5.4 Multivariate random number generation

Most interesting problems in statistics and applied probability don't involve just one random value, they will involve several values that are related. R has a few multivariate distributions built in, but in many cases users need to construct their own simulations. There are many techniques for doing this. We will describe just one: sequential conditional generation.

The idea of this technique is that even if $X = (X_1, X_2, \ldots, X_p)$ has a complicated p-dimensional distribution, X_1 will have a one-dimensional marginal distribution. So we start by simulating just X_1, then simulate X_2 from its conditional distribution given X_1, and X_3 conditional on X_1 and X_2, and so on. If it is difficult to calculate marginal and conditional distributions then this might not be feasible, but in a surprising number of cases, it is not.

Example 5.15

A model commonly used in financial engineering says that the price of a stock X_{t+1} at the close of trading on day $t+1$ is equal to the closing price X_t on the previous day, multiplied by a value whose logarithm has a normal distribution. The multipliers are independent from day to day.

Suppose a stock starts at \$100.00 per share, and we expect a 1% change in price from one day to the next. What would be the joint distribution of the prices over the next five days? What is the probability that it remains above \$100 on all five days?

We can simulate this by starting with $X_0 = 100$, and multiplying by $\exp(Z_t)$, where Z_t, $t = 1, \ldots, 5$ are independent normal random variables with mean 0 and standard deviation of $\log(1.01)$. We do this in a `replicate()` loop to get 1000 examples of the simulation:

```
X <- numeric(6)
X[1] <- 100
results <- replicate(1000, {
  for (t in 1:5)
    X[t + 1]  <- X[t] * exp(rnorm(1, mean = 0, sd = log(1.01)))
  X
})
str(results)

##  num [1:6, 1:1000] 100 101.1 100 98.1 98.1 ...
```

We see that they have been returned in a matrix with 6 rows and 1000 columns. How many of those cases have all values greater than 100?

```
table(apply(results, 2, function(x) all(x[2:6] > 100)))

##
## FALSE  TRUE
##   749   251
```

We see that it happens about a quarter of the time.

5.5 Markov chain simulation

Markov chains are sequences of random variables $X_0, X_1, \ldots$ where the distribution of X_{t+1}, conditional on all previous values, depends only on X_t. They are commonly used in modeling systems with short memories, such as the stock market, where the price tomorrow is modeled to depend only on the price today, biological systems where the genome of the offspring depends only on the genome of the parents, etc. When X_t is a discrete random variable with a finite number of states $1, \ldots, n$, it is convenient to represent the conditional distributions in an $n \times n$ matrix P, where entry P_{ij} holds the conditional probability that $X_{t+1} = j$ given that $X_t = i$. Because probabilities sum to 1, we have row sums $\sum_{j=1}^{n} P_{ij} = 1$ for all $i = 1, \ldots, n$.

Simulation of such Markov chains in R is easy using the sequential conditional simulation method that was described in the previous section. The value of X_{t+1} is simply the value drawn by

```
sample(1:n, size = 1, prob = P[X[t], ])
```

An interesting fact about Markov chains is that they have an *invariant distribution*, i.e. a distribution $P(X_t = i) = \pi_i$ such that if X_t is drawn from the invariant distribution and updated using the P matrix, then the marginal distribution of X_{t+1} will also be the invariant distribution. (For some P, there will be more than one invariant distribution, but there is always at least one.) Even more interesting is the fact that for *some* P matrices, if X_0 is drawn from *any* distribution, then the marginal distribution of X_t for large t approximates the invariant distribution. This is used in methods called Markov chain Monte Carlo (or MCMC) to draw values with distribution close to π_i even when π_i can't be calculated directly.

Example 5.16

Consider a model of a disease with three stages. Stage 1 is healthy, stage 2 is mild disease, and stage 3 is severe disease. Healthy individuals remain healthy with probability 0.99 and develop mild disease with probability 0.01. Individuals with mild disease are cured and become healthy with probability 0.5, remain with mild disease with probability 0.4, and progress to serious disease with probability 0.1. Finally, those with severe disease stay in that state with probability 0.75, and improve to mild disease with probability 0.25.

