> u=runif(1)
> while(u > alpha) u=runif(1)
> U=u
```

produces a random variable U from $\mathcal{U}([0,\alpha])$. Compare it with the transform αU, $U \sim \mathcal{U}(0,1)$, for values of α close to 0 and close to 1, and with `runif(1,max=alpha)`.

Exercise 2.12 Referring to Example 2.2,

a. Generate gamma and beta random variables according to (2.1).
b. Show that if $U \sim \mathcal{U}_{[0,1]}$, then $X = -\log U/\lambda \sim \mathcal{E}xp(\lambda)$.
c. Show that if $U \sim \mathcal{U}_{[0,1]}$, then $X = \log \frac{u}{1-u}$ is a Logistic$(0,1)$ random variable.

Exercise 2.13 The Pareto $\mathcal{P}(\alpha)$ distribution is defined by its density $f(x|\alpha) = \alpha x^{-\alpha-1}$ over $(1,\infty)$. Show that it can be generated as the $-1/\alpha$ power of a uniform variate. Plot the histogram and the density.

Exercise 2.14 Referring to Example 2.5:

a. Verify the `R` code for the Poisson generator. Compare it with `rpois`.

b. The *negative binomial distribution*, with parameters r and p, has mass function

$$P(Y = y) = \binom{r+y-1}{y} p^r (1-p)^y, \quad y = 0, 1, \ldots,$$

with mean $r(1-p)/p$ and variance $r(1-p)/p^2$. For $r = 10$ and $p = .01, .1, .5$, generate 1000 random variables and draw their histograms. Compare the histograms with the probability functions and your generator with `rnegbin` (which is in the `MASS` package).

c. The *logarithmic series* distribution has mass function

$$P(X = x) = \frac{-(1-p)^x}{x \log p}, \quad x = 1, 2, \ldots, \quad 0 < p < 1.$$

For $p = .001, .01, .5$, generate 1000 random variables and draw a histogram. Compare the histograms with the probability functions.

Exercise 2.15 The Poisson distribution $\mathcal{P}(\lambda)$ is connected to the exponential distribution through the Poisson process in that it can be simulated by generating exponential random variables until their sum exceeds 1. That is, if $X_i \sim \mathcal{E}xp(\lambda)$ and if K is the first value for which $\sum_{i=1}^{K+1} X_i > 1$, then $K \sim \mathcal{P}(\lambda)$. Compare this algorithm with `rpois` and the algorithm of Example 2.5 for both small and large values of λ.

Exercise 2.16 An algorithm to generate beta random variables was given in Example 2.2 for $\alpha \geq 1$ and $\beta \geq 1$. Another algorithm is based on the following property: If U and V are iid $\mathcal{U}_{[0,1]}$, the distribution of

$$\frac{U^{1/\alpha}}{U^{1/\alpha} + V^{1/\beta}},$$

conditional on $U^{1/\alpha} + V^{1/\beta} \leq 1$, is the $\mathcal{B}e(\alpha, \beta)$ distribution. Compare this algorithm with `rbeta` and the algorithm of Example 2.2 for both small and large values of α, β.

Exercise 2.17 We saw in Example 2.2 that, if $\alpha \in \mathbb{N}$, the gamma distribution $\mathcal{G}a(\alpha, \beta)$ can be represented as the sum of α exponential random variables $\epsilon_i \sim \mathcal{E}xp(\beta)$. When $\alpha \notin \mathbb{N}$, this representation does not hold.

a. Show that we can assume $\beta = 1$ by using the transformation $y = \beta x$.
b. When the $\mathcal{G}(n, 1)$ distribution is generated from an $\mathcal{E}xp(\lambda)$ distribution, determine the optimal value of λ.
c. When $\alpha \geq 1$, show that we can use the Accept–Reject algorithm with candidate distribution $\mathcal{G}a(a, b)$ to generate a $\mathcal{G}a(\alpha, 1)$ distribution, as long as $a \leq \alpha$. Show that the ratio f/g is $b^{-a} x^{\alpha-a} \exp\{-(1-b)x\}$, up to a normalizing constant, yielding the bound
$$M = b^{-a} \left(\frac{\alpha - a}{(1-b)e} \right)^{\alpha - a}$$
for $b < 1$.
d. Show that the maximum of $b^{-a}(1-b)^{\alpha-a}$ is attained at $b = a/\alpha$, and hence the optimal choice of b for simulating $\mathcal{G}a(\alpha, 1)$ is $b = a/\alpha$, which gives the same mean for $\mathcal{G}a(\alpha, 1)$ and $\mathcal{G}a(a, b)$.
e. Defend the choice of $a = \lfloor \alpha \rfloor$ as the best choice of a among the integers.
f. Discuss the strategy to adopt when $\alpha < 1$.

Exercise 2.18 The rather strange density

$$f(x) \propto \exp(-x^2/2)\left\{\sin(6x)^2 + 3\cos(x)^2\sin(4x)^2 + 1\right\}$$

can be generated using the Accept–Reject algorithm.

a. Plot $f(x)$ and show that it can be bounded by $Mg(x)$, where g is the standard normal density $g(x) = \exp(-x^2/2)/\sqrt{2\pi}$. Find an acceptable if not necessarily optimal value of M. (*Hint:* Use the function `optimise`.)
b. Generate 2500 random variables from f using the Accept–Reject algorithm.
c. Deduce from the acceptance rate of this algorithm an approximation of the normalizing constant of f, and compare the histogram with the plot of the normalized f.

