<1>Chapter 4: Modeling in R

The statistical program *R* provides a straightforward approach to modeling, and has the advantages of being free, fast, available on any operating system, commonly-used, and widely-supported by an online community that helps users who get stuck. In this Chapter, we provide a detailed introducing to using R that will be useful for the modeling methods we introduce and practice in later Chapters.

<2>Setting up *R*

R is supported by a vast online support system called the Comprehensive R Archive Network (CRAN). The easiest way to get started using *R* is to download the *R* program by simply doing a web search for the letter "R", and opening the website for CRAN, which has a "Download" page where *R* can be downloaded and installed for any common operating system (https://cran.r-project.org/).

For our purposes, *R* can be thought of as analogous to the engine of a car: it's a powerful tool that we will use, but unless we're mechanics, we don't want to interfere with its underlying components. Just as we would sit comfortably in the cabin of a car—and use our steering wheel and pedals to control the engine—we also want a comfortable cabin to sit in and issue commands to the engine of *R*. Hence, we want to have an interface that's easier to use than just the standard *R* command line. We prefer *RStudio*, which is free and one of the most popular interfaces that will accept our commands, put them into the engine in *R*, and provide a nice output for us in a digestible format. If we web search "RStudio", a free download is also available for any common operating system (https://www.rstudio.com/products/rstudio/download/). For the purposes of this book, the “Desktop” not the “Server” version of *RStudio* should be installed.

Once we have installed both *R* and *RStudio*, we can open *RStudio* itself, which looks like Figure 4.1.

[INSERT FIGURE 4.1 HERE]

RStudio will send our commands to the engine of *R*, and retrieve the output for us; hence, we do not need to open *R* itself separately (just leave it on our hard drive, after installation). At the top left of Figure 4.1, we can see a white square with a “+” sign, which we should click to start a new “R Script”. This opens a blank page for us to code in. *RStudio* is organized into four sections by default, with a “Code” window that we just opened, to type in commands; a “Console” window that appears in the top right of Figure 4.1 (but maybe located on the bottom left corner of some systems), which tell us what the engine is doing and provides our output to our commands; a “Workspace” (bottom left of Figure 4.1, but occasionally top right on some systems) that shows us the names of our parameters and datasets; and a window with several tabs that allows us to find files, view plots, get help, and install *packages*. Packages in *R* are like apps on a smartphone; if we want extra features to make *R* easier to use or more versatile, we can install a package from the Internet, produced by expert users to save us time and effort.

To get started in *R*, we should learn a few common resources to help us whenever we have questions. We recommend Quick-R, a popular, free webpage that reminds us of useful commands in *R* such as how to import data, change directories, make plots, save our results, and so on (http://www.statmethods.net/).

<2>Useful *R* commands

For the purposes of solving public health or healthcare system problems, we need to become familiar with a few key commands in *R* that are commonly used for modeling. The code that we provide below can also be downloaded from our textbook website. Code files in *R* end with the extension ".R".

First, in the code window (the blank page at the upper left that we created), let's see how *R* works as a calculator. We can type in 1+2 in the code window, and then click the "Run" button at the top right of the window, as shown in Figure 4.2:

[INSERT FIGURE 4.2 HERE]

When we click "Run", we see that *R* provides us with the output in the Console window that looks like this:

```
> 1+2
[1] 3
```

This output shows us three things: (i) our command (which follows the > sign) was to add 1 plus 2; (ii) we had a single output from our command (shown as [1]); and (iii) the output was the number 3. Note that we can put our cursor on the end of a line of code and click "Run" to initiate just that line of code to *R*, or we can highlight a few lines and click "Run" to initiate several lines of code at once. We can store the value as a variable *test* by typing the command: test=1+2. *R* will then store the number "3" as variable *test.* If we type *test* and click "Run", *R* will output a single result, the number 3:

```
> test=1+2
> test
[1] 3
```

For modeling purposes, we often wish to store long lists of numbers, or *vectors*, rather than just a single number. For example, let's use the repeat command to create a vector of ten zeros: rep(0,10)

```
> rep(0,10)
 [1] 0 0 0 0 0 0 0 0 0 0
```

As another example, suppose that we want to count every year in a Markov model from year 1 through year 120. In *R*, creating long vectors is as simple as writing the starting number, a colon, and the ending number. For example, suppose we type 1:100 in the code window, and then click the “Run” button. We would get the output in the Console window that looks like this:

```
> 1:100
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
 [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
 [35]  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
 [52]  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
 [69]  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85
 [86]  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
```

This output shows us that *R* has quickly produced all the numbers from 1 through 100. The square brackets on the left help us keep track of which position in the vector is occupied by a given number, whenever our vector spills over from one line of the console window to the next line; for example, position 18 in the vector is the number 18, and position 35 in the vector is position 35.