This describes a Markov chain with three states, and

$$P = \begin{bmatrix} 0.99 & 0.01 & 0.00 \\ 0.50 & 0.40 & 0.10 \\ 0.00 & 0.25 & 0.75 \end{bmatrix}.$$

We will simulate two individuals for 10000 steps: one who starts healthy, and one who starts with severe disease:

```
P <- matrix(c(0.99, 0.01,  0,
              0.5,  0.4,   0.1,
              0,    0.25,  0.75), 3, 3, byrow = TRUE)
healthy <- numeric(10000)
healthy[1] <- 1
for (t in 1:9999)
  healthy[t + 1] <- sample(1:3, size = 1, prob = P[healthy[t], ])
table(healthy)
```

```
## healthy
##    1    2    3
## 9692  218   90
```

```
sick <- numeric(10000)
sick[1] <- 3
for (t in 1:9999)
  sick[t + 1] <- sample(1:3, size = 1, prob = P[sick[t], ])
table(sick)
```

```
## sick
##    1    2    3
## 9646  216  138
```

We see that the two individuals have very similar distributions of states over the 10000 steps. In fact, their distribution is close to the (unique) invariant distribution for P, which can be shown by other means (see Chapter exercise 1 in Chapter 6) to be approximately (0.973, 0.020, 0.008).

Example 5.17

The tune for L. van Beethoven's *Ode to Joy* can be represented in "Lilypond" format, a standard format for representing musical scores on a computer, as follows:

```
\score{
 {
 \tempo 4 = 120
 e'4 e'4 f'4 g'4  g'4 f'4 e'4 d'4  c'4 c'4 d'4 e'4  e'4. d'8 d'2
 e'4 e'4 f'4 g'4  g'4 f'4 e'4 d'4  c'4 c'4 d'4 e'4  d'4. c'8 c'2
 d'4 d'4 e'4 c'4  d'4 e'8 f'8 e'4  c'4 d'4 e'8 f'8 e'4 d'4 c'4 d'4 g2
 e'4 e'4 f'4 g'4  g'4 f'4 e'4 d'4  c'4 c'4 d'4 e'4  d'4. c'8 c'2
 }
 \layout{}
 \midi{}
}
```

The actual music starts on the fourth line; each note is represented by symbols containing two parts: a letter for the pitch, and a number for the duration. The punctuation marks modify the pitch or duration. This file can be converted to standard musical notation or to a "MIDI" file which

can be played on most computers. At the time of writing, the website `http://lilybin.com` contains online tools to do this.

We can compose music with some of the characteristics of the *Ode to Joy* by randomly sampling the same set of symbols with R. First, we save the lines to a file `ode.ly`, and read the symbols into a character vector, with one symbol per element:

```
notes <- scan("ode.ly", skip = 3, nlines = 4, what = character())
```

We can randomly sample notes and write the results to a file using the following code:

```
random <- sample(notes, 63, replace = TRUE)

writeLilypond <- function(notes, filename = "") {
  cat(c("\\score{\n {\n  \\tempo 4 = 120",
        paste(" ", strwrap(paste(notes, collapse = " "))),
        " }\n \\layout{}\n \\midi{}\n}"),
      sep = "\n", file = filename)
}

writeLilypond(random, filename = "indep.txt")
```

This gives a pleasant sound, but it's not much like Beethoven. We can make it more like the original by sampling from a Markov chain with transition probabilities similar to the original. But it would be nice to keep even more of the characteristics, for example having the selection of notes match the original frequencies conditional on *two* previous notes, not just one. That is, we would like the distribution of note Y_t to depend on both Y_{t-1} and Y_{t-2}.

It turns out we can do this, and still have a Markov chain. Define $Z_t = (Y_{t-1}, Y_t)$. Then Z_t is a Markov chain from which we can extract Y_t easily. We can generalize this to allow Y_t to depend on n previous observations.

One practical problem with simulating a process like this is that if Y_t has m states, then Z_t has m^n states. Our sample song has 14 different notes; if we want to match the distribution for 5 consecutive notes, we'd need $14^5 = 537824$ different possible states for Z_t, and the transition matrix P for Z_t might be too large to manage. However, the P matrix will be "sparse": most entries will be zero. For example, our song has only 61 transitions, with some repetition. It is easy to represent this sparse matrix, but we use `list()`, not `matrix()`. The following function counts the transitions in the original data:

```
countTransitions <- function(notes, n) {
  len <- length(notes)
  # Put special markers at the start and end
  notes <- c(rep("START", n), notes, "STOP")
  result <- list(n = n)
  # Loop over the data + STOP, counting symbols
  for (i in 1:(len+1)) {
    index <- notes[ seq_len(n+1) + i - 1 ]
    # We store only positive results, so if we've never seen this
```

```
    # transition before, it will be NULL; insert an empty list()
    for (j in seq_len(n)) {
      if (is.null(result[[ index[1:j] ]]))
        result[[ index[1:j] ]] <- list()
    }
    prevcount <- result[[ index ]]
    if (is.null(prevcount))
      result[[ index ]] <- 1
    else
      result[[ index ]] <- result[[ index ]] + 1
  }
  result
}
```

