CHAPTER 21

# Case studies

## 21.1 Introduction

In this chapter we present three case studies: extended examples intended to demonstrate some of our simulation techniques. Simulation is ubiquitous in science, so trying to list all the areas where it appears would be an endless task. To give you a taste, here are some (but not all) of the areas where simulation is being employed in the authors' home department.

- Spin systems: big lattices of interacting molecules.
- Granular materials: how do grains of dirt move about when you put a weight (like a building) on them?
- Molecular geometry: the shape of complex molecules has an important effect on how they act.
- Stock markets: how should we value financial instruments such as bonds, options, etc?
- Health care: modelling and then optimising patient care.
- Telecommunications: optimal design of communication networks.
- Carbon modelling: where is all the carbon, and how will it affect global warming?
- Forest management: where, when, and what should I plant?

The development of new simulation techniques is a scientific field in its own right. As computer power increases, numerical simulation and optimisation techniques become more sophisticated and more widely applicable. Here is a list (not exhaustive) of some of the simulation topics we have not been able to cover in this book. The interested reader is encouraged to explore!

- Stochastic processes: simulating and analysing systems that evolve over time. That is, instead of having independent samples, our random variables are *dependent*. Discrete event simulation is one of the most important methodologies in this area.
- Markov Chain Monte-Carlo: the simulation technique that underpins modern Bayesian statistics.

- Stochastic optimisation: using a stochastic (random) process to optimise a function. Techniques include simulated annealing, genetic algorithms, cross-entropy, ant-heaps, and many more.
- Bootstrapping: a very clever statistical technique for extracting information from a sample by resampling.
- Meta-modelling: using a simpler but faster simulation to approximate a complex but slow simulation.
- Perfect simulation: how to reach an asymptotic limit in a finite amount of time.

## 21.2 Epidemics

The science of epidemiology, the study of the spread of disease, includes mathematical/statistical models of how disease spreads. In this section we look at some of these models and investigate their behaviour using simulation.

### *21.2.1 SIR model*

SIR stands for Susceptible, Infected, and Removed. In this model we suppose that individuals can be one of three types: susceptible if they have not yet caught the disease, infected if they currently have the disease, and removed if they have had the disease and have since recovered (and are now immune) or died. In our following descriptions, we will use the type labels—susceptible, infected, and removed—as shorthand to describe individuals of that type. We measure time in discrete steps. At each time step, each infected can infect susceptibles or can recover/die, at which point the infected is removed.

Let $S(t)$, $I(t)$ and $R(t)$ be the number of susceptible, infected and removed individuals at time $t$. At each time step each infected has probability $\alpha$ of infecting each susceptible. (This assumes that each infected has equal contact with all susceptibles. This is called a *mixing* assumption.) At the end of each time step, after having had a chance to infect people, each infected has probability $\beta$ of being removed.

We take initial conditions

$$\begin{aligned} S(0) &= N; \\ I(0) &= 1; \\ R(0) &= 0. \end{aligned}$$

Note that the total population is $N + 1$ and this remains fixed. That is $S(t) + I(t) + R(t) = N + 1$ for all $t$.

Each time step $t$ the chance that a susceptible remains uninfected is $(1-\alpha)^{I(t)}$.

That is, each infected must fail to pass on the infection to the susceptible. Thus,

$$S(t+1) \sim \text{binom}(S(t), (1-\alpha)^{I(t)}).$$

As each infected has a chance $\boldsymbol{\beta}$ of being removed, we have

$$R(t+1) \sim R(t) + \text{binom}(I(t), \boldsymbol{\beta}).$$

Given $S(t+1)$ and $R(t+1)$ we get $I(t+1)$ from the total population

$$I(t+1) = N + 1 - R(t+1) - S(t+1).$$

These rules are enough to write a simulation of an SIR process.

```
# program spuRs/resources/scripts/SIRsim.r

SIRsim <- function(a, b, N, T) {
  # Simulate an SIR epidemic
  # a is infection rate, b is removal rate
  # N initial susceptibles, 1 initial infected, simulation length T
  # returns a matrix size (T+1)*3 with columns S, I, R respectively
  S <- rep(0, T+1)
  I <- rep(0, T+1)
  R <- rep(0, T+1)
  S[1] <- N
  I[1] <- 1
  R[1] <- 0
  for (i in 1:T) {
    S[i+1] <- rbinom(1, S[i], (1 - a)^I[i])
    R[i+1] <- R[i] + rbinom(1, I[i], b)
    I[i+1] <- N + 1 - R[i+1] - S[i+1]
  }
  return(matrix(c(S, I, R), ncol = 3))
}
```

In Figure 21.1 we plot $S(t)$, $I(t)$, and $R(t)$ for four separate simulations, with $\alpha = 0.0005$ and $\boldsymbol{\beta} = 0.1, 0.2, 0.3$, and $0.4$. We see that as $\boldsymbol{\beta}$ increases, the size of the epidemic decreases.

To see the range of behaviour possible for a single choice of $\alpha$ and $\boldsymbol{\beta}$, we plot several realisations of the simulation on the same graph: see Figure 21.2. We see that epidemics either die out quickly or else grow to be quite large.

It would be nice to know exactly how $\alpha$ and $\boldsymbol{\beta}$ affect the size of an epidemic. Using simulation we can estimate $\mathbb{E}S(T)$ for different values of $\alpha$ and $\boldsymbol{\beta}$ and see how it varies. The following program does this for $\alpha \in [0.0001, 0.001]$ and $\beta \in [0.1, 0.5]$ and plots the results on a 3D-graph. (See Section 7.7 for guidance on 3D-plotting.) The output is given in Figure 21.3.

```
# program spuRs/resources/scripts/SIR_grid.r
# discrete SIR epidemic model
#
```

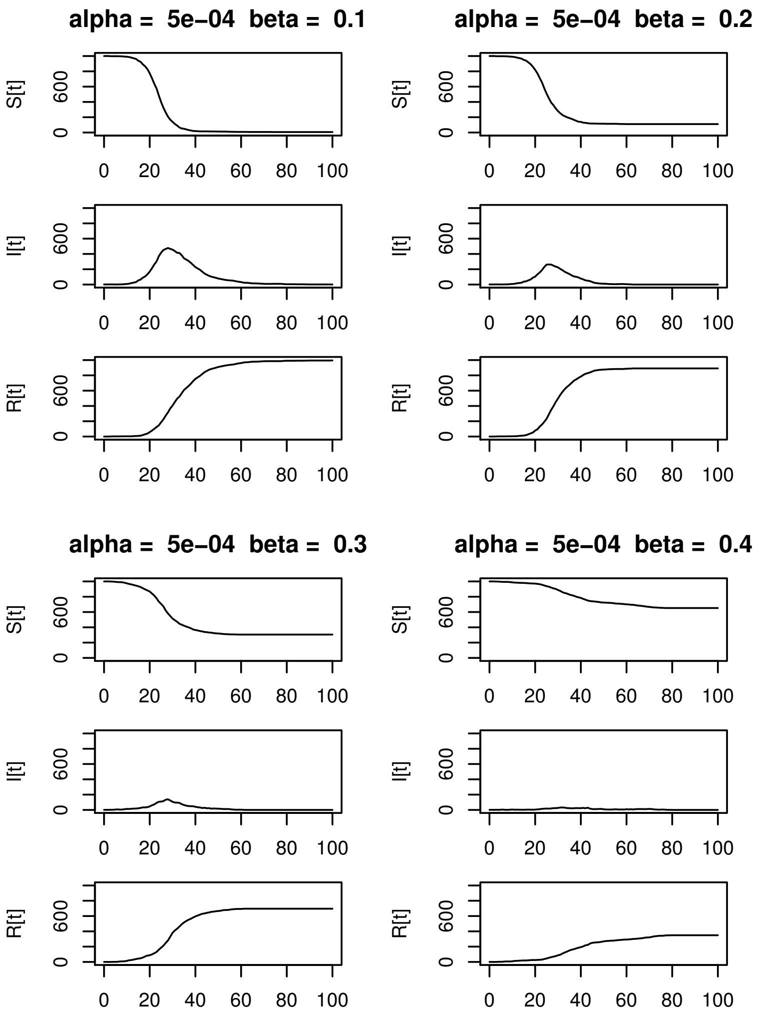

Figure 21.1 *Simulations of an SIR epidemic with* $\alpha = 0.0005$ *and* $\boldsymbol{\beta} = 0.1, 0.2, 0.3,$ *and* 0.4.

```
# initial susceptible population N
# initial infected population 1
# infection probability a
# removal probability b
#
# estimates expected final population size for different values of
# the infection probability a and removal probability b
# we observe a change in behaviour about the line Na = b
```

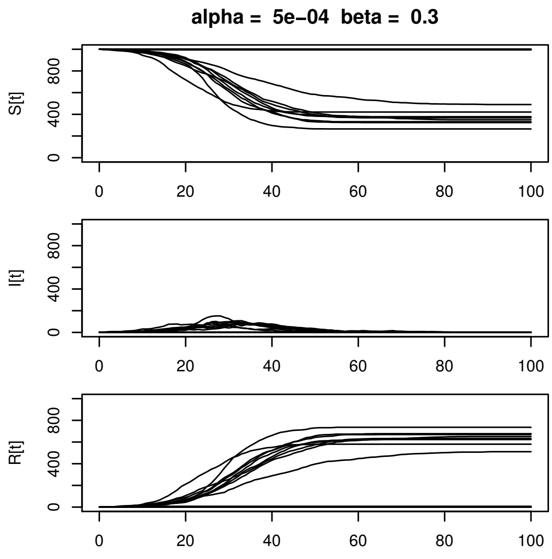

Figure 21.2 *Twenty realisations of an SIR epidemic with* $\alpha = 0.0005$ *and* $\beta = 0.3$.

```
# (Na is the expected number of new infected at time 1 and
# b is the expected number of infected who are removed at time 1)

SIR <- function(a, b, N, T) {
  # simulates SIR epidemic model from time 0 to T
  # returns number of susceptibles, infected and removed at time T
  S <- N
  I <- 1
  R <- 0
  for (i in 1:T) {
    S <- rbinom(1, S, (1 - a)^I)
    R <- R + rbinom(1, I, b)
    I <- N + 1 - S - R
  }
  return(c(S, I, R))
}

# set parameter values
N <- 1000
T <- 100
a <- seq(0.0001, 0.001, by = 0.0001)
b <- seq(0.1, 0.5, by = 0.05)

n.reps <- 400 # sample size for estimating E S[T]
f.name <- "SIR_grid.dat" # file to save simulation results
```

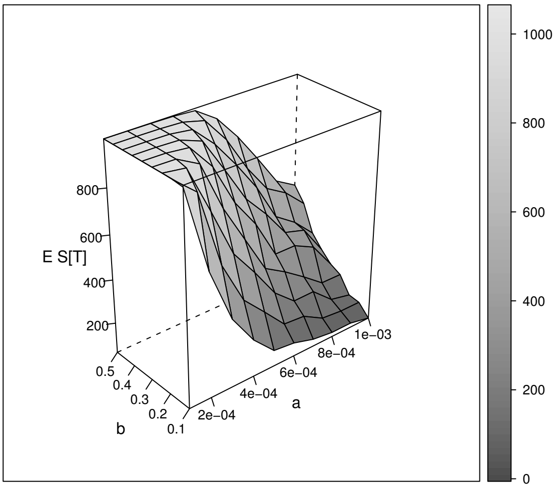

Figure 21.3 *Average epidemic size for various infection rates $\alpha$ and removal rates $\boldsymbol{\beta}$.*

```
# estimate E S[T] for each combination of a and b
write(c("a", "b", "S_T"), file = f.name, ncolumns = 3)
for (i in 1:length(a)) {
  for (j in 1:length(b)) {
    S.sum <- 0
    for (k in 1:n.reps) {
      S.sum <- S.sum + SIR(a[i], b[j], N, T)[1]
    }
    write(c(a[i], b[j], S.sum/n.reps), file = f.name,
      ncolumns = 3, append = TRUE)
  }
}

# plot estimates in 3D
g <- read.table(f.name, header = TRUE)
library(lattice)
print(wireframe(S_T ~ a*b, data = g, scales = list(arrows = FALSE),
                aspect = c(.5, 1), drape = TRUE,
                xlab = "a", ylab = "b", zlab = "E S[T]"))
```

We observe a change in behaviour about the line $N\alpha = \boldsymbol{\beta}$. $N\alpha$ is the expected number of new infected at time 1 and $\boldsymbol{\beta}$ is the expected number of infected who are removed at time 1. When $N\alpha > \boldsymbol{\beta}$ then we nearly always get a big epidemic, but when $N\alpha \leq \boldsymbol{\beta}$ the size of the epidemic drops away sharply.

