Chapter 11
Additional Programming Tools

As the last several chapters have shown, R offers users flexibility and opportunities for advanced data analysis that cannot be found in many programs. In this final chapter, we will explore R's programming tools, which allow the user to create code that addresses any unique problem he or she faces.

Bear in mind that many of the tools relevant to programming have already been introduced earlier in the book. In Chap. 10, the tools for matrix algebra in R were introduced, and many programs require matrix processing. Additionally, logical (or Boolean) statements are essential to programming. R's logical operators were introduced in Chap. 2, so see Table 2.1 for a reminder of what each operator's function is.

The subsequent sections will introduce several other tools that are important for programming: probability distributions, new function definition, loops, branching, and optimization (which is particularly useful for maximum likelihood estimation). The chapter will end with two large applied examples. The first, drawing from Monogan (2013b), introduces *object-oriented programming* in R and applies several programming tools from this chapter to finding solutions from an insoluble game theoretic problem. The second, drawing from Signorino (1999), offers an example of *Monte Carlo analysis* and more advanced maximum likelihood estimation in R.[1] Together, the two applications should showcase how all of the programming tools can come together to solve a complex problem.

Electronic supplementary material: The online version of this chapter (doi: 10.1007/978-3-319-23446-5_11) contains supplementary material, which is available to authorized users.

[1]See also: Signorino (2002) and Signorino and Yilmaz (2003).

J.E. Monogan III, *Political Analysis Using R*, Use R!,
DOI 10.1007/978-3-319-23446-5_11

Table 11.1 Using probability distributions in R

Prefix	Usage	Suffix	Distribution
p	Cumulative distribution function	norm	Normal
d	Probability density function	logis	Logistic
q	Quantile function	t	*t*
r	Random draw from distribution	f	*F*
		unif	Uniform
		pois	Poisson
		exp	Exponential
		chisq	Chi-squared
		binom	Binomial

11.1 Probability Distributions

R allows you to use a wide variety of distributions for four purposes. For each distribution, R allows you to call the cumulative distribution function (CDF), probability density function (PDF), quantile function, and random draws from the distribution. All probability distribution commands consist of a prefix and a suffix. Table 11.1 presents the four prefixes, and their usage, as well as the suffixes for some commonly used probability distributions. Each distribution's functions take arguments unique to that probability distribution's parameters. To see how these are specified, use help files (e.g., `?punif`, `?pexp`, or `?pnorm`).[2]

If you wanted to know the probability that a standard normal observation will be less than 1.645, use the *cumulative distribution function* (CDF) command `pnorm`:

```
pnorm(1.645)
```

Suppose you want to draw a scalar from the standard normal distribution: to draw $a \sim \mathcal{N}(0, 1)$, use the *random draw* command `rnorm`:

```
a <- rnorm(1)
```

To draw a vector with ten values from a χ^2 distribution with four degrees of freedom, use the random draw command:

```
c <- rchisq(10,df=4)
```

Recall from Chap. 10 that the `sample` command also allows us to simulate values, whenever we provide a vector of possible values. Hence, R offers a wide array of data simulation commands.

[2]Other distributions can be loaded through various packages. For example, another useful distribution is the multivariate normal. By loading the `MASS` library, a user can sample from the multivariate normal distribution with the `mvrnorm` command. Even more generally, the `mvtnorm` package allows the user to compute multivariate normal and multivariate *t* probabilities, quantiles, random deviates, and densities.

Suppose we have a given probability, 0.9, and we want to know the value of a χ^2 distribution with four degrees of freedom at which the probability of being less than or equal to that value is 0.9. This calls for the *quantile function*:

```
qchisq(.9,df=4)
```

We can calculate the probability of a certain value from the *probability mass function* (PMF) for a discrete distribution. Of a Poisson distribution with intensity parameter 9, what is the probability of a count of 5?

```
dpois(5,lambda=9)
```

Although usually of less interest, for a continuous distribution, we can calculate the value of the *probability density function* (PDF) of a particular value. This has no inherent meaning, but occasionally is required. For a normal distribution with mean 4 and standard deviation 2, the density at the value 1 is given by:

```
dnorm(1,mean=4,sd=2)
```

11.2 Functions

R allows you to create your own functions with the `function` command. The `function` command uses the basic following syntax:

```
function.name <- function(INPUTS){BODY}
```

Notice that the `function` command first expects the inputs to be listed in round parentheses, while the body of the function is listed in curly braces. As can be seen, there is little constraint in what the user chooses to put into the function, so a function can be crafted to accomplish whatever the user would like.