R will let us easily save vectors in order to use them later. For example, if we type into our code window the command: years = 1:100, then what we’re doing is creating a vector named *years* which contains all the numbers from 1 to 100. If we click “Run” then the output looks like this:

```
> years=1:100
```

In other words, *R* has not done anything but stored the output under the name *years*. The word “years” should now appear in our “Workspace” window. If we later want to do anything with the vector *years*, we can type in the name *years* rather than having to retype the underlying numbers. For example, suppose we wanted to create another vector called *months*. We could just type into our code window: months = years*12. When we click “Run” we see that years has been stored, and if we just type the word years into the code window and click “Run”, we will get the output:

```
> months=years*12
> months
  [1]   12   24   36   48   60   72   84   96  108  120  132  144
 [13]  156  168  180  192  204  216  228  240  252  264  276  288
 [25]  300  312  324  336  348  360  372  384  396  408  420  432
 [37]  444  456  468  480  492  504  516  528  540  552  564  576
 [49]  588  600  612  624  636  648  660  672  684  696  708  720
 [61]  732  744  756  768  780  792  804  816  828  840  852  864
 [73]  876  888  900  912  924  936  948  960  972  984  996 1008
 [85] 1020 1032 1044 1056 1068 1080 1092 1104 1116 1128 1140 1152
 [97] 1164 1176 1188 1200
```

We see here that *R* has created a vector named *months*, and has multiplied all elements of the vector *years* by 12 to get the values of time in units of months.

We can also create vectors by using the combine command, which means we can type numbers into a vector manually; for example, suppose we want a vector called *testvector* that contains the numbers 1, 3, 6, and 8. We can just type: testvector = c(1,3,6,8), where “c” with parentheses tells us to combine (link together) the four numbers in a vector.

```
> testvector=c(1,3,6,8)
> testvector
[1] 1 3 6 8
```

We can pull up any element of a vector by calling the vector name and the element number in square brackets. For example, if we want the third element of *testvector*, we can type: testvector[3] and get:

```
> testvector[3]
[1] 6
```

R makes it convenient to identify the individual values within a vector. For example, suppose we want to identify what the value of the vector *years* corresponds to month 60. We can simply type the command: years[months==60], which means “tell us which value of the vector years corresponds to the point at which the vector months is equal to a value of 60”. Note that the command includes a double equal sign “==” not a single equal sign; a single equal sign would tell *R* to set (or replace) the value of a variable or vector to a particular value, whereas a double equal sign means “go look for when this variable or vector equals this value”. We get the output:

```
> years[months==60]
[1] 5
```

Similarly, suppose we want to know all values of the vector *years* for which the variable months is greater than 100. We would type the command: years[months>100] and get:

```
> years[months>100]
 [1]   9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24
[17]  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40
[33]  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56
[49]  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
[65]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88
[81]  89  90  91  92  93  94  95  96  97  98  99 100
```

Suppose we just want to pull out all value of the vector *years* that are values between 5 and 10. We can type the command: years[(years>5)&(years<10)]. This tells us to look into the vector *years* and look for values greater than 5 that are also (&) less than a value of 10. Alternatively, if we want "or" then we can include the vertical line "|" in place of the ampersand "&", to find all years greater than 5 or less than 10. If we use the "and" command, we get:

```
> years[(years>5)&(years<10)]
[1] 6 7 8 9
```

Just as we can create vectors in *R*, we can also create matrices, such as transition matrices for Markov models. To create matrix, we can create a vector, and then tell *R* to organize the vector into a series of columns and rows. For example, suppose we want a matrix to look like this:

[Equation 4.1] $$\begin{bmatrix} 0.90 & 0.05 & 0.00 & 0.00 \\ 0.10 & 0.85 & 0.05 & 0.00 \\ 0.00 & 0.10 & 0.85 & 0.05 \\ 0.00 & 0.00 & 0.10 & 0.95 \end{bmatrix}$$