For more insight into why this threshold occurs, we look at a class of models called *branching processes.*

### *21.2.2 Branching processes*

An epidemic has the potential to be large if, in its early stages, $\mathbb{E}$(new infected) $>$ $\mathbb{E}$(infected removed). For a general epidemic, calculating $\mathbb{E}$(new infected) is difficult because individuals interact:

- Finite population size means infected individuals are 'competing' for individuals to infect;
- Spatial restrictions restrict contact between infected and susceptible.

The SIR model ignores spatial interactions but does model the finite population. Branching processes ignore the finite population restriction as well. This results in a simpler but hopefully still useful model. The branching process can be viewed as a model for the early stages of an epidemic.

Branching processes are typically described in terms of births and population growth rather than infection. Let $Z_n$ be the size of the population at generation/time $n$. At each time step every individual independently gives birth to a random number of offspring, with distribution $X$, then dies. (You can include the case where the individual does not die by adding 1 to $X$.) Put $Z_0 = 1$ then we have

$$Z_{n+1} = X_{n,1} + \cdots + X_{n,Z_n},$$

where $X_{n,i}$ is the $i$-th family size in generation $n$. The $X_{n,i}$ are iid with the same distribution as $X$.

If you just look at the infected, then the first step of an SIR epidemic is the same as the first step of a branching process, with $X_{0,1} = A + B$ where $A \sim \text{binom}(N, \alpha)$ are the new infected and $B \sim \text{binom}(1, 1 - \boldsymbol{\beta})$ is 1 if the initial infected is not removed and 0 otherwise. Note that $\mathbb{E}X = N\alpha + 1 - \boldsymbol{\beta}$ so the condition for an epidemic to grow, $N\alpha > \boldsymbol{\beta}$, is equivalent to $\mathbb{E}X > 1$.

Here is some code for simulating and plotting a branching process. It makes use of the construct `...` for passing a variable number of inputs to a function.

```
# Program spuRs/resources/scripts/bp.r
# branching process simulation

bp <- function(gen, rv.sim, ...) {
  # population of a branching process from generation 0 to gen
  # rv.sim(n, ...) simulates n rv's from the offspring distribution
  # Z[i] is population at generation i-1; Z[1] = 1
  Z <- rep(0, gen+1)
  Z[1] <- 1
  for (i in 1:gen) {
    if (Z[i] > 0) {
      Z[i+1] <- sum(rv.sim(Z[i], ...))
    }
  }
  return(Z)
}

bp.plot <- function(gen, rv.sim, ..., reps = 1, logplot = TRUE) {
  # simulates and plots the population of a branching process
  # from generation 0 to gen; rv.sim(n, ...) simulates n rv's
  # from the offspring distribution
  # the plot is repeated reps times
  # if logplot = TRUE then the population is plotted on a log scale
  # Z[i,j] is population at generation j-1 in the i-th repeat
  Z <- matrix(0, nrow = reps, ncol = gen+1)
  for (i in 1:reps) {
    Z[i,] <- bp(gen, rv.sim, ...)
  }
  if (logplot) {
    Z <- log(Z)
  }
  plot(c(0, gen), c(0, max(Z)), type = "n", xlab = "generation",
    ylab = if (logplot) "log population" else "population")
  for (i in 1:reps) {
    lines(0:gen, Z[i,])
  }
  return(invisible(Z))
}
```

Figure 21.4 gives some sample output where we took $X \sim \text{binom}(2, 0.6)$. There are 20 simulations over 20 generations. Note that in half the simulations the population has died out, in the other half it appears to be growing exponentially. The command used was `bp.plot(20, rbinom, 2, .6, 20, logplot = F)`.

What is the relationship between the offspring distribution $X$ and the growth of the process? To investigate this question we fixed $T$ then used simulation to estimate $\log \mathbb{E}Z_T$ for a number of different $X$ and then plotted this against $\mu = \mathbb{E}X$. We put $T = 50$ and $X \sim \text{binom}(2, p)$ for $p \in [.3, .6]$. Here is the code we used; the output is given in Figure 21.5. Note that values of $\log(0)$ $(= -\infty)$ are not plotted.

```
# program spuRs/resources/scripts/bp_grid.r

bp.sim <- function(gen, rv.sim, ...) {
  # population of a branching process at generation gen
```

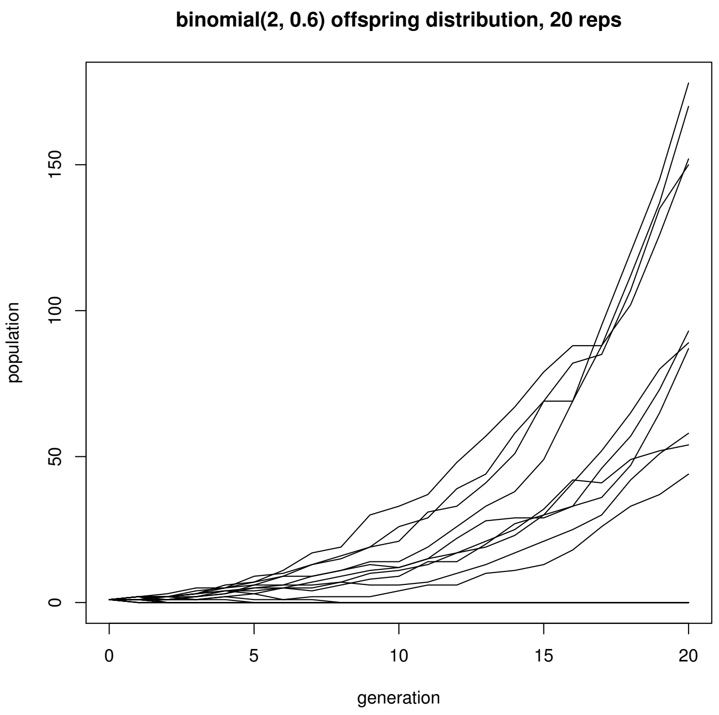

Figure 21.4 *Several realisations of a branching process.*

```
  # rv.sim(n, ...) simulates n rv's from the offspring distribution
  Z <- 1
  for (i in 1:gen) {
    if (Z > 0) {
      Z <- sum(rv.sim(Z, ...))
    }
  }
  return(Z)
}


# set parameter values
gen <- 50
size <- 2
prob <- seq(0.3, 0.6, by = 0.01)
n.reps <- 100 # sample size for estimating E Z

# estimate E Z for each value of prob
mu <- rep(0, length(prob))
Z.mean <- rep(0, length(prob))
```

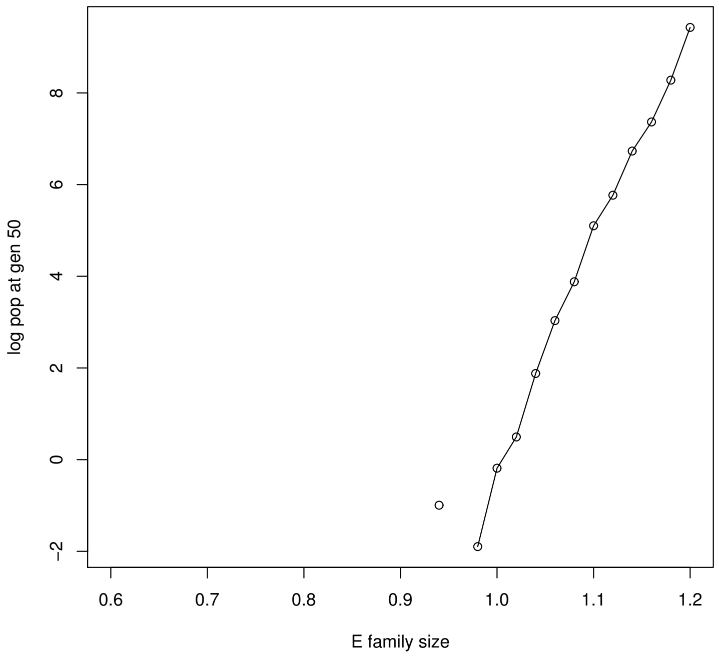

Figure 21.5 *Expected population at time $T$ agianst the expected family size.*

```
for (i in 1:length(prob)) {
  Z.sum <- 0
  for (k in 1:n.reps) {
    Z.sum <- Z.sum + bp.sim(gen, rbinom, size, prob[i])
  }
  mu[i] <- size*prob[i]
  Z.mean[i] <- Z.sum/n.reps
}

# plot estimates
# note that values of log(0) (= -infinity) are not plotted
plot(mu, log(Z.mean), type = "o",
     xlab = "E family size", ylab = paste("log pop at gen", gen))
```

There is a quite convincing linear relationship between $\mathbb{E}X$ and $\log \mathbb{E}Z_T$, with an $x$-intercept at 1. That is, for some constant $c = c(T)$, we have

$$\begin{aligned} \log \mathbb{E}Z_T &\approx c(\mathbb{E}X - 1) \\ \mathbb{E}Z_T &\approx e^{c(\mathbb{E}X-1)}. \end{aligned}$$

Thus if $\mathbb{E}X > 1$ then $\mathbb{E}Z_T > 1$ but if $\mathbb{E}X < 1$ then $\mathbb{E}Z_T < 1$.

Because the branching process is a relatively simple model, we can prove some exact results for it. In particular it is possible to show that

$$\mathbb{E}Z_n = (\mathbb{E}X)^n. \tag{21.1}$$

So if $\mathbb{E}X > 1$ the process grows exponentially (on average), while if $\mathbb{E}X < 1$ then it dies out exponentially fast (on average). This agrees with our previous observations of the SIR process.

A useful exercise is to verify the relationship (21.1) using simulation.

### *21.2.3 Forest fire model*

The forest fire model incorporates spatial interactions. Like the SIR model we suppose that we have a population made up of susceptible (unburnt), infected (on fire), and removed (burnt out) individuals. The difference is that the individuals are placed on a grid and an infected individual can only infect a susceptible individual if they are neighbours. We define the neighbours of a point $(x, y)$ to be the eight points $(x-1, y-1)$, $(x-1, y)$, $(x-1, y+1)$, $(x, y-1)$, $(x, y+1)$, $(x+1, y-1)$, $(x+1, y)$, and $(x+1, y+1)$ (smaller or larger neighbourhoods can also be considered).