For example, suppose we were interested in the equation $y = 2 + \frac{1}{x^2}$. We could define this function easily in R. All we have to do is specify that our input variable is x and our body is the right-hand side of the equation. This will create a function that returns values for y. We define our function, named `first.fun`, as follows:

```
first.fun<-function(x){
    y<-2+x^{-2}
    return(y)
    }
```

Although broken-up across a few lines, this is all one big command, as the curly braces ({}) span multiple lines. With the `function` command, we begin by declaring that `x` is the name of our one input, based on the fact that our equation of interest has x as the name of the input variable. Next, inside the curly braces, we assign the output `y` as being the exact function of x that we are interested in. As a last step before closing our curly braces, we use the `return` command, which tells the function what the output result is after it is called. The `return` command is useful when determining function output for a couple of reasons: First,

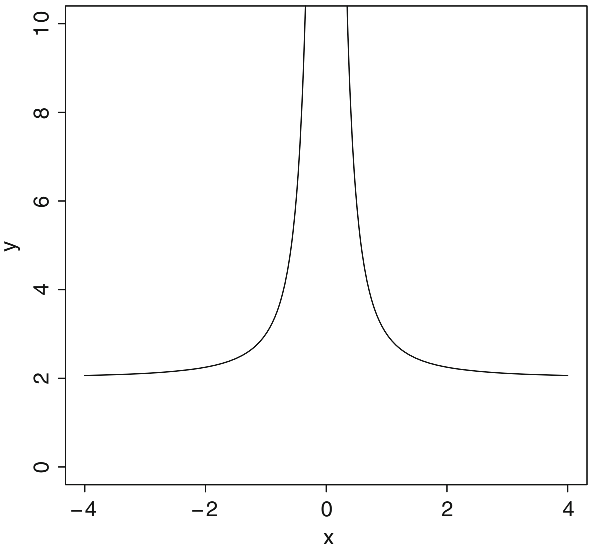

Fig. 11.1 Plot of the function $y = 2 + \frac{1}{x^2}$

if we define multiple objects inside of a command, this allows us to specify what should be printed in the output versus what was strictly internal to the function. Second, `return` allows us to report a list of items as an output, if we would like. Alternatively, we could have substituted the command `invisible`, which works the same way as `return`, except that it does not print the output on the screen, but simply stores the output.

With this function, if we want to know what value y takes when $x = 1$, we need only type: `first.fun(1)`. Since we used `return` instead of `invisible`, the printout simply reads:

```
[1] 3
```

Of course, the result that $y = 3$ here can easily be verified by hand. Similarly, we know that if $x = 3$, then $y = 2\frac{1}{9}$. To verify this fact, we can type: `first.fun(3)`. R will give us the corresponding printout:

```
[1] 2.111111
```

As a final task with this function, we can plot what it looks like by inserting a vector of x values into it. Consider the code:

```
my.x<-seq(-4,4,by=.01)
plot(y=first.fun(my.x),x=my.x,type="l",
      xlab="x",ylab="y",ylim=c(0,10))
```

This will produce the graph featured in Fig. 11.1.

As a more complicated example, consider a function that we will use as part of a larger exercise in Sect. 11.6. The following is an expected utility function for a political party in a model of how parties will choose an issue position when competing in sequential elections (Monogan 2013b, Eq. (6)):

$$\begin{aligned} EU_A(\theta_A, \theta_D) = & \Lambda\{-(m_1 - \theta_A)^2 + (m_1 - \theta_D)^2 + V\} \\ & + \delta\Lambda\{-(m_2 - \theta_A)^2 + (m_2 - \theta_D)^2\} \end{aligned} \qquad (11.1)$$

In this equation, the expected utility to a political party (party A) is the sum of two cumulative logistic distribution functions (the functions Λ), with the second downweighted by a discount term ($0 \leq \delta \leq 1$). Utility depends on the positions taken by party A and party D (θ_A and θ_D), a valence advantage to party A (V), and the issue position of the median voter in the first and second election (m_1 and m_2). This is now a function of several variables, and it requires us to use the CDF of the logistic distribution. To input this function in R, we type:

```
Quadratic.A<-function(m.1,m.2,p,delta,theta.A,theta.D){
    util.a<-plogis(-(m.1-theta.A)^2+
         (m.1-theta.D)^2+p)+
       delta*plogis(-(m.2-theta.A)^2+
         (m.2-theta.D)^2)
    return(util.a)
    }
```

The `plogis` command computes each relevant **p**robability from the **logis**tic CDF, and all of the other terms are named self-evidently from Eq. (11.1) (except that `p` refers to the valence term V). Although this function was more complicated, all we had to do is make sure we named every input and copied Eq. (11.1) entirely into the body of the function.