This returns the result in a list that records n. We can use this list to generate new music using this function:

```
generateTransitions <- function(counts, len, filename = "") {
  # Initialize with START
  n <- counts$n
  len <- len + n
  notes <- rep("START", len)
  i <- n
  while (i < len && notes[i] != "STOP" ) {
    i <- i + 1
    index <- notes[i - n:1]
    distn <- counts[[ index ]]
    notes[i] <- sample(names(distn), 1, prob = distn)
  }
  # Now leave off START and STOP, and return the result
  notes[ !(notes %in% c("START", "STOP")) ]
}
```

Here we give a maximum length, but if we reach the `STOP` marker, we stop early. Using these functions with $n = 2$ produces this output, which is shown in standard musical notation in Figure 5.2:

```
counts <- countTransitions(notes, 2)
writeLilypond(generateTransitions(counts, 100))
```

```
\score{
 {
  \tempo 4 = 120
  e'4 e'4 f'4 g'4 g'4 f'4 e'4 d'4 c'4 c'4 d'4 e'4 c'4 d'4 e'4
  e'4. d'8 d'2 e'4 e'4 f'4 g'4 g'4 f'4 e'4 d'4 c'4 c'4 d'4 e'4
  e'4. d'8 d'2 e'4 e'4 f'4 g'4 g'4 f'4 e'4 d'4 c'4 c'4 d'4 e'4
  d'4. c'8 c'2
 }
 \layout{}
 \midi{}
}
```


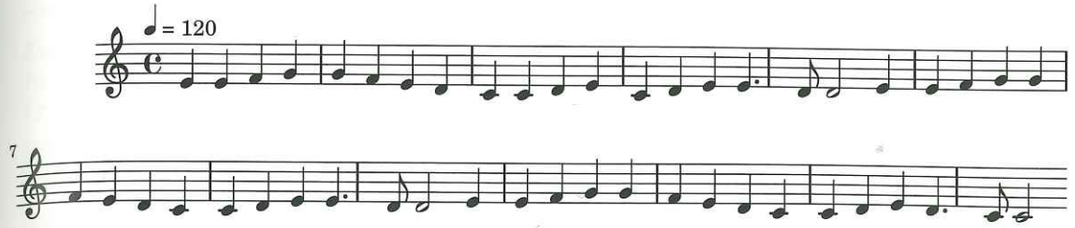

Fig. 5.2 A random variation on Beethoven's *Ode to Joy*. The display was produced by Lilypond on `lilybin.com`.

This sounds like the *Ode to Joy* played by a musician who can't quite remember it. If we had started with a longer score, or mixed the transitions from two tunes, we would get something more original.

5.6 Monte Carlo integration

Suppose $g(x)$ is any function that is integrable on the interval $[a, b]$. The integral

$$\int_a^b g(x)dx$$

gives the area of the region with $a < x < b$ and y between 0 and $g(x)$ (where negative values count towards negative areas).

Monte Carlo integration uses simulation to obtain approximations to these integrals. It relies on the law of large numbers. This law says that a sample mean from a large random sample will tend to be close to the expected value of the distribution being sampled. If we can express an integral as an expected value, we can approximate it by a sample mean.

For example, let $U_1, U_2, \ldots, U_n$ be independent uniform random variables on the interval $[a, b]$. These have density $f(u) = 1/(b - a)$ on that interval. Then

$$E[g(U_i)] = \int_a^b g(u)\frac{1}{b-a}\,du,$$

so the original integral $\int_a^b g(x)dx$ can be approximated by $b - a$ times a sample mean of $g(U_i)$.

Example 5.18

To approximate the integral $\int_0^1 x^4\,dx$, use the following lines:

```
u <- runif(100000)
mean(u^4)
```

```
## [1] 0.2005908
```