We take time in discrete steps. At each step an infected individual has a probability $\alpha$ of infecting each of its susceptible neighbours. Thus for a susceptible individual, the probability of remaining uninfected is $(1-\alpha)^x$ where $x$ is the number of infected neighbours. After having had a chance to infect its neighbours, an individual is removed with probability $\boldsymbol{\beta}$.

We restrict our forest fire to a grid of size $N \times N$. Let $X_t$ be a matrix of size $N \times N$ representing the population at time $t$. We put

$$X_t(i,j) = \begin{cases} 2 & \text{if the individual at } (i,j) \text{ is susceptible;} \\ 1 & \text{if the individual at } (i,j) \text{ is infected;} \\ 0 & \text{if the individual at } (i,j) \text{ is removed.} \end{cases}$$

Here is some code for simulating the forest fire model and printing the results. An example of the output is provided in Figure 21.6. If you play around with this for a while you will see that we still see a threshold below which the fire rarely gets going but above which there is a chance that it can grow quite large. Again there is a balance between how fast new infections appear and how fast infected individuals are removed.

```
# program: spuRs/resources/scripts/forest_fire.r
# forest fire simulation
rm(list = ls())

neighbours <- function(A, i, j) {
  # calculate number of neighbours of A[i,j] that are infected
  # we have to check for the edge of the grid
  nbrs <- 0
  # sum across row i - 1
  if (i > 1) {
    if (j > 1) nbrs <- nbrs + (A[i-1, j-1] == 1)
    nbrs <- nbrs + (A[i-1, j] == 1)
    if (j < ncol(A)) nbrs <- nbrs + (A[i-1, j+1] == 1)
  }
  # sum across row i
  if (j > 1) nbrs <- nbrs + (A[i, j-1] == 1)
  nbrs <- nbrs + (A[i, j] == 1)
  if (j < ncol(A)) nbrs <- nbrs + (A[i, j+1] == 1)
  # sum across row i + 1
  if (i < nrow(A)) {
    if (j > 1) nbrs <- nbrs + (A[i+1, j-1] == 1)
    nbrs <- nbrs + (A[i+1, j] == 1)
    if (j < ncol(A)) nbrs <- nbrs + (A[i+1, j+1] == 1)
  }
  return(nbrs)
}

forest.fire.plot <- function(X) {
  # plot infected and removed individuals
  for (i in 1:nrow(X)) {
    for (j in 1:ncol(X)) {
      if (X[i,j] == 1) points(i, j, col = "red", pch = 19)
      else if (X[i,j] == 0) points(i, j, col = "grey", pch = 19)
    }
  }
}

forest.fire <- function(X, a, b, pausing = FALSE) {
  # simulate forest fire epidemic model
  # X[i, j] = 2 for susceptible; 1 for infected; 0 for removed

  # set up plot
  plot(c(1,nrow(X)), c(1,ncol(X)), type = "n", xlab = "", ylab = "")
  forest.fire.plot(X)

  # main loop
  burning <- TRUE
  while (burning) {
    burning <- FALSE
    # check if pausing between updates
    if (pausing) {
      input <- readline("hit any key to continue")
    }

    # update
    B <- X
    for (i in 1:nrow(X)) {
      for (j in 1:ncol(X)) {
        if (X[i, j] == 2) {
          if (runif(1) > (1 - a)^neighbours(X, i, j)) {
            B[i, j] <- 1
          }
        } else if (X[i, j] == 1) {
          burning <- TRUE
          if (runif(1) < b) {
            B[i, j] <- 0
          }
        }
      }
    }
    X <- B

    # plot
    forest.fire.plot(X)
  }

  return(X)
}

# spark
set.seed(3)
X <- matrix(2, 21, 21)
X[11, 11] <- 1
# big fires
#X <- forest.fire(X, .1, .2, TRUE)
X <- forest.fire(X, .2, .4, TRUE)
# medium fires
#X <- forest.fire(X, .07, .2, TRUE)
#X <- forest.fire(X, .1, .4, TRUE)
# small fires
#X <- forest.fire(X, .05, .2, TRUE)
#X <- forest.fire(X, .07, .4, TRUE)
```

Clearly as $\alpha$ increases and/or $\beta$ decreases, the chance of a large fire will increase. Like the SIR and branching process models, we imagine that there will be a threshold above which large fires become much more likely. For example, suppose that the fire is burning along a straight front. Along the front each susceptible tree is adjacent to three burning trees, so the probability of catching on fire is $1-(1-\alpha)^3$. Thus, given burning trees are removed with probability $\beta$, we might conjecture that the fire will grow if $1-(1-\alpha)^3 > \beta$.

As it turns out, this conjecture understates the chance of a large fire. The reason is that fire fronts are not straight, and an irregular front will move faster than a straight front. Even a front that starts straight will quickly

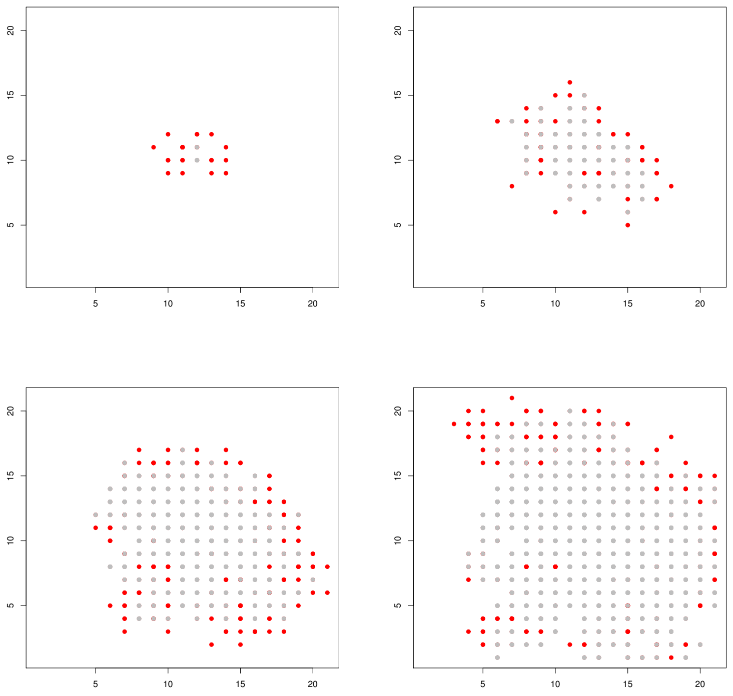

Figure 21.6 *Simulation of a forest fire epidemic at times 5, 10, 15, and 20. Infected individuals are dark grey and removed individuals light grey. Here* $\alpha = 0.2$ *and* $\beta = 0.4$*; we started with a single infected individual in the centre of the grid.*

contort, which one can easily see in the simulation, using the following initial condition.

```
X <- matrix(2, 21, 21)
X[21,] <- 1
```

## 21.3 Inventory

To meet demand in time and compete in the market, a company needs to keep stock in hand. The purpose of inventory theory is to determine rules or policies that minimise the cost of running an inventory system, while meeting customer demand. The following are possible costs associated with an inventory system

1. *Ordering and setup cost:* This includes the cost of paperwork and billing associated with an order, and may include overheads on the cost of delivery. If the product is produced internally, this cost may also include the cost of setting up and shutting down a machine in a production system.

The *Lead time* is the length of time between when an order is placed and when the order arrives.

2. *Purchasing cost:* For outsourced products this will include per-item transportation costs as well as the cost of the product. For goods produced internally, this includes the cost of raw materials and labour.
3. *Holding cost:* This is the cost of holding one unit of inventory for one period of time. If the period is one year then it is the annual holding cost. This cost can include insurance costs, the cost of renting space, security costs, loss due to spoilage, and the effects of inflation.
4. *Shortage cost:* When a demand cannot be met in time, a shortage is said to have occurred. There are two possible cases:

   (a) The customers accept delivery on a later date, which is called a *backlogged demand*;

   (b) The customers refuse to have the delivery on a later date, which is called a *lost sale.*

   In the second case the shortage cost is primarily the lost revenue. In the first case the shortage cost includes penalties paid for late delivery. In both cases the shortage cost can also include a component that represents lost future sales due to the lack of service shortage represents.

### *21.3.1 Continuous Review Inventory Model*

The continuous review inventory model makes the following assumptions.

1. The inventory system is under *continuous review*, which means that sales are recorded when they occur so that the level of inventory in the system $I(t)$ is known at all times $t$.
2. The demand is a Poisson process with a rate of $D$ items per year.
3. The lead time $L$ is a known constant.
4. There is an ordering cost of $K$ and a price per unit of $p$.
5. The unit holding cost is $h$ per year.
6. Shortage results in lost sales, with a shortage cost of $s$ per item.

We suppose that the inventory policy (or the ordering policy) is a so-called $(q, r)$ policy. That is, when the inventory level is $r$ (reorder point), an order of size $q$ is placed, which will arrive after a lead time of $L$. Our objective is to choose $q$ and $r$ to minimise cost.

The expected demand over a lead time period is $LD$. Hence, if we reorder when $I(t) = r$ the expected minimum inventory level will be $m = r - LD$. The quantity $m$ is called the *safety stock*. We will assume that $m \geq 0$, that is $r \geq LD$, and that $q \geq r$.

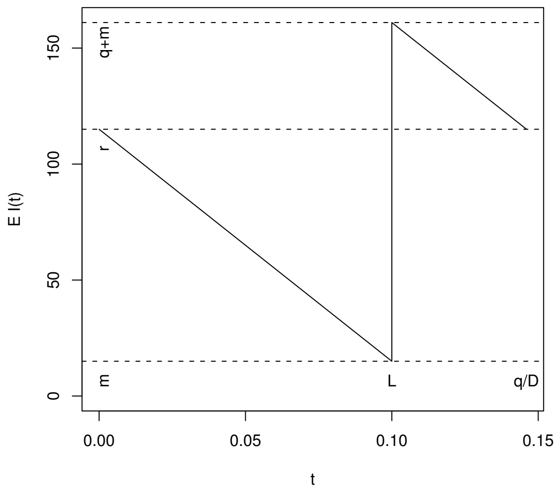

Figure 21.7 *Expected inventory level under a $(q, r)$ policy, over a single cycle.*

We would like to estimate $c(q, r)$, which is the expected cost per unit time of running the system (that is, the annual cost), and then choose $q$ and $r$ to minimise it. However we have to be careful what we mean by 'cost per unit time', because the costs change as the level of inventory changes. The way around this problem is to consider *cycles*. Define a cycle to be the time from one reorder point to the next, when the stock is at level $r$. With a little thought you should see that these cycle lengths are *independent*.[1] Let $C$ be the running costs and $T$ the length of a single cycle, then we define

$$c(q, r) = \mathbb{E}\left(\frac{C}{T}\right) \approx \frac{\mathbb{E}C}{\mathbb{E}T}.$$

In Figure 21.7 we plot the expected inventory level over a single cycle. The expected demand is $D$ per year. Thus the graph of $\mathbb{E}I(t)$ will decrease from $r$ to $m$ with a constant slope of $-D$, jump to $q + m$, then decrease to $r$ with slope $-D$. We see immediately that $\mathbb{E}T = q/D$.