This more complex function, `Quadratic.A`, still behaves like our simple function. For instance, we could go ahead and supply a numeric value for every argument of the function like this:

```
Quadratic.A(m.1=.7,m.2=-.1,p=.1,delta=0,theta.A=.7,theta.D=.7)
```

Doing so prints an output of:

```
[1] 0.5249792
```

Hence, we now know that 0.52 is the expected utility to party A when the terms take on these numeric values. In isolation, this output does not mean much. In game theory, we are typically interested in how party A (or any player) can maximize its utility. Hence, if we take all of the parameters as fixed at the above values, except we allow party A to choose θ_A as it sees fit, we can visualize what A's best choice is. The following code produces the graph seen in Fig. 11.2:

```
positions<-seq(-1,1,.01)
util.A<-Quadratic.A(m.1=.7,m.2=-.1,p=.1,delta=0,
     theta.A=positions,theta.D=.7)
plot(x=positions,y=util.A,type="l",
     xlab="Party A's Position",ylab="Utility")
```

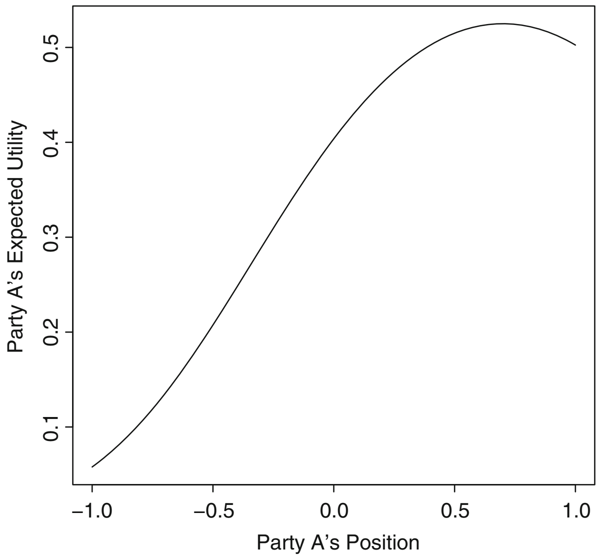

Fig. 11.2 Plot of an advantaged party's expected utility over two elections contingent on issue position

In the first line, we generate a range of positions party A may choose for θ_A. On the second line, we calculate a vector of expected utilities to party A at each of the issue positions we consider and setting the other parameters at their previously mentioned value. Lastly, we use the `plot` function to draw a line graph of these utilities. It turns out that the maximum of this function is at $\theta_A = 0.7$, so our previously mentioned utility of 0.52 is the best party A can do after all.

11.3 Loops

Loops are easy to write in R and can be used to repeat calculations that are either identical or vary only by a few parameters. The basic structure for a loop using the `for` command is:

```
for (i in 1:M) {COMMANDS}
```

In this case, `M` is the number of times the commands will be executed. R also supports loops using the `while` command which follow a similar command structure:

```
j <- 1
while(j < M) {
     COMMANDS
     j <- j + 1
     }
```

Whether a `for` loop or a `while` loop works best can vary by situation, so the user must use his or her own judgment when choosing a setup. In general, `while` loops tend to be better for problems where you would have to do something until a criterion is met (such as a convergence criterion), and `for` loops are generally better for things you want to repeat a fixed number of times. Under either structure, R will allow a wide array of commands to be included in a loop; the user's task is how to manage the loop's input and output efficiently.

As a simple demonstration of how loops work, consider an example that illustrates the law of large numbers. To do this, we can simulate **r**andom observations from the standard **norm**al distribution (easy with the `rnorm`) command. Since the standard normal has a population mean of zero, we would expect the sample mean of our simulated values to be near zero. As our sample size gets larger, the sample mean should generally be closer to the population mean of zero. A loop is perfect for this exercise: We want to repeat the calculations of simulating from the standard normal distribution and then taking the mean of the simulations. What differs from iteration to iteration is that we want our sample size to increase.