We can store this matrix in *R* by typing the full vector of numbers, left to right, and top to bottom, and use the matrix command in *R* to turn that vector into a matrix named *testmatrix*: `testmatrix = matrix(c(0.90, 0.05, 0.00, 0.00, 0.10, 0.85, 0.05, 0.00, 0.00, 0.10, 0.85, 0.05, 0.00, 0.00, 0.10, 0.95),ncol=4,byrow=TRUE).` In this command, "matrix" tells *R* to turn the vector into a matrix, "c" tells *R* what numbers to combine into the vector, "ncol" tells *R* how many columns the matrix should be, and "byrow=TRUE" tells *R* that we want the vector to be read into the matrix row by row (so that 0.90 is followed by 0.05 in row 1, not by 0.10):

```
> testmatrix = matrix(c(0.90, 0.05, 0.00, 0.00, 0.10, 0.85, 0.05, 0.00,
0.00, 0.10, 0.85, 0.05, 0.00, 0.00, 0.10, 0.95),ncol=4,byrow=TRUE)
> testmatrix
```

```
     [,1] [,2] [,3] [,4]
[1,]  0.9 0.05 0.00 0.00
[2,]  0.1 0.85 0.05 0.00
[3,]  0.0 0.10 0.85 0.05
[4,]  0.0 0.00 0.10 0.95
```

<2>A Markov model in *R*

Let's create our Markov model of the male pattern baldness "epidemic" from Chapter 3 in *R*. Doing so will help us to understand what happens in non-steady-state conditions—early in an epidemic, or soon after an intervention is introduced, but before the long-term steady state equilibrium has been reached. Programming the model in *R* will also facilitate performing cost-effectiveness analysis for an intervention. To recall, we modeled how susceptible men could transition among multiple states of disease; as shown in Figure 4.3, we'll start our model in the "pre-intervention" case where we have not yet introduced our spray-on hair solution.

[INSERT FIGURE 4.3 HERE]

To create this Markov model, we need to program in the three essential components of the model: the transition matrix, a set of initial conditions, and a series of vectors. The transition matrix is the same at the one above: it is the matrix showing us the probability of moving among the four states in our model, hence we can simply adopt the command from above:

```
> transition = matrix(c(0.90, 0.05, 0.00, 0.00, 0.10, 0.85,
0.05, 0.00, 0.00, 0.10, 0.85, 0.05, 0.00, 0.00, 0.10,
0.95),ncol=4,byrow=TRUE)
```

The initial conditions refer to the probability of being in each of the four states at the start of the simulation. For our purposes, let's suppose the epidemic of male pattern baldness is just beginning. We can be set a probability of 1 of being hairy at the start, and probabilities of 0 of having frontal bossing, a combover, or a bowling ball at the start. We can create four vectors that will be empty at the start of the model, and that we will fill in for every year of our simulation for a total of 100 years of simulated time. We can do that by typing the commands:

```
timesteps = 100

h = rep(0,timesteps)

f = rep(0,timesteps)

c = rep(0,timesteps)

b = rep(0,timesteps)
```

Among these commands, *timesteps* tells us how many periods of time (in this case, years) will be simulated in the model. The subsequent commands create vectors of 100 zeros to fill in later. We want to fill in the first value of the *h* vector with the value "1" to correspond to the initial conditions. We can do that by typing:

```
h[1]=1
```

Note that by using a single equals sign, not a double equals sign, we have assigned a value of 1 to the *h* vector at time step 1. We can check by typing h which now outputs:

```
> h
  [1] 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 [33] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 [65] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 [97] 0 0 0 0
```

We can see here that the first vector value has been changed to "1" and the remaining components remain as values of zero.