To estimate $\mathbb{E}C$ we split the cost into four parts—holding cost, ordering cost, purchasing cost, and shortage cost—and consider each in turn.

1. Put $I(0) = r$. Noting that $m = r - LD$, the expected holding cost over a

[1] In fact, our inventory system is an example of a *renewal process*, and the reorder points are known as *renewal times*.

single cycle is

$$\begin{aligned}\mathbb{E}\int_0^T hI(t)\,dt &\approx h\int_0^{q/D}\mathbb{E}I(t)\,dt\\ &= h\int_0^{q/D}(m+Dt)\,dt\\ &= h\left(\frac{q^2}{2D}+\frac{(r-LD)q}{D}\right).\end{aligned}$$

Here we have approximated $T$ by $\mathbb{E}T$.

2. The ordering cost per cycle is exactly $K$.
3. The purchasing cost per cycle is exactly $pq$.
4. To calculate the shortage cost we note that demand during the lead time has a Poisson$(DL)$ distribution. To simplify things we will approximate the demand during the lead time by a continuous distribution with probability density function $f(x)$. Given this the expected shortage during the lead time will be

$$n(r)=\int_r^\infty (x-r)f(x)\,dx$$

and so the expected shortage cost will be $sn(r)$.

Putting these together, the expected cost per unit time under a $(q,r)$ policy is (approximately)

$$c(q,r)=h\left(r-LD+\frac{q}{2}\right)+\frac{KD}{q}+pD+\frac{sDn(r)}{q}.$$

*Theorem* A necessary condition for $c(q,r)$ to be minimised is that $q$ and $r$ satisfy the equations

$$q=\sqrt{\frac{2D(K+sn(r))}{h}}\quad\text{and}\quad 1-F(r)=\frac{qh}{sD}, \tag{21.2}$$

where $F(r)=\int_0^r f(x)dx$.

*Proof.* We note that

$$\frac{\partial c(q,r)}{\partial q}=\frac{h}{2}-\frac{KD}{q^2}-\frac{sDn(r)}{q^2}$$

and

$$\frac{\partial c(q,r)}{\partial r}=h+\frac{sDn'(r)}{q}$$

where

$$\begin{aligned}n'(r) &= \frac{d}{dr}\left(\int_r^\infty xf(x)dx-r\int_r^\infty f(x)dx\right)\\ &= -rf(r)-\int_r^\infty f(x)dx+rf(r)\end{aligned}$$

$$= \quad F(r) - 1.$$

Setting $\partial c(q,r)/\partial q = \partial c(q,r)/\partial r = 0$ gives the result.

The *service level* $\alpha$ is the probability of not running out of stock in any given cycle, namely $F(r)$. In practice, rather than solve the above equations for $q$ and $r$, what practitioners often do is specify the required service level beforehand, based on perceived customer requirements. Typically we take $\alpha = 0.95$ or 0.99. Having specified $\alpha$ and thus $r$, the expected cost is now a function of $q$ alone, which we minimise in the usual way. We have

$$c(q) = h\left(r - LD + \frac{q}{2}\right) + \frac{KD}{q} + pD + \frac{sDn(r)}{q}.$$

From this the optimal value of $q$ is

$$q^* = \sqrt{\frac{2D(K + sn(r))}{h}} \approx \sqrt{\frac{2KD}{h}},$$

noting that for $\alpha$ close to 1, $n(r)$ will be small. This last value is known as the *Economic Order Quantity* (EOQ) in the inventory literature.

For example, suppose that $D = 1000$ per year, $L = 0.1$ years, $K = 1000$, $p = 100$, $h = 100$ per year, and $s = 200$. Let $X$ be the demand during the lead time, then $X \sim \text{pois}(100) \approx N(100, 100)$. Using the normal approximation we have $f(x) = \frac{1}{\sqrt{200\pi}} \exp(-(x - 100)^2/200)$.

If we specify a service level of $\alpha = 0.95$, then $r$ satisfies $\mathbb{P}(X \le r) = 0.95$. To calculate the left-hand side we can use Simpson's rule for numerical integration. To solve the equation we can use the Newton-Raphson algorithm. Let $F(x) = \int_{-\infty}^{x} f(u)du = 0.5 + \int_{100}^{x} f(u)du$ (the second form avoids having an infinite domain to integrate over).

```
> rm(list = ls())
> source("../scripts/simpson.r")
> f <- function(x) exp(-(x - 100)^2/200)/sqrt(200 * pi)
> F <- function(x) {
+     if (x > 100)
+         return(0.5 + simpson(f, 100, x))
+     else if (x < 100)
+         return(0.5 - simpson(f, x, 100))
+     else return(0.5)
+ }
> source("../scripts/newtonraphson.r")
> g <- function(r) c(F(r) - 0.95, f(r))
> r <- newtonraphson(g, 100)
```

```
At iteration 1 value of x is: 111.2798
At iteration 2 value of x is: 115.0524
At iteration 3 value of x is: 116.3077
At iteration 4 value of x is: 116.4469
At iteration 5 value of x is: 116.4485
At iteration 6 value of x is: 116.4485
Algorithm converged
```

Rounding to the nearest integer we get $r = 116$. Using the EOQ to approximate $q$ we have $q = \sqrt{2KD/h} = 141$ (rounding to the nearest integer).

How good are these values of $q$ and $r$? To find out, we solve the Equations (21.2) and compare the answers.

Let $A$ be any $2 \times 2$ non-singular matrix, then the optimal $(q, r)^T$ is a fixed point of the equation

$$G\begin{pmatrix} q \\ r \end{pmatrix} = A\begin{pmatrix} q^2h - 2D(K + s\,n(r)) \\ (1 - F(r))sD - qh \end{pmatrix} + \begin{pmatrix} q \\ r \end{pmatrix}.$$

If we can choose $A$ so that $G$ is a *contraction*, then we can obtain the fixed point by iterating $G$. We say $G$ is a contraction if there exists $\delta \in (0, 1)$ such that, for any vectors $\mathbf{x}$ and $\mathbf{y}$, $\|G(\mathbf{x}) - G(\mathbf{y})\| \le \delta\|\mathbf{x} - \mathbf{y}\|$. In this case, putting $\mathbf{x}_n = G(\mathbf{x}_{n-1})$ we have

$$\begin{aligned} \|\mathbf{x}_{n+1} - \mathbf{x}_n\| &= \|G(\mathbf{x}_n) - G(\mathbf{x}_{n-1})\| \\ &\le \delta\|\mathbf{x}_n - \mathbf{x}_{n-1}\| \\ &\le \delta^n\|\mathbf{x}_1 - \mathbf{x}_0\| \;\rightarrow\; 0 \text{ as } n \to \infty. \end{aligned}$$

It follows that for any $k$, $\|\mathbf{x}_{n+k} - \mathbf{x}_n\| \le \delta^n\|\mathbf{x}_1 - \mathbf{x}_0\|/(1 - \delta)$, and thus that $\mathbf{x}_n$ converges, to $\mathbf{x}_*$ say (this is Cauchy's convergence criterion). Since $G$ is continuous,

$$\mathbf{x}_* = \lim_{n\to\infty} \mathbf{x}_{n+1} = \lim_{n\to\infty} G(\mathbf{x}_n) = G(\lim_{n\to\infty} \mathbf{x}_n) = G(\mathbf{x}_*).$$

That is, $\mathbf{x}_*$ is a fixed point of $G$.

To calculate $G$ we need $n(r) = \int_r^\infty (x - r)f(x)dx$. Using the change of variables $y = (x - 100)^2/2$, we can rewrite $n$ as

$$n(r) = \sqrt{50/\pi}\exp(-(r - 100)^2/200) - (r - 100)(1 - F(r)).$$

This reformulation has the advantage that it does not require an integral over an infinite domain.

After some trial and error, it turns out that a suitable $A$ is

$$\begin{pmatrix} -1/50,000 & 0 \\ 0 & 1/50,000 \end{pmatrix}.$$

Using $(141, 116)^T$ as the starting point for our iteration, $G$ does indeed converge to a fixed point:

```
> n <- function(r) {
+   return(sqrt(50/pi)*exp(-(r - 100)^2/200) - (r - 100)*(1 - F(r)))
+ }
```

```
> G <- function(x) {
+   q <- x[1]
+   r <- x[2]
+   A <- matrix(c(-1, 0, 0, 1), 2, 2)/50000
+   return( A %*% c(100*q^2 - 2000*(1000 + 200*n(r)),
+                   (1 - F(r))*200000 - 100*q) + c(q, r) )
+ }
> tol <- 1e-3
> x <- c(141, 116)
> x.diff <- 1
> while (x.diff > tol) {
+   x.old <- x
+   x <- G(x)
+   x.diff <- sum(abs(x - x.old))
+ }
> x

         [,1]
[1,] 145.9390
[2,] 114.5481
```

Rounding to the nearest integer we get $q = 146$ and $r = 115$. Comparing our two solutions, first note that the service level corresponding to $r = 115$ is $F(r) = 0.933$ (to 3 significant figures), a little lower than the level of 0.95 we initially assumed. Calculating the annual cost we have

$$c(141, 116) = 116{,}071.9, \quad c(146, 115) = 116{,}050.8.$$

So in this case using the EOQ gave a reasonable approximation to the optimal value of $q$.

## *21.3.2 Simulated inventory level*

The cost per unit time $c(q, r)$, derived in the previous section, incorporated some simplifying assumptions. In particular we assumed

$$\mathbb{E}\left(\frac{C}{T}\right) \approx \frac{\mathbb{E}C}{\mathbb{E}T}$$

and

$$\mathbb{E}\int_0^T hI(t)\,dt \approx h\int_0^{q/D} \mathbb{E}I(t)\,dt.$$

We also assumed that the demand during the lead time could be approximated by a continuous distribution.

To judge how much of an effect these simplifying assumptions have on $c(q, r)$, we use simulation to provide an independent estimate. We will use a technique called *discrete event simulation.*

Let $I(t)$ be the level of stock (that is, inventory) at time $t$, and $c(t)$ the accumulated costs at time $t$. The triple $(t, I(t), c(t))$ describes the *state* of our system. Discrete event simulation updates the state only when certain events occur. In our case the relevant events are *purchases* and the arrival of *new stock*.

Suppose that at the previous event the state was $(u, I(u), c(u))$ and that the next event after time $u$ happens at time $v$.

If the new event is a purchase then $I(v) = \max\{I(u) - 1, 0\}$. Updating the costs is more complex:

- Over the time interval $(u, v]$ the holding costs have increased by $I(u)(v-u)$;
- If $I(u) = 0$ then there will be a shortage cost of $s$;
- If $I(v) = r$ then there will be a reordering cost of $K + qp$.

If the new event is the arrival of new stock, then $I(v) = I(u) + q$ and $c(v) = c(u) + I(u)(v - u)$.

We maintain a list of events and when they will occur. We update this list every time an event occurs, by removing the event that has just occurred, and adding any new events we now know about. In our case, if the event at time $v$ is a purchase, then we generate a new purchase event at time $v + A$, where $A \sim \exp(D)$. That is, $A$ is the time between arrivals in a Poisson process of rate $D$. Moreover, if the stock level drops to $r$ then we generate a new stock arrival event at time $v + L$. The arrival of new stock does not trigger any new events. (By assumption $q > r$, so we know that it is never necessary to order new stock immediately.)