We can set this up easily with the following code:

```
set.seed(271828183)
store <- matrix(NA,1000,1)
for (i in 1:1000){
     a <- rnorm(i)
     store[i] <- mean(a)
     }
plot(store, type="h",ylab="Sample Mean",
     xlab="Number of Observations")
abline(h=0,col='red',lwd=2)
```

In the first line of this code, we call the command `set.seed`, in order to make our simulation experiment replicable. When R randomly draws numbers in any way, it uses a pseudo-random number generator, which is a list of 2.1 billion numbers that resemble random draws.[3] By choosing any number from 1 to 2,147,483,647, others should be able to reproduce our results by using the same numbers in their simulation. Our choice of 271,828,183 was largely arbitrary. In the second line of code, we create a blank vector named `store` of length 1000. This vector is where we will store our output from the loop. In the next four lines of code, we define our loop. The loop runs from 1 to 1000, with the index of each iteration being named `i`. In each iteration, our sample size is simply the value of `i`, so the first iteration simulates one observation, and the thousandth iteration simulates 1000 observations. Hence, the sample size increases with each pass of the loop. In each pass of the program, R samples from a $\mathcal{N}(0,1)$ distribution and then takes the mean of that sample. Each mean is recorded in the `i`th cell of `store`. After the loop closes, we plot our sample means against the sample size and use `abline` to draw a red line

[3]Formally, this list makes-up draws from a standard uniform distribution, which is then converted to whatever distribution we want using a quantile function.

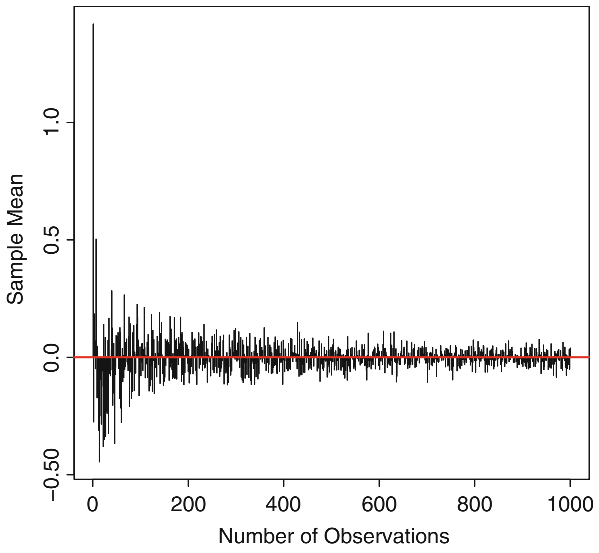

Fig. 11.3 The law of large numbers and loops in action

at the population mean of zero. The result is shown in Fig. 11.3. Indeed, our plot shows the mean of the sample converging to the true mean of zero as the sample size increases.

Loops are necessary for many types of programs, such as if you want to do a Monte Carlo analysis. In some (but not all) cases, however, loops can be slower than vectorized versions of commands. It may be worth trying the `apply` command, for instance, if it can accomplish the same goal as a loop. Which is faster often depends on the complexity of the function and the memory overhead for calculating and storing results. So if one approach is too slow or too complicated for your computer to handle, it may be worth trying the other.

11.4 Branching

R users have the option of executing commands conditional on a Boolean statement using the `if` command. This can be handy whenever the user only wants to implement something for certain cases, or if different data types should be treated differently. The basic syntax of an `if` statement is:

```
if (logical_expression) {
     expression_1
     ...
}
```

In this framework, the `logical_expression` is some Boolean statement, and `expression_1` represents whatever commands the user would like to apply whenever the logical expression is true.