Now we need to update the model to calculate the probabilities of being hairy, or having frontal bossing, a combover, or a bowling ball for subsequent years. To do so, let's note that the

Markov model's equations can be expressed in the following form with matrix multiplication:

[Equation 4.2]
$$\begin{bmatrix} 0.90 & 0.05 & 0.00 & 0.00 \\ 0.10 & 0.85 & 0.05 & 0.00 \\ 0.00 & 0.10 & 0.85 & 0.05 \\ 0.00 & 0.00 & 0.10 & 0.95 \end{bmatrix} \begin{bmatrix} h(t-1) \\ f(t-1) \\ c(t-1) \\ b(t-1) \end{bmatrix} = \begin{bmatrix} h(t) \\ f(t) \\ c(t) \\ b(t) \end{bmatrix}$$

We can first create a vector for what the values of *h, f, c* and *b* were at a given time *t*-1, and then multiply that vector by the transition matrix to get the values at time *t*. For example, let's create a vector called *priorstate* which will consistent of the values of *h, f, c* and *b* in year 1. We can write the code: priorstate = c(h[1],f[1], c[1], b[1]), which would mean that the *priorstate* vector is the first value of vectors *h, f, c* and *b*:

```
> priorstate =c(h[1],f[1],c[1],b[1])

> priorstate

[1] 1 0 0 0
```

What will the values of *h, f, c* and *b* be at the next time point, year 2? Based on Equation 4.2, it would be the value of our transition matrix multiplied by the value of our current state vector. In *R*, while regular multiplication is conducted by writing the asterisk (*) symbol, matrix multiplication is conducted just by wrapping the asterisk between percent signs (%*%). We can see the multiplying the transition matrix by the current state matrix (the probability of being in each state in year 1) provides us with the probability of being in each state during year 2:

```
> transition%*%priorstate

     [,1]

[1,]  0.9

[2,]  0.1

[3,]  0.0

[4,]  0.0
```

Now we want to update that *currentstate* vector to change values for all subsequent years of our simulation, which is 100 years long. We can do that using a *for loop*, which says that for values of time from time point 1 through time point 100, we should update the *currentstate* vector to provide the updated values of *h, f, c* and *b*. We can do this by writing `for (t in 2:timesteps)` in *R*.

The `for` command tells *R* to repeat doing something from year 2 until the value of *timesteps*, which is year 100. What should *R* do? In curved brackets "{}" we can insert a series of commands that *R* should repeat. Let's tell *R* to do the following set of commands:

```
for (t in 2:timesteps)
{
  priorstate =c(h[t-1],f[t-1],c[t-1],b[t-1])
  newstate= transition%*%priorstate
  h[t]=newstate[1]
  f[t]=newstate[2]
  c[t]=newstate[3]
  b[t]=newstate[4]
}
```

Here, we see that within the for loop, we're asking *R* to do three things. First, we're asking it to declare what the prior state is. We're create a vector called *priorstate* in which we take the values from the *h, f, c* and *b* vectors from the prior time period. For example, if *t* is year 2, the *priorstate* vector is constructed using year 1's values. Then we multiply the transition matrix by this *priorstate* vector to get the updated probabilities in the next year. Finally, we tell *R* to assign these new values to the corresponding position in vectors *h, f, c* and *b*. For example, the value in *newstate* position 3 if *t* is 2 will be the value of *c* in year 2.

If we highlight this whole set of code and click "Run", we will produce an output that is equivalent to filling out an entire set of columns in Excel: the vectors *h, f, c* and *b* will all be filled in for all values of time through year 100. We can check this by typing in the names of the vectors and seeing their updated states at year 100:

```
> h[100]
[1] 0.06978277
> f[100]
[1] 0.1364494
> c[100]
[1] 0.2666666
> b[100]
[1] 0.5271013
```

As we can see, *h, f, c* and *b* have achieved probabilities that are getting close to the long-term steady state values we estimated by hand in the previous Chapter. But they aren't perfectly equal to the long-term values, because that long term may take quite a few centuries. *R* is, in general, computationally efficient, so we can easily simulate 1,000 years instead of just 100 years of time. Suppose we just change the value of the timesteps variable to 1,000. Then if we re-run the model, it typically only takes a matter of a few additional seconds to get this long-term steady state result:

```
> h[1000]
[1] 0.06666667
> f[1000]
[1] 0.1333333
> c[1000]
[1] 0.2666667
> b[1000]
```

```
[1] 0.5333333
```

If we want to plot the probability over time of having a combover, we can just type: plot(c), and obtain Figure 4.4. The full *R* code for programming our model in *R* and producing Figure 4.4 is provided on the book website.