Once we have defined rules for updating the state for each type of event, and for generating new events, the simulation has the following simple form (pseudo-code):

```
initialise state and event list
while (stopping condition not met) {
  get next event
  if event type = a
    update state and event list
  else if event type = b
    update state and event list
  else ...
}
```

*The event list* We will implement the event list as a `list` in R. Each element will itself be a list, with two named elements: `type` and `time`. We will assume that the elements of the event list are ordered according to their `time` components.

Given this structure, to get the next event we just need the first element of the event list:

```
current.event <- event.list[[1]]
event.list <- event.list[-1]
```

Inserting a new event into the event list requires more work, as we need to preserve the ordering. Here is a function to do this for us:

```
add_event <- function(event.list, new.event) {
  # add new.event to event.list
  N <- length(event.list)
  if (N == 0) return(list(new.event))
  # find position n of new.event
  n <- 1
  while ((n <= N) && (new.event$time > event.list[[n]]$time)) {
    n <- n + 1
  }
  # add new.event to event.list
  if (n == 1) {
    event.list <- c(list(new.event), event.list)
  } else if (n == N + 1) {
    event.list <- c(event.list, list(new.event))
  } else {
    event.list <- c(event.list[1:(n-1)], list(new.event), event.list[n:N])
  }
  return(event.list)
}
```

In our case the event list will only ever contain the next purchase event and sometimes also the next stock arrival event.

Here is our program for simulating an inventory system. To simulate a single cycle we put $I(0) = r$ and $c(0) = 0$ and then run the simulation until the next time $I(t) = r$.

```
# program: spuRs/resources/scripts/inventory_sim.r

rm(list=ls())
set.seed(1939)
source("../scripts/add_event.r")

# inputs
# system parameters
D <- 1000
L <- 0.1
K <- 1000
p <- 100
h <- 100
s <- 200
# control parameters
q <- 146
r <- 115
```

```
# initialise system and event list
n <- 0  # number of events so far
t <- 0  # time
stock <- r
costs <- 0
event.list <- list(list(type = "purchase", time = rexp(1, rate = D)))
event.list <- add_event(event.list, list(type = "new stock", time = L))
# initialise stopping condition
time.to.stop <- FALSE
# simulation
while (!time.to.stop) {
  # get next event
  current.event <- event.list[[1]]
  event.list <- event.list[-1]
  n <- n + 1
  # update state and event list according to type of current event
  if (current.event$type == "purchase") {
    # update system state
    t[n+1] <- current.event$time
    if (stock[n] > 0) {  # reduce inventory, update holding costs
      costs[n+1] <- costs[n] + h*stock[n]*(t[n+1] - t[n])
      stock[n+1] <- stock[n] - 1
    } else {             # lost sale
      costs[n+1] <- costs[n] + s
      stock[n+1] <- stock[n]
    }
    # generate next purchase
    new.event <- list(type = "purchase", time = t[n+1] + rexp(1, rate = D))
    event.list <- add_event(event.list, new.event)
    # check for end of cycle
    if (stock[n+1] == r) {
      # order more stock
      new.event <- list(type = "new stock", time = t[n+1] + L)
      event.list <- add_event(event.list, new.event)
      costs[n+1] <- costs[n+1] + K + q*p
    }
  } else if (current.event$type == "new stock") {
    # update system state
    t[n+1] <- current.event$time
    costs[n+1] <- costs[n] + h*stock[n]*(t[n+1] - t[n])
    stock[n+1] <- stock[n] + q
  }
  # check stopping condition
  if (stock[n+1] == r) time.to.stop <- TRUE
}
```

It is worthwhile to plot the stock (inventory) level over a single cycle, and compare with the expected stock level, which was the basis of the analysis used in Section 21.3.1.

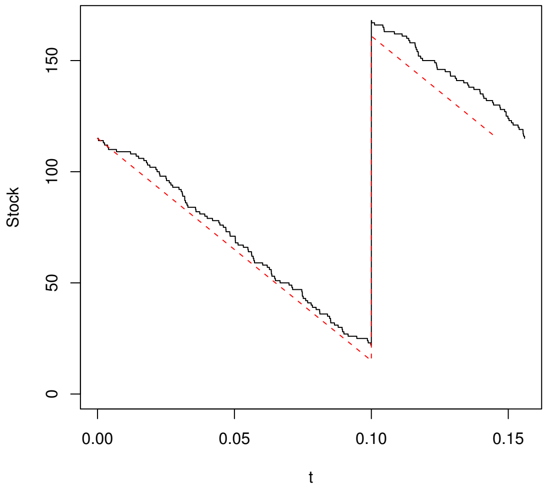

Figure 21.8 *Simulated and expected stock (inventory) level for a continuous review inventory model.*

```
plot(t, stock, type = "s", ylim=c(0, max(stock)))
lines(c(0, L, L, q/D), c(r, r-L*D, q+r-L*D, r), lty=2, col="red")
```

The output is given in Figure 21.8. We see that qualitatively the simulated stock level looks a lot like the expected stock level.

To estimate $c(q, r)$ we need to run the simulation for several cycles. The program `inventory_sim.r` incrementally updates the state vectors `t`, `stock` and `costs` at each event. This gives us a complete record of the process, but is too slow for simulating more than a few cycles. Thus to estimate $c(q, r)$ we rewrite the program so that it only keeps the current state, not the whole history. We also need to change the stopping condition, so that we stop after a fixed number of cycles. Finally, for each cycle we need to record the observed value of $C/T$. The rewritten program can be found as `inventory2_sim.r` in the `resources/scripts` directory within the `spuRs` archive.

Simulating 1000 cycles we obtained $\hat{c}(q, r) = 116,338.3$ with a 95% CI of (115,826.9, 116,849.8). Our approximation from Section 21.3.1 was 116,050.8, which sits comfortably in the confidence interval.

### *21.3.3 A two-stage inventory system*

Our approximation of $c(q, r)$ for a continuous review inventory system worked quite well. Unfortunately such an analysis becomes much harder for more complex systems, and we have to rely more on simulation.

Consider an inventory system with a retail store and a depot. The store sells items one at a time, keeps a small amount of inventory on site, and frequently orders replacement stock from the depot. The depot supplies batches of stock to the store, keeps a large amount of inventory, and infrequently orders large quantities of replacement stock. Delivery from the depot to the store should be quite quick, but the lead time for deliveries to the depot could be quite large. Such systems are used when storage at the store is expensive, but storage at the depot is cheap.

In practice a depot will often serve several stores, however we will restrict ourselves to a single store.

The parameters needed to describe this two-stage system are

- $D$ demand (at store);
- $L_1$, $L_2$ lead time for store and depot;
- $K_1$, $K_2$ ordering/delivery cost for store and depot;
- $p$ per item cost (depot only);
- $h_1$, $h_2$ holding cost per item per unit time at store and depot;
- $s$ shortage cost (at store);
- $q_1$, $q_2$ order quantities for store and depot;
- $r_1$, $r_2$ reorder point for store and depot.

Using discrete event simulation, we describe the state of the system using the variables

- Time $t$;
- Inventory at the store $I_1$;
- Inventory at the depot $I_2$;
- Cumulative cost $c$;

and we have the following events

- Purchase at the store;
- Stock arrives at the store;
- Stock arrives at the depot.

There is a complication to the two-stage system that does not appear in the simple continuous review model. It is possible that when the store orders stock from the depot, the depot is empty. We cannot treat this as a lost sale, rather the order has to be backlogged, then filled when the depot gets new stock. A convenient way to deal with backlogged orders is to create a new event

- Backlogged order

When the depot fails to fill an order we just create a backlogged order at some predetermined time $b$ in the future. That is, we wait time $b$ then try again.

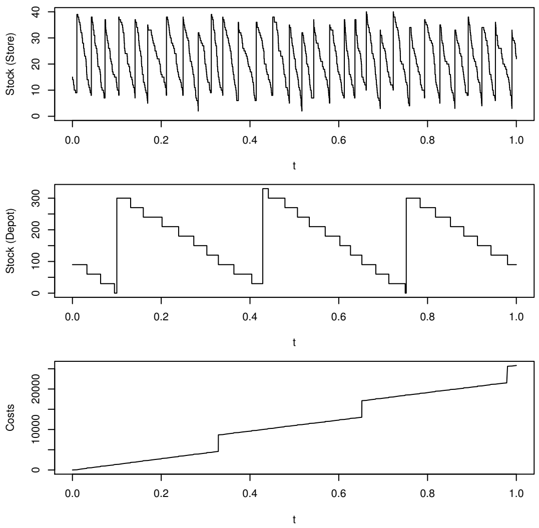

Figure 21.9 *A simulation of the two-stage inventory system, showing the level of stock (inventory) at the store and depot, and cumulative costs.*

*Pseudo-code* Rather than give a full implementation of the two-stage inventory system here, we will map out a suitable structure using pseudo-code. The interested reader can find a working version as `inventory_2stage_sim.r` in the `spuRs` archive, and some sample output is given in Figure 21.9, using a plausible set of parameter values.

The two-stage inventory system is still a renewal process, though the cycles are now more complex. Observe that when the inventory level at the depot reaches the reorder point $r_2$, it must be that the inventory level at the store has just reached $r_1$, because the only time we take stock from the depot is when it is ordered by the store. At this point we also know all there is to know about coming events: there will be a stock arrival at the store after a lead time of $L_1$; a stock arrival at the depot after a lead time of $L_2$; the time to the next purchase event is exponentially distributed, with rate $D$; and there will be no pending backlogged order, because we know the depot has just filled an order. Thus the point where the inventory levels $I_1$ and $I_2$ hit $r_1$ and $r_2$ is a renewal point, and marks the start/finish of independent cycles.