As a toy example of how this works, suppose we wanted to simulate a process in which we draw two numbers from 1 to 10 (with repeated numbers allowed). The `sample` command allows us to simulate this process easily, and if we embed it in a `for` loop we can repeat the experiment several times (say, 100 trials). If we wanted to know in how many trials both numbers came up even, we could determine this with an `if` statement. Our code comes together like this:

```
even.count<-0
for (i in 1:100){
     a<-sample(c(1:10),2,replace=TRUE)
     if (sum(a%%2)==0){
          even.count<-even.count+1
          }
}
even.count
```

The first line creates a scalar named `even.count` and sets it at zero. The next line starts the loop that gives us 100 trials. The third line creates our sample of two numbers from 1 to 10, and names the sample `a`. The fourth line defines our `if` statement: We use the modulo function to find the remainder when each term in our sample is divided by two.[4] If the remainder for every term is zero, then all numbers are even and the sum is zero. In that case, we have drawn a sample in which numbers are even. Hence, when `sum(a%%2)==0` is true, then we want to add one to a running tally of how many samples of two even numbers we have. Therefore the fifth line of our code adds to our tally, but only in cases that meet our condition. (Notice that a recursive assignment for `even.count` is acceptable. R will take the old value, add one to it, and update the value that is saved.) Try this experiment yourself. As a hint, probability theory would say that 25 % of trials will yield two even numbers, on average.

Users also may make `if...else` branching statements such that one set of operations applies whenever an expression is true, and another set of operations applies when the expression is false. This basic structure looks like this:

```
if (logical_expression) {
      expression_1
      ...
} else {
      expression_2
      ...
}
```

[4]Recall from Chap. 1 that the modulo function gives us the remainder from division.

In this case, `expression_1` will only be applied in cases when the logical expression is true, and `expression_2` will only be applied in cases when the logical expression is false.

Finally, users have the option of branching even further. With cases for which the first logical expression is false, thereby calling the expressions following `else`, these cases can be branched again with another `if...else` statement. In fact, the programmer can nest as many `if...else` statements as he or she would like. To illustrate this, consider again the case when we simulate two random numbers from 1 to 10. Imagine this time we want to know not only how often we draw two even numbers, but also how often we draw two odd numbers and how often we draw one even number and one odd. One way we could address this is by keeping three running tallies (below named `even.count`, `odd.count`, and `split.count`) and adding additional branching statements:

```
even.count<-0
odd.count<-0
split.count<-0
for (i in 1:100){
     a<-sample(c(1:10),2,replace=TRUE)
     if (sum(a%%2)==0){
          even.count<-even.count+1
          } else if (sum(a%%2)==2){
               odd.count<-odd.count+1
                    } else{
                         split.count<-split.count+1
                         }
}
even.count
odd.count
split.count
```

Our `for` loop starts the same as before, but after our first `if` statement, we follow with an `else` statement. Any sample that did not consist of two even numbers is now subject to the commands under `else`, and the first command under `else` is... another `if` statement. This next `if` statement observes that if both terms in the sample are odd, then the sum of the remainders after dividing by two will be two. Hence, all samples with two odd entries now are subject to the commands of this new `if` statement, where we see that our `odd.count` index will be increased by one. Lastly, we have a final `else` statement—all samples that did not consist of two even or two odd numbers will now be subject to this final set of commands. Since these samples consist of one even and one odd number, the `split.count` index will be increased by one. Try this out yourself. Again, as a hint, probability theory indicates that on average 50 % of samples should consist of one even and one odd number, 25 % should consist of two even numbers, and 25 % should consist of two odd numbers.

11.5 Optimization and Maximum Likelihood Estimation

The optim command in R allows users to find the minimum or maximum of a function using a variety of numerical **optim**ization methods.[5] In other words, optim has several algorithms that efficiently search for the highest or lowest value of a function, several of which allow the user to constrain the possible values of input variables. While there are many uses of optimization in Political Science research, the most common is *maximum likelihood estimation* (MLE).[6]

The optim command, paired with R's simple function definition (discussed in Sect. 11.2), allows R to use readily the full flexibility of MLE. If a canned model such as those described in Chaps. 1–7 does not suit your research, and you want to derive your own estimator using MLE, then R can accommodate you.

To illustrate how programming an MLE works in R, consider a simple example—the estimator for the probability parameter (π) in a binomial distribution. By way of reminder, the motivation behind a binomial distribution is that we are interested in some trial that takes on one of two outcomes each time, and we repeat that trial multiple times. The simplest example is that we flip a coin, which will come up heads or tails on each flip. Using this example, suppose our data consist of 100 coin flips, and 43 of the flips came up heads. What is the MLE estimate of the probability of heads? By intuition or derivation, you should know that $\hat{\pi} = \frac{43}{100} = 0.43$. We still will estimate this probability in R to practice for more complicated cases, though.