[INSERT FIGURE 4.4 HERE]

The figure illustrates that it takes at least 40 years or so before the probability of having a combover comes near its steady-state prevalence in this population. Especially for epidemics, it may be quite useful to program a model in *R* and identify what the early changes are in the epidemic, well before a long-term steady state occurs. Additionally, it may be useful to know those early-year changes that take place once an intervention is introduced. We will incorporate the more-complex short-term dynamics of the epidemic into a cost-effectiveness analysis of our intervention (spray-on hair for combovers) in the next section.

<2>A cost-effectiveness analysis in *R*

We can recall from the previous chapter that we wanted to understand the impact of a new treatment for combovers—a spray-on solution that helps people transition from the combover state to the hairy state at a rate of 5% or 0.05 per year. The new treatment changed our transiton matrix to the following equation:

[Equation 4.3]

$$\begin{bmatrix} 0.90 & 0.05 & \mathbf{0.05} & 0.00 \\ 0.10 & 0.85 & 0.05 & 0.00 \\ 0.00 & 0.10 & \mathbf{0.80} & 0.05 \\ 0.00 & 0.00 & 0.10 & 0.95 \end{bmatrix} \begin{bmatrix} h(t-1) \\ f(t-1) \\ c(t-1) \\ b(t-1) \end{bmatrix} = \begin{bmatrix} h(t) \\ f(t) \\ c(t) \\ b(t) \end{bmatrix}$$

We can create a copy of the code from which to calculate the new outcomes given the new transition matrix. We can just cut and paste our *R* code and label the new transition matrix and new probabilities with "_new" to distinguish them from our old "baseline" simulation:

```
transition_new = matrix(c(0.90, 0.05, 0.05, 0.00, 0.10, 0.85, 0.05,
0.00, 0.00, 0.10, 0.80, 0.05, 0.00, 0.00, 0.10,
0.95),ncol=4,byrow=TRUE)
timesteps = 100
h_new = rep(0,timesteps)
f_new = rep(0,timesteps)
c_new = rep(0,timesteps)
b_new = rep(0,timesteps)
h_new[1]=1
priorstate_new =c(h[1],f[1],c[1],b[1])
transition_new%*%priorstate_new
for (t in 2:timesteps)
{
  priorstate_new =c(h[t-1],f[t-1],c[t-1],b[t-1])
  newstate_new= transition_new%*%priorstate_new
  h_new[t]=newstate_new[1]
  f_new[t]=newstate_new[2]
  c_new[t]=newstate_new[3]
  b_new[t]=newstate_new[4]
}
h_new[100]
f_new[100]
c_new[100]
b_new[100]
```

These results reveal that the prevalence of hairy people has increased, while the prevalence of frontal bossing is unchanged, and the prevalence of combovers and bowling balls is decreased:

```
> h_new[100]
[1] 0.0831161
> f_new[100]
[1] 0.1364494
> c_new[100]
[1] 0.2533333
> b_new[100]
[1] 0.5271013
```

Suppose we have the parameters in Table 4.1 below for calculating the cost-effectiveness of the intervention. As shown in the Table, there are costs to the usual hair regrowth treatment, which everyone with frontal bossing, combovers, and bowling ball head receives each year. This occurs before and after the new spray-on treatment is introduced. In addition, once the spray-on treatment is introduced, there is an additional cost of the new spray-on treatment, which applies only to people with combovers. Our objective is to compare the costs of male pattern baldness at the population level both before the new spray-on treatment is available (when only the usual hair regrowth treatment is available), and after the new spray-on treatment is available (when presumably less of the usual hair regrowth treatment will be used given fewer combovers and bowling balls, but the new spray-on treatment will produce new costs). We'll have to consider the changes in the probability of incurring these costs, given how the new treatment affects the probabilities of being in each state over time. We will also need to consider whether the reduction in combovers from the new spray-on treatment is "worth it" from a cost-effectiveness perspective, given the quality-adjusted life-years (QALYS) lost due to frontal bossing, combovers, and bowling ball head before the new spray-on treatment is introduced (when only the usual hair regrowth treatment is available), and after the new spray-on treatment is introduced (when fewer QALYs will be lost due to combovers and bowling balls).