For this example, even though we have a renewal structure, instead of running the simulation for a given number of cycles, we have chosen to run it for a fixed length of time, $T$ say. The basic structure of our program is as follows:

```
# initialise state variables
t <- 0
```

```
I1 <- r1
I2 <- r2
c <- 0
# initialise event list
create empty event list
add stock_arrival_at_store event at time L1
add stock_arrival_at_depot event at time L2
add purchase event at time X ~ exp(D)
# run the simulation
while (t < T) {
  t.old <- t
  get next event from event list
  if (next event is a purchase) {
    # update state and event list for a purchase
    ...
  } else if (next event is a stock_arrival_at_store) {
    # update state and event list for a stock_arrival_at_store
    ...
  } else if (next event is a stock_arrival_at_depot) {
    # update state and event list for a stock_arrival_at_depot
    ...
  } else { # next event is a backlogged_order
    # update state and event list for a backlogged_order
    ...
  }
}
```

With each event we need to adjust the time and add accumulated holding costs to $c$, other changes to the state variables and event list depend on the event in question. We consider the purchase event first:

```
# update state and event list for a purchase
# update time
t <- new event time
# update holding costs
c <- c + h1*I1*(t - t.old) + h2*I2*(t - t.old)
# update stock level
if (I1 > 0) {
  I1 <- I1 - 1
} else {
  # incur shortfall cost
  c <- c + s
}
# check store reorder level
if (I1 == r1) {
  # order from depot
  ...
}
# schedule next purchase
add purchase event at time t + X where X ~ exp(D)
```

The process of making an order from the depot requires some thought, as it will affect the level of stock at the depot, which means we also need to check the depot reorder point. Moreover, if the depot does not have enough stock to fill the order, then we have to generate a backlogged order. We will assume that the level of stock at the depot is always a multiple of $q_1$, which means that $r_2$ must also be a multiple of $q_1$. The advantage of this assumption is that we know to reorder only when $I_2 = r_2$, rather than when $I_2 \leq r_2$. If we reorder whenever $I_2 \leq r_2$ we can make several orders while we are waiting for the first one to arrive. (A more general way of dealing with this issue is to include in the state description a logical variable that indicates whether or not the store is waiting for an order to arrive.)

```
# order from depot
if (I2 >= q1) {
  # depot can fill order
  I2 <- I2 - q1
  c <- c + K1
  add stock_arrival_at_store event at time t + L1
  # check depot reorder level
  if (I2 == r2) {
    # order from supplier
    c <- c + K2 + q2*p
    add stock_arrival_at_depot event at time t + L2
  }
} else {
  # depot cannot fill order
  add backlogged_order event at time t + d
}
```

A backlogged order event involves updating the state, then attempting an order from the depot, as above.

```
# update state and event list for a backlogged_order
# update time
t <- new event time
# update holding costs
c <- c + h1*I1*(t - t.old) + h2*I2*(t - t.old)
# order from depot
...
```

The stock arrival events are both straightforward.

```
# update state and event list for a stock_arrival_at_store
# update time
t <- new event time
# update holding costs
c <- c + h1*I1*(t - t.old) + h2*I2*(t - t.old)
# update stock
I1 <- I1 + q1
```

```
# update state and event list for a stock_arrival_at_depot
# update time
t <- new event time
# update holding costs
c <- c + h1*I1*(t - t.old) + h2*I2*(t - t.old)
# update stock
I2 <- I2 + q2
```

Putting all these bits together we get our complete program. The process of breaking down a problem into smaller manageable tasks is sometimes called top-down programming or top-down refinement, and is an important technique for dealing with large problems. In this case we have used what is called an event based viewpoint to structure the problem, but there are other possibilities, such as the *process based* viewpoint, the *activity based* viewpoint, or the *three-phase* approach. For further reading on the topic of discrete event simulation, have a look at the book 'Computer Simulation in Management Science', by Mike Pidd, or 'Simulation Modelling and Analysis', by Law and Kelton.

## 21.4 Seed dispersal

Plant ecologists who perform research in plant propagation are often interested in how far plant seeds disperse from a parent plant. Information about dispersal enables ecologists to make predictions about the ability of an invasive species to colonise a new area, for example.

One of the first questions we can ask is, 'what is the mean displacement of a seed from the parent plant?' In order to frame this question in the context of a model, we can ask, 'what is the distribution of $(R, \Theta)$, the polar coordinates of the displacement from the parent plant of a randomly chosen seed?' To collect suitable data to answer these questions, the ecologists install seedtraps in lines that extend out from the parent plant (see Figure 21.10). These lines are called *transects*. After a specified amount of time (for example, a single flowering season), the seeds in each trap are counted; these seed counts at given distances form the experimental data that we have to work with.

We will assume that the seed rain is radially symmetric around the plant, although this is usually untrue. This assumption is called *isotropy*. An immediate consequence of the assumption is that $\Theta \sim U(0, 2\pi)$, independently of $R$. Moreover, the dispersal of seeds along each transect will be identically distributed, so it is sufficient for us to restrict our attention to a single transect.

Let $T$ be the distance from the parent plant of a seed chosen at random *from the transect*. Importantly, $T$ has a different distribution to $R$, which is what we really want to know. The reason is that the seeds in the closer traps are over weighted relative to the seeds in the remote traps, because their traps subtend a greater angle than do the remote traps. That is, the near traps sample a

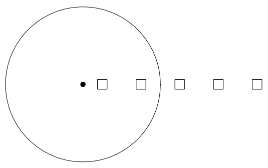

Figure 21.10 *Transect of seedtraps from plant; squares represent seed traps, the circle represents the median of the overall seed shadow, the black dot is the focal parent plant.*

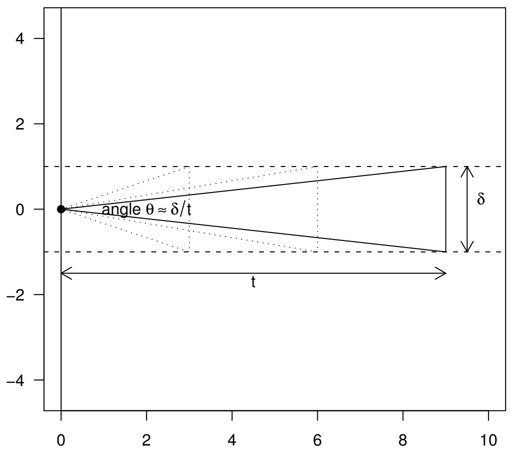

Figure 21.11 *Relating distance along the transect $t$ to the radial distance $r$.*

larger slice of the circular seed rain than do the remote traps. The situation is illustrated in Figure 21.11. Suppose the traps have width $\delta$ (assumed small), then seeds that fall at distance $t$ on the transect will have polar coordinates $(r, \alpha)$, where $r = t$ and $-\theta/2 < \alpha < \theta/2$, for $\theta$ such that

$$t \sin\theta = \delta.$$

If $\delta/t$ is small then so is $\theta$, in which case $\sin\theta \approx \theta$ and we get

$$-\frac{\delta}{2t} < \alpha < \frac{\delta}{2t}.$$

That is, the overcounting of seeds at distance $t$ along the transect is inversely proportional to $t$.

A further problem is that we do not actually observe $T$. Suppose that trap $i$ covers area $[x_i - \epsilon/2, x_i + \epsilon/2] \times [-\delta/2, \delta/2]$, for $j = 1, \ldots, k$. Using the trap centres as our displacements, we observe a discretised version of $T$, call it $T^*$, where

$$\mathbb{P}(T^* = x_i) = \frac{\mathbb{P}(x_i - \epsilon/2 < T < x_i + \epsilon/2)}{\sum_{j=1}^{k} \mathbb{P}(x_j - \epsilon/2 < T < x_j + \epsilon/2)}.$$

In practice, if the traps are regularly spaced and reasonably close together (relative to the range of observations), we will just treat our observations of $T^*$ as if they are observations of $T$. That is, we will ignore this problem.

Let $t_1, \ldots, t_n$ be our sample from $T$. A probability density function can be used to represent the relative number of seeds that are located along the transect, as a function of distance from the parent plant. We will refer to this as the *transect* pdf. For the moment, we shall assume that the transect pdf follows the exponential function; that is, for $0 \le t < \infty$ and $\tau > 0$,

$$f_T(t) = \tau e^{-t\tau},$$

where $\tau$ is the rate parameter. The expected mean and variance of $T$ in terms of the parameters of the model are $1/\tau$ and $1/\tau^2$ respectively, and we can estimate $\tau$ using $\hat{\tau} = 1/\bar{t}$, where $\bar{t}$ is the mean distance from the seeds to the plant.

Fitting the transect pdf is straightforward, but how does this give us the density of $R$, which we call the *radial pdf*? Let $f_R$ be the pdf of $R$, then from Figure 21.11 we see that

$$\begin{aligned} f_T(t)\,dt &= \mathbb{P}(t < T < t + dt) \\ &\approx \mathbb{P}\left(t < R < t + dt \text{ and } -\frac{\delta}{2t} < \Theta < \frac{\delta}{2t}\right) \\ &= f_R(t)\,dt\,\frac{\delta}{2t} \qquad \text{as } R \text{ and } \Theta \text{ are independent.} \end{aligned}$$

That is

$$f_R(r) \propto r f_T(r). \tag{21.3}$$

The approximation step above comes from putting $\sin\Theta \approx \Theta$. The approximation becomes exact in the limit as $\delta \to 0$.

Thus, in the case $T \sim \exp(\tau)$ we have $f_R(r) = kre^{-r\tau}$, for $0 \le r < \infty$ and $\tau > 0$, where $k$ is some normalising constant, chosen so that the density function integrates to 1. Using integration by parts it is easy to check that $k = \tau^2$, so

$$f_R(r) = r\tau^2 e^{-r\tau}.$$

This is the gamma distribution, with shape of 2 and rate of $\tau$. Thus, the mean and variance of the distance that a seed travels, determined radially, are $\mathbb{E}R = 2/\tau$ and $\operatorname{Var} R = 2/\tau^2$, respectively. If we mistakenly use the exponential distribution instead of the gamma distribution, then our model for the seeds will place them too close to the plant and insufficiently variable.

In short, if we measure a transect of seedtraps and fit an exponential distribution to the numbers using $\hat{\tau} = 1/\bar{t}$, then to model the seed rain in two dimensions, we use a $\Gamma(\tau, 2)$ distribution for the radial distance and an independent $U(0, 2\pi)$ distribution for the angle.

Note, by integrating both sides of Equation 21.3, we can deduce that in general

$$f_R(r) = \frac{r f_T(r)}{\mathbb{E}(T)}. \tag{21.4}$$

That is, $\mathbb{E}(T)$ is the appropriate rescaling factor for the length weighted radial distribution.

### *21.4.1 Simulating the radial distance R*

Consider now the problem of simulating the process by which a plant species colonises a new area. If we model this at the level of individual plants, then we need to be able to simulate where the seeds of each plant land. That is, we need to be able to simulate $(R, \Theta)$. Of course, we also need to know how many seeds a plant produces and when, how long the plant lives, and the chance that a seed will successfully germinate, which will depend on where it lands, but these are questions for another time.

As we have seen, if we know the transect density $f_T$ then we can obtain the radial density $f_R$ using Equation 21.4. However the exact functional form of the transect density may not be known. What we would like is a general technique which, assuming we can simulate $T$, allows us to simulate $R$. For example, if we wanted to make no assumptions at all about the distribution of $T$, we could simulate $T$ directly from the observations $t_1, \dots, t_n$. That is, put $\mathbb{P}(T = t_i) = 1/n$ for $i = 1, \dots, n$. A more sophisticated approach would be to use a non-parametric estimate of $f_T$, but this is beyond the scope of this book.

We now demonstrate that it is possible to simulate $f_R$ using $f_T$ and rejection sampling. That is, we can simulate $f_R$ without knowing its closed-form expression, just so long as we know $f_T$. Suppose that the range of $T$ is bounded by $a$. That is $0 \le T \le a$. Take $U \sim U(0, a)$ independently of $T$ then define

$$S = T \,|\, T > U.$$

That is, for $r \in [0, a]$,

$$\mathbb{P}(S \le r) = \mathbb{P}(T \le r \,|\, T > U).$$

To calculate the right-hand side probability we need the following version of the Law of Total Probability, which we give here without proof. For any random variables $X$ and $Y$, with $Y$ continuous, and any set $A \subset \mathbb{R}^2$, we have

$$\mathbb{P}((X, Y) \in A) = \int_y \mathbb{P}((X, y) \in A \,|\, Y = y) f_Y(y) dy.$$

In our case, noting that $T$ and $U$ are independent, we have

$$\begin{aligned}
\mathbb{P}(T \le r \,|\, T > U) &= \frac{\mathbb{P}(U < T \le r)}{\mathbb{P}(T > U)} \\
&= \frac{\int_0^a \mathbb{P}(U < t \le r) f_T(t)dt}{\int_0^a \mathbb{P}(t > U) f_T(t)dt} \\
&= \frac{\int_0^r (t/a) f_T(t)dt}{\int_0^a (t/a) f_T(t)dt} \\
&= \frac{\int_0^r f_R(t)dt}{\int_0^a f_R(t)dt} \\
&= F_R(t) \;=\; \mathbb{P}(R \le t).
\end{aligned}$$

That is, $S = T \,|\, T > U$ has the same distribution as $R$, the radial displacement. Moreover, if we can simulate $T$ then we can simulate $S$ easily using a rejection algorithm. Suppose that `T.sim()` simulates $T$, then to simulate $S$ (or equivalently $R$), we can use the function below. The argument $a$ gives an upper bound on the range of $T$.

```
R.sim <- function(a) {
  while (TRUE) {
    U <- runif(1, 0, a)
    T <- T.sim()
    if (T > U) return(T)
  }
}
```

Recall from Equation 21.3 that compared to $R$, displacements as measured by $T$ are over represented by a factor proportional to the inverse distance from the origin. Our rejection algorithm *thins out* our observations of $T$, by a factor inversely proportional to the distance from the origin, negating the over representation caused by measuring along a transect.