To more formally define our problem, a binomial distribution is motivated by completing n Bernoulli trials, or trials that can end in either a success or a failure. We record the number of times we succeed in our n trials as y. Additionally, by definition our probability parameter (π) must be greater than or equal to zero and less than or equal to one. Our likelihood function is:

$$L(\pi|n, y) = \pi^y(1-\pi)^{n-y} \tag{11.2}$$

For ease of calculation and computation, we can obtain an equivalent result by maximizing our log-likelihood function:

$$\ell(\pi|n, y) = \log L(\pi|n, y) = y \cdot \log(\pi) + (n-y)\log(1-\pi) \tag{11.3}$$

We can easily define our log-likelihood function in R with the function command:

```
binomial.loglikelihood <- function(prob, y, n) {
   loglikelihood <- y*log(prob) + (n-y)*log(1-prob)
   return(loglikelihood)
}
```

[5]See Nocedal and Wright (1999) for a review of how these techniques work.

[6]When using maximum likelihood estimation, as an alternative to using optim is to use the maxLik package. This can be useful for maximum likelihood specifically, though optim has the advantage of being able to maximize or minimize other kinds of functions as well, when appropriate.

Now to estimate $\hat{\pi}$, we need R to find the value of π that maximizes the log-likelihood function given the data. (We call this term `prob` in the code to avoid confusion with R's use of the command `pi` to store the geometric constant.) We can do this with the `optim` command. We compute:

```
test <- optim(c(.5),                   # starting value for prob
   binomial.loglikelihood,             # the log-likelihood function
   method="BFGS",                      # optimization method
   hessian=TRUE,                       # return numerical Hessian
   control=list(fnscale=-1),           # maximize instead of minimize
   y=43, n=100)                        # the data
print(test)
```

Remember that everything after a pound sign (#) is a comment that R ignores, so these notes serve to describe each line of code. We always start with a vector of starting values with all parameters to estimate (just π in this case), name the log-likelihood function we defined elsewhere, choose our optimization method (**B**royden-**F**letcher-**G**oldfarb-**S**hanno is often a good choice), and indicate that we want R to return the numerical Hessian so we can compute standard errors later. The fifth line of code is vitally important: By default `optim` is a *minimizer*, so we have to specify `fnscale=-1` to make it a *maximizer*. Any time you use `optim` for *maximum* likelihood estimation, this line will have to be included. On the sixth line, we list our data. Often, we will call a matrix or data frame here, but in this case we need only list the values of y and n.

Our output from `print(test)` looks like the following:

```
$par
[1] 0.4300015

$value
[1] -68.33149

$counts
function gradient
      13        4

$convergence
[1] 0

$message
NULL

$hessian
          [,1]
[1,] -407.9996
```

To interpret our output: The `par` term lists the estimates of the parameters, so our estimate is $\hat{\pi} = 0.43$ (as we anticipated). The log-likelihood for our final solution is -68.33149, and is presented under `value`. The term `counts` tells us how often `optim` had to call the function and the gradient. The term `convergence` will be coded 0 if the optimization was successfully completed; any other value is an error code. The `message` item may return other necessary information from the optimizer. Lastly, the `hessian` is our matrix of second derivatives of the log-likelihood function. With only one parameter here, it is a simple 1×1 matrix. In general, if the user wants *standard errors* from an estimated maximum likelihood model, the following line will return them:

```
sqrt(diag(solve(-test$hessian)))
```

This line is based on the formula for standard errors in maximum likelihood estimation. All the user will ever need to change is to replace the word `test` with the name associated with the call to `optim`. In this case, R reports that the standard error is $\text{SE}(\hat{\pi}) = 0.0495074$.

Finally, in this case in which we have a single parameter of interest, we have the option of using our defined log-likelihood function to draw a picture of the optimization problem. Consider the following code:

```
ruler <- seq(0,1,0.01)
loglikelihood <- binomial.loglikelihood(ruler, y=43, n=100)
plot(ruler, loglikelihood, type="l", lwd=2, col="blue",
   xlab=expression(pi),ylab="Log-Likelihood",ylim=c(-300,-70),
   main="Log-Likelihood for Binomial Model")
abline(v=.43)
```

The first line defines all values that π possibly can take. The second line inserts into the log-likelihood function the vector of possible values of π, plus the true values of y and n. The third through fifth lines are a call to `plot` that gives us a line graph of the log-likelihood function. The last line draws a vertical line at our estimated value for $\hat{\pi}$. The result of this is shown in Fig. 11.4. While log-likelihood functions with many parameters usually cannot be visualized, this does remind us that the function we have defined can still be used for purposes other than optimization, if need be.