[INSERT TABLE 4.1 HERE]

To conduct this cost-effectiveness analysis, we can first run both of the `for` loops in *R*, which will generate the two sets of vectors (one set being *h, f, c* and *b* for the baseline probabilities of being in a given state over time, before the spray-on treatment; the other set being *h_new, f_new, c_new* and *b_new* for the probabilities of being in a given state over time, after the spray-on treatment has been introduced).

In a population of 100,000 people, costs for the pre-intervention model can be calculated as:

```
costs = sum(100000*(f*10+c*100+b*500))
```

This command first calculates $10 times the probability of having frontal bossing in a given year, plus $100 times the probability of having a combover in a given year, plus $500 times the probability of having a bowling ball in a given year, all multiplied by the population size to get the total costs for our population, then summed across all years of the simulation. (If we wanted to include time-discounting as discussed in an earlier chapter, we could divide the expression within the `sum()` command by `1.03^(1:timesteps)`. The command raises 1.03 to the power of the vector `1:timesteps` (the years, multiplying each year by its corresponding probability of each state from *f* and *b*), where 1.03 reflects a 3% annual discount rate).

Similarly, costs for the post-intervention model, adding in the new higher-cost spray-on therapy to the cost of being in a combover state, would be:

```
costs_new = sum(100000*(f_new*10+c_new*(100+1000)+b_new*500))
```

Analogously, we could sum QALYs under the pre-intervention and post-intervention conditions:

```
qalys = sum(100000*(h*1+f*(1-0.01)+c*(1-0.05)+b*(1-0.10)))

qalys_new = sum(100000*(h_new*1+f_new *(1-0.01)+c_new *(1-0.05)+b_new
*(1-0.10)))
```

Here, we've summed the quality-adjusted life-year values to obtain QALYs. Note that the Table 4.1 values provide an estimate of the QALYs *lost* for being in a given state for a year, hence we must subtract them from 1 to get the QALYs accumulated in each year of the simulation. (Again, we can also include discounting, by dividing each expression within the sum() command by `1.03^(1:timesteps)`).

Finally, we can compute the incremental cost-effectiveness ratio (ICER) as: `(costs_new-costs)/(qalys_new-qalys)`, which gives:

```
> (costs_new-costs)/(qalys_new-qalys)
[1] 382528.3
```

Therefore, we find that the new spray-on therapy for combover costs approximately $382,528 per QALY gained (a high cost for hair, in my opinion!). The full *R* code for programming our model in *R* and producing the cost-effectiveness analysis is provided on the book website.

<2>Uncertainty analysis in *R*

A key advantage of programming models in *R* is the ability to perform *uncertainty analysis*, or the incorporation of uncertain parameters into our models. All models are loose representations of reality, helping us to conceptualize what our futures might be under different circumstances. In no way should we confuse such modeling for precising predicting the future. Rather, models such as this one help us conceptualize what our assumptions are, what knowledge we have about a disease, and how we believe we are going to intervene. We can get a sense of how much uncertainty we might have around our predictions, subject to the assumptions inherent in our model.

In our spray-on hair treatment example, we assumed the values in Table 4.1 were the same for everyone in the population. But suppose we have conducted a study and found that the input parameters to our model are not the same for everyone, but can vary substantially. Let's take just one parameter for illustrative purposes: the cost of the new spray-on hair treatment. Suppose the cost is $1000 per person with combover per year on average, but that the cost can be variable and uncertain, because more of the spray is needed for people with big heads, while less of the spray is needed for people with small heads. In reality, the mean cost is $1000 per person with combover per year, but has a standard deviation of $50. Hence, if the cost is normally-distributed (Gaussian), the 95% confidence intervals around the $1000 estimate are $1000 – (1.96 x 50) and $1000 + (1.96 x 50), or from $902 to $1,098 (note that 1.96 times a standard deviation is the length of the 95% confidence interval). This is an incredibly large amount of variation from the perspective of applying the program to 100,000 people among whom a large portion have a combover. Hence, we really should know—especially for planning our budget—how much variation we might have in the cost of the program because of the uncertain head-size-related costs.

How can we incorporate this uncertainty into the Markov model? In *R*, we can quickly incorporate the uncertainty in this key parameter, and identify how much it might affect the results of our cost-effectiveness analysis.