Our definition of $S$ required the range of $T$ to be bounded. In practice, provided we are prepared to live with some occasional errors, if $T$ has an unbounded range then we just take $a$ large enough that $\mathbb{P}(T > a) \le \epsilon$, for some small $\epsilon$. For example, if $T \sim \exp(1/2)$ then $\mathbb{P}(T > a) = \exp(-a/2)$, so for $\epsilon = 0.0001$ we get $a \ge -2\log\epsilon = 18.42$ (to 2 decimal places). What happens is that simulated values of $T$ greater than $a$ are always accepted, rather than being thinned.

To test that the distributions of $S$ and $R$ really are the same, we consider a case where we know what the distribution of $R$ is, and compare that with an empirical estimate of the density of $S$, calculated from a simulated sample. We take the case $T \sim \exp(1/2)$ and $R \sim \Gamma(1/2, 2)$ and put $a = 20$. The output of our simulation experiment is given in Figure 21.12. In the left panel, we show the transect pdf (solid line) and the analytically computed radial pdf (dotted line). In the right panel, we include the analytical radial pdf again, and add an empirical estimate of the density of $S$. The two are extremely close.

To simulate $S$ we use a vectorised version of `R.sim`. This has the advantage of speeding up the simulation, but the disadvantage that we don't know exactly how many observations of $S$ we are going to get. In this case, provided we get enough observations to estimate the density of $S$, this is not a problem.

```
# program spuRs/resources/scripts/seed-test.r

# set up two plots side-by-side
par(las=1, mfrow=c(1,2), mar=c(4,5,0,2))

# graph f_R and f_T on the LHS plot
curve(dgamma(x, shape=2, rate=1/2), from=0, to=20,
      ylim=c(0, dexp(0, rate=1/2)), lty=2,
      xlab="r", ylab=expression(paste(f[T](r), " and ", f[R](r))))
curve(dexp(x, rate=1/2), add=TRUE)
abline(h=0, col="grey")

# generate T, U, and S samples for case 1
T <- rexp(1000000, rate=1/2)
U <- runif(1000000, min=0, max=20)
S <- T[T > U]

# graph estimate of f_S and f_R on the RHS plot
hist(S, breaks=seq(0, max(S)+0.5, 0.5), freq=FALSE,
     xlim=c(0,20), ylim=c(0, dexp(0, rate=1/2)),
     main="", xlab="r",
     ylab=expression(paste(f[R](r), " and ", hat(f)[S](r))),
     col="lightgrey", border="darkgrey")
curve(dgamma(x, shape=2, rate=1/2), add=TRUE)
```

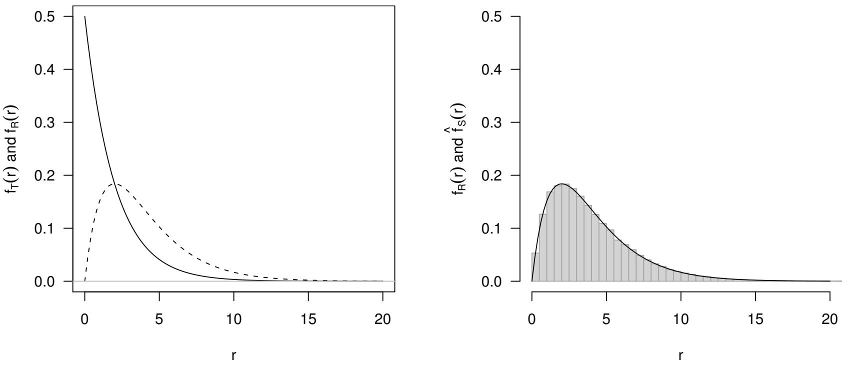

Figure 21.12 *Acceptance/rejection sampling for seed shadows using an exponential transect pdf.*

To conclude this example, we consider a couple of other cases where the radial pdf of $f_R$ can be obtained analytically and can also be easily simulated, thereby providing a further test of our rejection algorithm.

In the first case we suppose that the transect pdf $f_T$ is a lognormal density with parameters $\mu$ and $\sigma^2$. That is,

$$f_T(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}x\sigma} e^{-(\log x - \mu)^2/(2\sigma^2)}.$$

The radial pdf is thus

$$f_R(r) \propto e^{-(\log r - \mu)^2/(2\sigma^2)}.$$

This suggests deriving the distribution of $\log R$ (by applying the transformation theory from Section 14.5.2), which gives $\log R \sim N(\mu + \sigma^2, \sigma^2)$. Hence, $R$ is still lognormal but with parameters $\mu + \sigma^2$ and $\sigma^2$.

In the second case we use a Weibull distribution, with parameters $a$ and $b$, which has density

$$f_T(x \mid a, b) = \frac{a}{b} \left(\frac{x}{b}\right)^{a-1} \exp\left(-\left(\frac{x}{b}\right)^a\right).$$

Thus, for a Weibull transect with parameters $a = 2$ and $b = 2$ say, we have the radial pdf

$$f_R(r) \propto x^2 e^{-x^2/4},$$

which we recognise as a chi distribution with three degrees of freedom, scaled by a factor of $\sqrt{2}$. We write $R \sim \sqrt{2}\chi_3$. A $\chi$ random variable with $k$ degrees of freedom is defined as the square root of a $\chi^2_k$ random variable.

To estimate $f_R$ we use code very similar to the code that created Figure 21.12. The only substantial changes are in simulating $T$, the distribution of the transect pdf (Figure 21.13).

```
> # set up two plots side-by-side
> par(las=1, mfrow=c(1,2), mar=c(4,5,3,2))
> # Construct a graphic for the Lognormal transect pdf
> T <- rlnorm(1000000, meanlog = 0.5, sdlog = 0.55)
> U <- runif(1000000, min=0, max=20)
> S <- T[T > U]
> hist(S, breaks=seq(0, max(S)+0.5, 0.125), freq=FALSE,
+      xlim=c(0,7),  ylim=c(0, dexp(0, rate=1/2)),
+      main="Lognormal", xlab="r",
+      ylab=expression(paste(f[R](r), " and ", hat(f)[S](r))),
+      col="lightgrey", border="darkgrey")
> curve(dlnorm(x, meanlog = 0.5, sdlog = 0.55), add=TRUE, lty=2)
> curve(dlnorm(x, meanlog = 0.8025, sdlog = 0.55), add=TRUE)
> # Construct a graphic for the Weibull transect pdf
> T <- rweibull(1000000, shape=2, scale=2)
> U <- runif(1000000, min=0, max=20)
> S <- T[T > U]
> hist(S, breaks=seq(0, max(S)+0.5, 0.125), freq=FALSE,
+      xlim=c(0,7),  ylim=c(0, dexp(0, rate=1/2)),
+      main="Weibull", xlab="r",
+      ylab=expression(paste(f[R](r), " and ", hat(f)[S](r))),
+      col="lightgrey", border="darkgrey")
> curve(dweibull(x, shape=2, scale=2), add=TRUE, lty=2)
> curve((1/(2*sqrt(pi)))*x^2*exp(-(x^2)/4),add=TRUE)
```

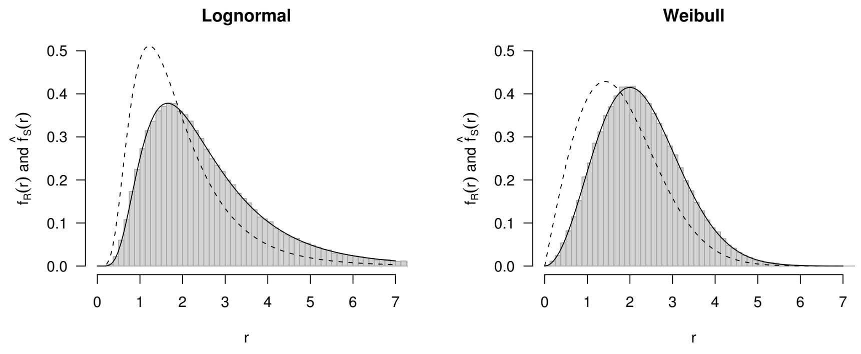

Figure 21.13 *Acceptance/rejection sampling for seed shadows using lognormal and Weibull transect pdfs. The dotted lines represent the transect pdfs, the shaded histograms represent the radial pdfs from simulation, and the solid lines are the exact radial pdfs we derived.*

## *21.4.2 Object-oriented programming implementation*

In Section 8.4 (on object-oriented programming, or OOP), we developed a `trapTransect` class. In this section we will construct a `transectHolder` class that contains one or more `trapTransect` objects, and methods to fit a nominated pdf to the seed distances along the transect, and to simulate random seed locations from the fitted transect pdf using rejection sampling as outlined above.

Here is the S3 `trapTransect` constructor function from Section 8.4, and two methods, `print` and `mean`. Recall that the seed data are stored as seed counts at given distances.

```
> trapTransect <- function(distances, seed.counts, trap.area = 0.0001) {
+   if (length(distances) != length(seed.counts))
+     stop("Lengths of distances and counts differ.")
+   if (length(trap.area) != 1) stop("Ambiguous trap area.")
+   trapTransect <- list(distances = distances,
+                        seed.counts = seed.counts,
+                       trap.area = trap.area)
+   class(trapTransect) <- "trapTransect"
+   return(trapTransect)
+ }
> print.trapTransect <- function(x, ...) {
+   str(x)
+ }
> mean.trapTransect <- function(x, ...) {
+   return(weighted.mean(x$distances, w=x$seed.counts))
+ }
```

We imagine a situation in which we have a large number of plants, around each of which a transect of seedtraps has been installed. Critically, the number of seedtraps may vary by plant. For example, the seedtrap count might change by plant height. In order to store the observations for later analysis, we need a storage device that does not require that everything be the same length, for which a `list` is ideal. That is, a `transectHolder` should contain a list of `trapTransect` objects (and possibly some other bits and pieces).

We wish to have a function `fitDistances`, that takes a `transectHolder` object and fits a nominated pdf to the observed (transect) seed distances. That is, we assume that the transect distribution is the same for each plant, and fit a single pdf to all of our observations. There are various ways we could do this; our approach is to fit the nominated pdf to each transect, then summarise across the transects by taking the mean of the computed parameters. This strategy is not optimal, but is not unreasonable.