11.6 Object-Oriented Programming

This section takes a turn towards the advanced, by presenting an application in object-oriented programming. While this application does synthesize many of the other tools discussed in this chapter, novice users are less likely to use object-oriented programming than the features discussed so far. That said, for advanced uses object-oriented work can be beneficial.

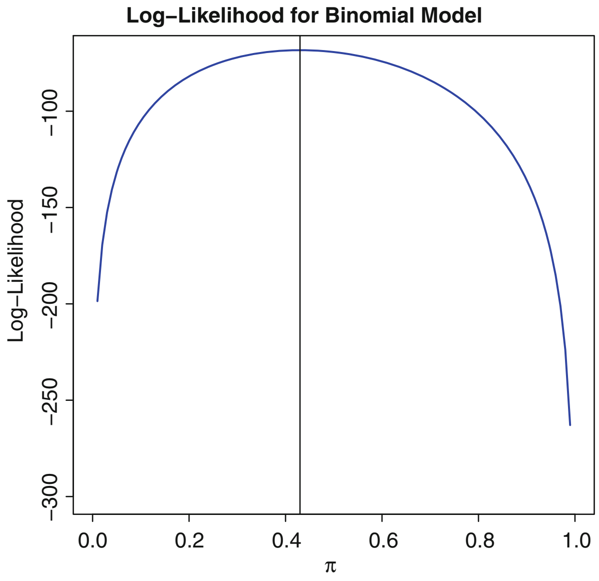

Fig. 11.4 Binomial log-likelihood across all possible values of probability parameter π when the data consist of 43 successes in 100 trials

As was mentioned in Chap. 1, R is an object-oriented environment. This means that you, as a researcher, have the opportunity to use sophisticated programming tools in your own research. In object-oriented programming, you create a *class* of item that has a variety of features. You then can create items within the class (called *objects* or *variables*) that will have unique features. In R, you can create classes from the object systems S3 or S4.

To illustrate, a "linear model" is a class of objects created by the `lm` command. It is an S3 class. Back in Chap. 6, we estimated a model that we named `mod.hours`, which was an object of the linear model class. This object had features including `coefficients`, `residuals`, and `cov.unscaled`. In all S3-class objects, we can call a feature by writing the object name, the dollar sign, and the name of the feature we want. For example, `mod.hours$coefficients` would list the vector of coefficients from that model. (S4 uses slightly different referencing, described later.) When you define your own objects, you may use either S3 or S4 object system. The following example saves our outputs in the S3 object system, though footnotes illustrate how you could use the S4 object system if you preferred.

11.6.1 Simulating a Game

As a working example of object-oriented programming, we consider a somewhat simplified version of the work presented in Monogan (2013b).[7] We touched on this model in Sect. 11.2, introducing one actor's utility function. In brief, this article develops a game theoretic model of how two parties will choose a position on an issue according to the spatial proximity model of politics. The spatial model, described somewhat in Chap. 7, assumes that issue positions can be represented as locations in space. Voters choose parties that take issue positions closer to their own position in issue space, so parties try to maximize votes by strategically choosing their issue positions.

In the game we consider, the substantive motivation is the fact that public opinion shows strong trends on certain policy issues because of demographic trends or generational differences. Issues like this include environmental policy, immigration policy, and protections for gay rights. We assume that two parties will compete in two elections. However, to consider the fact that parties should be mindful of the future and may be held to account for past issue positions, the position they take in the first election is the position they are stuck with in the second election. We also assume that the issue position of the median voter changes from one election to the next in a way that parties anticipate, since on these issues public opinion trends are often clear.

Another feature is that one party has an advantage in the first election (due to incumbency or some other factor), so for this reason the players are party *A* (a party with a non-issue valence *advantage* in the first election) and party *D* (a disadvantaged party). The second election is assumed to be less valuable than the first, because the reward is farther away. Finally, voters also consider factors unknown to the parties, so the outcome of the election is probabilistic rather than certain. In this game, then, parties try to maximize their probability of winning each of the two elections. How do they position themselves on an issue with changing public opinion?