Exercise 2.19 In an Accept–Reject algorithm that generates a $\mathcal{N}(0,1)$ random variable from a double-exponential distribution with density $g(x|\alpha) = (\alpha/2)\exp(-\alpha|x|)$, compute the upper bound M over f/g and show that the choice $\alpha = 1$ optimizes the corresponding acceptance rate.

Exercise 2.20 In each of the following cases, construct an Accept–Reject algorithm, generate a sample of the corresponding random variables, and draw the density function on top of the histogram.

a. Generate normal random variables using a Cauchy candidate in Accept–Reject.
b. Generate gamma $\mathcal{G}(4.3, 6.2)$ random variables using a gamma $\mathcal{G}(4, 7)$ candidate.

Exercise 2.21 The noncentral chi-squared distribution, $\chi^2_p(\lambda)$, can be defined by

(i). a mixture representation (2.2), where $g(x|y)$ is the density of χ^2_{p+2y} and $p(y)$ is the density of $\mathcal{P}(\lambda/2)$, and
(ii). the sum of a χ^2_{p-1} random variable and the square of a $\mathcal{N}(||\theta||, 1)$.

a. Show that both those representations hold.
b. Show that the representations are equivalent if $\lambda = \theta^2/2$.
c. Compare the corresponding algorithms that can be derived from these representations among themselves and also with `rchisq` for small and large values of λ.

Exercise 2.22 *Truncated normal distributions* $\mathcal{N}^+(\mu, \sigma^2, a)$, in which the range of a normal random variable is truncated, appear in many contexts. If $X \sim \mathcal{N}(\mu, \sigma^2)$, conditional on the event $\{x \geq a\}$, its density is proportional to

$$\exp\left\{-(x-\mu)^2/2\sigma^2\right\}\ \mathbb{I}_{x \geq a}.$$

a. The naïve method of simulating this random variable is to generate a $\mathcal{N}(\mu, \sigma^2)$ until the generated value is larger than a. Implement the `R` code

```
> Nsim=10^4
> X=rep(0,Nsim)
> for (i in 1:Nsim){
+  z=rnorm(1,mean=mu,sd=sigma)
+  while(z<a) z=rnorm(1,mean=mu,sd=sigma)
+  X[i]=z}
```

and evaluate the algorithm for $\mu = 0$, $\sigma = 1$, and various values of a.

b. Show that the algorithm in part a requires, on average, $1/\Phi((\mu-a)/\sigma)$ simulations from $\mathcal{N}(\mu,\sigma^2)$ for one acceptance. Deduce that, if a is in the tail of the distribution, this algorithm will take a very long time.

c. We now consider the case where $\mu = 0$ and $\sigma = 1$. Show that an Accept–Reject algorithm based on a normal $\mathcal{N}(\overline{\mu},1)$ candidate can be implemented to generate from the $\mathcal{N}^+(0,1,a)$ distribution for $\overline{\mu} > 0$. For a given a, discuss the optimization in $\overline{\mu}$.

d. Another potential candidate distribution for the Accept–Reject algorithm is the translated exponential distribution, $\mathcal{E}xp(\alpha,a)$, with density

$$g_\alpha(z) = \alpha e^{-\alpha(z-a)}\, \mathbb{I}_{z\geq a}\,.$$

Show that the ratio $(f/g_\alpha)(z) \propto e^{\alpha(z-a)}\, e^{-z^2/2}$ is then bounded by $\exp(\alpha^2/2-\alpha a)$ if $\alpha \geq a$. Deduce that $a = \alpha$ gives a legitimate candidate density. Compare the performance of the corresponding Accept–Reject algorithm based on an $\mathcal{E}xp(a,a)$ candidate with the algorithm in part c, especially for a located in the tails of the normal $\mathcal{N}(0,1)$ distribution.

(The scale of the exponential distribution in part d can be optimized, but this may not lead to explicit expressions for the candidate scale. Two-sided normal truncation (that is, when $b \leq x \leq a$) is a bit more tricky to deal with. See Robert (1995b) for a resolution, or use `rtrun` from the package `bayesm`. Also see Exercise 7.21 for another truncated normal generator.)

Exercise 2.23 Given a sampling density $f(x|\theta)$ and a prior density $\pi(\theta)$, if we observe $\mathbf{x} = x_1,\ldots,x_n$, the posterior distribution of θ is

$$\pi(\theta|\mathbf{x}) = \pi(\theta|x_1,\ldots,x_n) \propto \prod_i f(x_i|\theta)\pi(\theta),$$

where $\prod_i f(x_i|\theta) = L(\theta|x_1,\ldots,x_n)$ is the likelihood function.

a. If $\pi(\theta|\mathbf{x})$ is the target density in an Accept–Reject algorithm, and if $\pi(\theta)$ is the candidate density, show that the optimal bound M is the likelihood function evaluated at the MLE.

b. For estimating a normal mean, a robust prior is the Cauchy. For $X_i \sim \mathcal{N}(\theta,1)$, $\theta \sim \mathcal{C}(0,1)$, the posterior distribution is

$$\pi(\theta|\mathbf{x}) \propto \frac{1}{\pi}\,\frac{1}{1+\theta^2}\,\frac{1}{2\pi}\prod_{i=1}^n e^{-(x_i-\theta)^2/2}.$$

Set $\theta_0 = 3$, $n = 10$, and generate $X_1,\ldots,X_n \sim \mathcal{N}(\theta_0,1)$. Use the Accept–Reject algorithm with a Cauchy $\mathcal{C}(0,1)$ candidate to generate a sample from the posterior distribution. Evaluate how well the value θ_0 is recovered. How much better do things get if n is increased?