We could hide the `fitDistances` function inside the `transectHolder` constructor function, which we present below, but it might be useful to try different models, so we prefer the `fitDistances` function to be readily available to the user. The work of actually fitting a pdf is performed using maximum likelihood by the convenient `fitdistr` function from the `MASS` package: we just need to get our data into the correct format. To this end we define `getDistances`, which takes a `trapTransect` object and returns a vector of seed distances.

```
> fitDistances <- function(x, family=NULL) {
+   # x$transects is a list of trapTransect objects
+   # family is a string giving the name of a pdf
+   require(MASS)  # we need this package for the fitdistr() function
+   getDistances <- function(y) {
+     rep(y$distances, y$seed.counts)
+   }
+   getEstimates <- function(distance) {
+     fitdistr(distance, family)$estimate
+   }
+   distances <- lapply(x$transects, getDistances)
+   parameter.list <- lapply(distances, getEstimates)
+   parameters <- colMeans(do.call(rbind, parameter.list))
+   return(parameters)
+ }
```

Observe the function `do.call`, which accepts as arguments a function name and a list of arguments for the function, and calls the function using the list of arguments.

As before, for the sake of brevity, we omit useful checks for correct object class, and we omit informative behaviour in the case of an empty transect. Note that even though we are operating upon arbitrary numbers of objects, we never need to invoke a loop. Instead we vectorise all the operations using the graceful `lapply` function.

We now give a constructor for our `transectHolder` class. Rather than just create a list of `trapTransect` objects, we will add infrastructure in order to simplify its use for our purposes. The extra infrastructure that we will add is the automatic fitting of a nominated pdf to the transects, and the ability to simulate from the fitted model.

```
> transectHolder <- function(..., family="exponential") {
+   transectHolder <- list()
+   transectHolder$transects <- list(...)
+   distname <- tolower(family)
+   transectHolder$family <- family
+   transectHolder$parameters <-  fitDistances(transectHolder, distname)
+   transectHolder$rng <- switch(distname,
+                                "beta" = "rbeta",
+                                "chi-squared" = "rchisq",
+                                "exponential" = "rexp",
+                                "f" = "rf",
+                                "gamma" = "rgamma",
+                                "log-normal" = "rlnorm",
+                                "lognormal" = "rlnorm",
+                                "negative binomial" = "rnbinom",
+                                "poisson" = "rpois",
+                                "weibull" = "rweibull",
+                                NULL)
+   if (is.null(transectHolder$rng))
+     stop("Unsupported distribution")
+   class(transectHolder) <- "transectHolder"
+   return(transectHolder)
+ }
```

This simple constructor again omits checks for suitable arguments, successful fitting of the probability density function, and so on. The list of `trapTransect` objects is stored as `transectHolder$transects`. `fitDistances` is used to estimate the parameters for the nominated pdf; the pdf family and fitted parameters are stored as `transectHolder$family` and `transectHolder$parameters`. `transectHolder$rng` stores the name of the function that will be used to simulate from the fitted transect distribution. When complete, the object is then assigned the transectHolder class.

A function to print the object might look like this.

```
> print.transectHolder <- function(x, ...){
+   print(paste("This object of class transectHolder contains ",
+               length(x$transects), " transects.", sep=""))
+   str(x)
+ }
```

We can construct a function to simulate $n$ random seed locations, given a transectHolder object, using the generic function simulate.

```
> methods(simulate)

[1] simulate.lm*

   Non-visible functions are asterisked

> simulate

function (object, nsim = 1, seed = NULL, ...)
UseMethod("simulate")
<environment: namespace:stats>
```

We need to write a version that will be specific to our class. We make sure that we match the argument names of the generic function.

```
> simulate.transectHolder <- function(object, nsim=1, seed=NULL, ...) {
+   if (!is.null(seed)) set.seed(seed)
+   distances <- c()
+   while(length(distances) < nsim) {
+     unfiltered <- do.call(object$rng,
+                           as.list(c(10*nsim, object$parameters)))
+     filter <- runif(10*nsim, 0, max(unfiltered))
+     distances <- c(distances, unfiltered[unfiltered > filter])
+     }
+   distances <- distances[1:nsim]
+   angles <- runif(nsim, 0, 2*pi)
+   return(data.frame(distances = distances,
+                     angles = angles,
+                     x = cos(angles) * distances,
+                     y = sin(angles) * distances))
+ }
```

Notice that, using do.call, we directly invoke the random-number generator and pass to it the estimated parameters, without knowing what distribution it is, nor how many parameters it requires.

We now demonstrate the construction of a transectHolder object, using data that mimics the structure of a field experiment, and simulate five random seedlings using the distribution fitted to the trap data.

```
> transect.1 <- trapTransect(distances = 1:4,
+                            seed.counts = c(4, 3, 2, 0))
> transect.2 <- trapTransect(distances = 1:3,
+                            seed.counts = c(3, 2, 1))
> transect.3 <- trapTransect(distances=(1:5)/2,
+                            seed.counts = c(3, 4, 2, 3, 1))
> allTraps <- transectHolder(transect.1, transect.2, transect.3,
+                               family="Weibull")
> allTraps
```

```
[1] "This object of class transectHolder contains 3 transects."
List of 4
 $ transects :List of 3
  ..$ :List of 3
  .. ..$ distances  : int [1:4] 1 2 3 4
  .. ..$ seed.counts: num [1:4] 4 3 2 0
  .. ..$ trap.area  : num 1e-04
  .. ..- attr(*, "class")= chr "trapTransect"
  ..$ :List of 3
  .. ..$ distances  : int [1:3] 1 2 3
  .. ..$ seed.counts: num [1:3] 3 2 1
  .. ..$ trap.area  : num 1e-04
  .. ..- attr(*, "class")= chr "trapTransect"
  ..$ :List of 3
  .. ..$ distances  : num [1:5] 0.5 1 1.5 2 2.5
  .. ..$ seed.counts: num [1:5] 3 4 2 3 1
  .. ..$ trap.area  : num 1e-04
  .. ..- attr(*, "class")= chr "trapTransect"
 $ family    : chr "Weibull"
 $ parameters: Named num [1:2] 2.37 1.8
  ..- attr(*, "names")= chr [1:2] "shape" "scale"
 $ rng       : chr "rweibull"
 - attr(*, "class")= chr "transectHolder"
```

```
> simulate(allTraps, 5, seed = 123)
```

```
  distances   angles          x          y
1 1.9707469 3.769842 -1.5944481 -1.1582653
2 0.7456877 2.091192 -0.3707734  0.6469754
3 2.8898740 3.070046 -2.8824807  0.2065838
4 1.4863482 5.997136  1.4259521 -0.4193945
5 1.6207433 3.034165 -1.6114000  0.1737775
```

This brief demonstration concludes the first phase of development of our class.

We are now able to test our earlier conjecture (that the displacement pdf $f_R$ and the transect pdf $f_T$ are related by $f_R(r) \propto r f_T(r)$), for a wider range of transect distributions, using simulation. We can proceed as follows.

1. Choose one of the available transect distributions, and simulate a two-dimensional seed shadow using the acceptance sampling algorithm.
2. Using only the random points located within a fixed-width transect, compare the quantiles with the original transect distribution.

In short, we should be able to recover our original transect distribution by the correct simulation followed by sampling along a transect.

We simulate from the Weibull distribution, with arbitrary but known shape and scale parameters, and discretise the random numbers to mimic the process of sampling for seedtraps. The pdf of the Weibull density that we used is as follows:

$$f(x) = \frac{a}{b}\left(\frac{x}{b}\right)^{a-1} \exp\left(-\left(\frac{x}{b}\right)^{a}\right)$$

where the shape parameter is $a$ and scale parameter is $b$.

```
> simulated.seed.points <- table(round(rweibull(1000, shape = 2,
+     scale = 5)))[-1]
```

We drop the first measure to remove zeros from the observations. We use these simulated seedtrap points to construct a transect, and store that in a `transectHolder`, in the process fitting the Weibull pdf to the seedtrap data.

```
> simulated.transect <-
+   trapTransect(distances = as.numeric(names(simulated.seed.points)),
+                seed.counts = simulated.seed.points)
> simulated.holder <- transectHolder(simulated.transect, family="Weibull")
```

Finally we simulate a new site, using the fitted model, and select only those points that are in our new transect, which for convenience's sake we will not discretise. We arbitrarily set our transect as being $x > 0$ and $-0.5 < y < 0.5$. If our conjecture is correct then the distribution of these points should be close to the original fitted density, which is Weibull.

```
> good.site <- simulate(simulated.holder, 100000)
> good.points <- good.site$x[abs(good.site$y) < 0.5 & good.site$x > 0]
```

We compare the distributions using the following code, with output in Figure 21.14. Rather than try to visually compare simulated and theoretical distributions, here we provide a scatterplot of the simulated and theoretical quantiles, called a quantile-quantile plot. If the simulated distribution matches the theoretical distribution well, then the simulated quantiles should line up well with the theoretical quantiles. The comparison seems favourable; the simulated distribution along the transect matches the theoretical distribution quite well.

```
> par(las = 1)
> quantiles <- (1:99)/100
> plot(quantile(good.points, probs = quantiles), do.call(qweibull,
+     c(list(quantiles), unlist(simulated.holder$parameters))),
+     xlab = "Simulated Quantiles", ylab = "Theoretical Quantiles")
> abline(0, 1, col = "darkgrey")
```

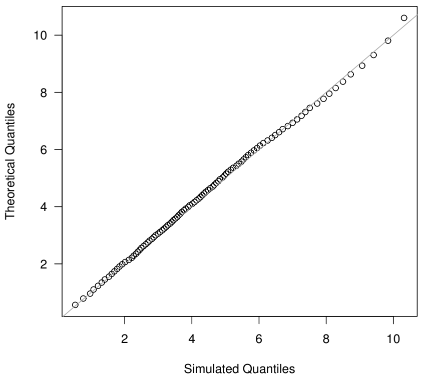

Figure 21.14 *Quantile-quantile plot of observed and theoretical seedling distance distributions.*

To conclude our development, we will add functions to compute the mean and standard deviation of the transect distance, using all the transects contained in the `transectHolder` object. We will assume that each of the plants should have equal weight. Because each object contained within `transectHolder$transects` is a `trapTransect`, we can reuse the `mean.trapTransect` function that we have already written.

```
> mean.transectHolder <- function(x) {
+     mean(sapply(x$transects, mean))
+ }
```

Note that the call to `mean` within the `sapply` function will automatically deploy `mean.trapTransect` if the objects to which the function is being applied are of class `trapTransect`. If at any time we need to use a different function for the mean of `trapTransect` objects, all we have to do is rewrite `mean.trapTransect`.

`sd` is *not* a generic function, so although we can create `sd.transectHolder`, it will not be automatically used in place of `sd` if the latter is called. An explicit call to the function `sd.transectHolder` is needed, as below.

```
> var.trapTransect <- function(x) {
```

```
+     return(var(rep(x$distances, x$seed.counts)))
+ }
> sd.transectHolder <- function(x) {
+     sqrt(mean(sapply(x$transects, var.trapTransect)))
+ }
```

We can now invoke these functions to find the mean of the means of the transect seed distances, and the quadratic mean of the standard deviations of the transect seed distances.

```
> mean(allTraps)

[1] 1.584046

> sd.transectHolder(allTraps)

[1] 0.7746426
```