To incorporate uncertainty, we can first recognize that *R* can quickly generate random numbers. To generate normally-distributed random numbers, we use the `rnorm` command: `rnorm(n = 10, mean = 1000, sd = 50)`, and *R* will produce 10 random numbers from a normal distribution with mean 1000 and standard deviation 50:

```
> rnorm(n = 10, mean = 1000, sd = 50)
 [1] 1047.1704 1001.1454 1027.3702
 [4] 1056.2331 1019.5273  918.1870
 [7] 1009.4602 1072.7324  947.5591
[10]  994.2515
```

We can use the *rnorm* command to analyze the uncertainty around our cost-effectiveness analysis estimate of the ICER. For example, let's see how much our estimate of the ICER varies if we take into account the uncertainty in spray-on treatment costs.

To conduct such an uncertainty analysis, we should first create a random vector of zeros to fill in, for the values of our costs for the spray-on treatment. Just as we created vectors of zeros to fill in for the *h, f, c* and *b* vectors, we can similarly create a vector to fill in for *costs_new* vector. Let's say that we want to repeatedly calculate ICER values 10,000 times, while sampling repeatedly from the distribution of possible values for the spray-on therapy's cost. In other words, we want to find out how much our ICER estimates might vary after we've sampled extensively from the possible values of the spray-on treatment's cost. Hence, we can create a vector of zeros for *costs_new* that is 10,000 numerals long:

```
costs_new = rep(0,10000)
```

Next, we can create a vector with 10,000 values of the spray-on treamtent's program cost per person with a combover per year; let's call it *costvec*:

```
costvec = rnorm(n = 10000, mean = 1000, sd = 50)
```

Finally, we can make use of the same `for` loop strategy we adopted in programming our Markov model to produce estimates of *costs_new*:

```
for (val in 1:10000)
{
  costs_new[val] =
  sum(100000*(f_new*10+c_new*(100+costvec[val])+b_new*500))
}
```

Here, we are repeating a loop 10,000 times. Within each loop, we are iterating the value of parameter *val* from 1 to 10,000. Our equation for *costs_new* is identical to its value previously, except that we have replaced the fixed cost number "1000" with the expression *costvec[val]*, which means that *R* will sample value number *val* from vector *costvec*, which is the vector of random normal numbers with mean $1000 and standard deviation $50. Hence, we are filling in the value of *costs_new* in position *val* to calculate the cost at each value of the vector *costvec*.

Once the "for loop" is done, we can re-calculate the ICER just as we did before:

```
icer = (costs_new-costs)/(qalys_new-qalys)
```

This time, however, icer has become a vector of 10,000 values, because *costs_new* has 10,000 values. *R* smartly kept the other variables the same value when performing the calculation.

How variable is our ICER estimate? We can plot a histogram describing the distribution of the ICER variable, by typing the command: `hist(icer)`, which produces Figure 7.5.

[INSERT FIGURE 4.5 HERE]

As we see in Figure 4.5, there is really wide variability in our ICER estimate due to the seemingly-small variability in the $1000 per person per year estimate of spray-on therapy's cost per person with a combover. We can quantify just how wide the variability is by calculating 95% confidence intervals around our estimate, by typing: `quantile(icer, c(0.025, 0.975)).` This expression calculates any quantiles of choice, and the vector c(0.025, 0.975) means that we want to calculate the 2.5th percentile and the 97.5th percentile, which corresponds to the 95% confidence intervals:

```
> quantile(icer, c(0.025,0.975))
    2.5%    97.5%
345176.4 419982.7
```

As shown here, our ICER estimate has a very wide 95% confidence interval, from $345,176 to $419,983.

We can add further uncertainty analyses by following the above protocol, applied to multiple variables sample from multiple distributions. *R* has many distributions other than the normal (Gaussian) distribution, and can even fit distributions to empirical data, using packages such as the `fitdistrplus` package. Uncertainty analysis in *R* allows us to get a better sense of what level of confidence we might have in our model estimates, and therefore what level of uncertainty we have during budgeting and implementation of a health program in practice.

In the next few chapters, we'll learn more about the power of using *R* to create models that move beyond the limitations of Markov modeling, including models that can account for the spread of infection, or models that can account for complex histories of illness in more versatile ways than we have seen here. For now, the examples in this Chapter should provide a foundation for experimenting with simple models that help in healthcare and public health decision-making, and for using *R* as a software that can help generate simulations from models to evaluate the effectiveness and cost-effectiveness of an intervention at the population level.