# <1>Chapter 4: Modeling in R

In previous Chapters, we implemented our models in spreadsheets for several reasons: (i) spreadsheet are straightforward to work with, and do not require extensive knowledge of programming; (ii) spreadsheets are easy to communicate with among people with varying levels of expertise; (iii) spreadsheets are convenient for optimization problems, given the availability of pre-packaged Solver algorithms; and (iv) spreadsheets are commonly used for modeling in the world of business management, non-profit organization budgeting, consulting, and even academia. But spreadsheets have their limitations, particularly when we want to model large populations, long periods of time, or complex situations in which we have many states or conditions and lots of equations. For larger-scale models where we desire more flexibility than can be offered by a spreadsheet, the free 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 more advanced 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 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 take our commands and run it through 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. If we web search "RStudio", a free download is available for any 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). 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 use. *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, we can install a package from the Internet, produced by expert users to save us time for common tasks.

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/).

so on (http://www.statmethods.net/).

## <2>Useful *R* commands

For the purposes of solving public health or health system problems, we need to become familiar with a few key commands in *R* that are useful 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 one result (shown as [1]); and (iii) the result is the number 3. Note that we can put our cursor on the end of a line of code and click "Run" to transfer just that line of code to *R*, or we can highlight a few lines and click "Run" to run 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 10 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 month in a Markov model from month 1 through month 120. In Excel, we would need to make a cell in our spreadsheet that had the number "1" (let's say in cell A1), then create an equation such as "=1+A1" in cell A2, and drag that formula down 120 cells to create the numbers 1 through 120. In *R*, creating vectors is as simple as writing the starting number, a colon, and the ending number. For example, suppose we type 1:120 in the code window, and then click the "Run" button. We would get the output in the Console window that looks like this:

```
> 1:120
  [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  99 100 101 102
[103] 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119
[120] 120
```

This output shows us that *R* has quickly produced all the numbers from 1 through 120. 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: months = 1:120, then what we're doing is creating a vector named *months* which contains all the numbers from 1 to 120. If we click "Run" then the output looks like this:

```
> months=1:120
```

In other words, *R* has not done anything but stored the output under the name *months*. The word "months" should now appear in our "Workspace" window. If we later want to do anything with the vector months, we can type in the name *months* rather than having to retype the underlying numbers. For example, suppose we wanted to create another vector called *years* the underlying numbers. For example, suppose we wanted to create another vector called *years*. We could just type into our code window: years = months/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:

```
> years=months/12
> years
 [1]  0.08333333  0.16666667  0.25000000  0.33333333  0.41666667
 [6]  0.50000000  0.58333333  0.66666667  0.75000000  0.83333333
[11]  0.91666667  1.00000000  1.08333333  1.16666667  1.25000000
[16]  1.33333333  1.41666667  1.50000000  1.58333333  1.66666667
[21]  1.75000000  1.83333333  1.91666667  2.00000000  2.08333333
[26]  2.16666667  2.25000000  2.33333333  2.41666667  2.50000000
[31]  2.58333333  2.66666667  2.75000000  2.83333333  2.91666667
[36]  3.00000000  3.08333333  3.16666667  3.25000000  3.33333333
[41]  3.41666667  3.50000000  3.58333333  3.66666667  3.75000000
[46]  3.83333333  3.91666667  4.00000000  4.08333333  4.16666667
[51]  4.25000000  4.33333333  4.41666667  4.50000000  4.58333333
[56]  4.66666667  4.75000000  4.83333333  4.91666667  5.00000000
[61]  5.08333333  5.16666667  5.25000000  5.33333333  5.41666667
[66]  5.50000000  5.58333333  5.66666667  5.75000000  5.83333333
[71]  5.91666667  6.00000000  6.08333333  6.16666667  6.25000000
[76]  6.33333333  6.41666667  6.50000000  6.58333333  6.66666667
[81]  6.75000000  6.83333333  6.91666667  7.00000000  7.08333333
[86]  7.16666667  7.25000000  7.33333333  7.41666667  7.50000000
[91]  7.58333333  7.66666667  7.75000000  7.83333333  7.91666667
[96]  8.00000000  8.08333333  8.16666667  8.25000000  8.33333333
[101]  8.41666667  8.50000000  8.58333333  8.66666667  8.75000000
[106]  8.83333333  8.91666667  9.00000000  9.08333333  9.16666667
[111]  9.25000000  9.33333333  9.41666667  9.50000000  9.58333333
[116]  9.66666667  9.75000000  9.83333333  9.91666667 10.00000000
```

We see here that *R* has created a vector named *years*, and has divided all elements of the vector *months* by 12 to get the values of time in units of years. You can imagine this is equivalent to listing all the months in one column of Excel, and then creating a second column that divided the first column by 12.

We can also create vectors by using the concatenate 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 concatenate (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* is in month 70. We can simply type the command: years[months==70], 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 70". Note that the command includes a double equal sign "==" not a single equal sign; a single equal sign would tell *R* to set 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==70]
[1] 5.833333
```

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]  8.416667  8.500000  8.583333  8.666667  8.750000  8.833333  8.916667
[8]  9.000000  9.083333  9.166667  9.250000  9.333333  9.416667  9.500000
[15]  9.583333  9.666667  9.750000  9.833333  9.916667 10.000000
```

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] 5.083333 5.166667 5.250000 5.333333 5.416667 5.500000 5.583333
[8] 5.666667 5.750000 5.833333 5.916667 6.000000 6.083333 6.166667
[15] 6.250000 6.333333 6.416667 6.500000 6.583333 6.666667 6.750000
[22] 6.833333 6.916667 7.000000 7.083333 7.166667 7.250000 7.333333
[29] 7.416667 7.500000 7.583333 7.666667 7.750000 7.833333 7.916667
[36] 8.000000 8.083333 8.166667 8.250000 8.333333 8.416667 8.500000
[43] 8.583333 8.666667 8.750000 8.833333 8.916667 9.000000 9.083333
[50] 9.166667 9.250000 9.333333 9.416667 9.500000 9.583333 9.666667
[57] 9.750000 9.833333 9.916667
```

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:

[Basu model of male pattern baldness– 4 states]

## <2>A Markov model in *R*

[Basu model of male pattern baldness– 4 states]

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

[Basu model of male pattern baldness– 4 states]

## <2>Unique flexibilities in *R*: Uncertainty analysis

So far, we have shown how to reproduce our Markov model from Excel in *R*, but not clearly indicated anything—except for computational speed—that might convince us that programming in *R* is uniquely advantageous and provides opportunities we didn't have in Excel.

The first key advantage is to perform *uncertainty analysis*, or the incorporation of uncertain parameters into our models.

[Basu model of male pattern baldness– 4 states]

We can add further uncertainty analyses by following the above protocol, and can even account for the uncertainty in multiple variables at once using this procedure to get a better sense of what level of confidence we might have in our model estimates.

In the next few chapters, we'll learn more about the power of using *R* to create models that are difficult or impossible to create using spreadsheets, and see how we can further incorporate uncertainty analysis and related forms of probabilistic thinking into our analyses.