The tricky feature of this game is that it cannot be solved algebraically (Monogan 2013b, p. 288). Hence, we turn to R to help us find solutions through simulation. The first thing we need to do is clean up and define the expected utility functions for parties *A* and *D*:

```
rm(list=ls())
Quadratic.A<-function(m.1,m.2,p,delta,theta.A,theta.D){
    util.a<-plogis(-(m.1-theta.A)^2+
         (m.1-theta.D)^2+p)+
       delta*plogis(-(m.2-theta.A)^2+
```

[7]To see the full original version of the program, consult http://hdl.handle.net/1902.1/16781. Note that here we use the S3 object family, but the original code uses the S4 family. The original program also considers alternative voter utility functions besides the quadratic proximity function listed here, such as an absolute proximity function and the directional model of voter utility (Rabinowitz and Macdonald 1989).

```
        (m.2-theta.D)^2)
    return(util.a)
    }
Quadratic.D<-function(m.1,m.2,p,delta,theta.A,theta.D){
    util.d<-(1-plogis(-(m.1-theta.A)^2+
        (m.1-theta.D)^2+p))+
      delta*(1-plogis(-(m.2-theta.A)^2+
        (m.2-theta.D)^2))
    return(util.d)
    }
```

We already considered the function `Quadratic.A` in Sect. 11.2, and it is formally defined by Eq. (11.1). `Quadratic.D` is similar in structure, but yields differing utilities. Both of these functions will be used in the simulations we do.

Our next step is to define a long function that uses our utility functions to run a simulation and produces an output formatted the way we want.[8] All of the following code is one big function definition. Comments after pound signs (#) again are included to help distinguish each party of the function body. The function, called `simulate`, simulates our game of interest and saves our results in an object of class `game.simulation`:

```
simulate<-function(v,delta,m.2,m.1=0.7,theta=seq(-1,1,.1)){

    #define internal parameters, matrices, and vectors
    precision<-length(theta)
    outcomeA<-matrix(NA,precision,precision)
    outcomeD<-matrix(NA,precision,precision)
    bestResponseA<-rep(NA,precision)
    bestResponseD<-rep(NA,precision)
    equilibriumA<- 'NA'
    equilibriumD<- 'NA'

    #matrix attributes
    rownames(outcomeA)<-colnames(outcomeA)<-rownames(outcomeD)<-
         colnames(outcomeD)<-names(bestResponseA)<-
         names(bestResponseD)<-theta

    #fill-in the utilities for all strategies for party A
    for (i in 1:precision){
            for (j in 1:precision){
                outcomeA[i,j]<-Quadratic.A(m.1,m.2,v,delta,
                     theta[i],theta[j])
                }
```

[8]If you preferred to use the S4 object system for this exercise instead, the next thing we would need to do is define the object class before defining the function. If we wanted to call our object class `simulation`, we would type: `setClass("simulation", representation(outcomeA="matrix", outcomeD="matrix", bestResponseA="numeric", bestResponseD="numeric", equilibriumA="character", equilibriumD="character"))`. The S3 object system does not require this step, as we will see when we create objects of our self-defined `game.simulation` class.

```
        }

    #utilities for party D
    for (i in 1:precision){
            for (j in 1:precision){
                outcomeD[i,j]<-Quadratic.D(m.1,m.2,v,delta,
                     theta[i],theta[j])
                }
        }

    #best responses for party A
    for (i in 1:precision){
        bestResponseA[i]<-which.max(outcomeA[,i])
        }

    #best responses for party D
    for (i in 1:precision){
        bestResponseD[i]<-which.max(outcomeD[i,])
        }

    #find the equilibria
    for (i in 1:precision){
        if (bestResponseD[bestResponseA[i]]==i){
            equilibriumA<-dimnames(outcomeA)[[1]][
                 bestResponseA[i]]
            equilibriumD<-dimnames(outcomeD)[[2]][
                 bestResponseD[bestResponseA[i]]]
                }
    }

    #save the output
    result<-list(outcomeA=outcomeA,outcomeD=outcomeD,
        bestResponseA=bestResponseA,bestResponseD=bestResponseD,
        equilibriumA=equilibriumA,equilibriumD=equilibriumD)
    class(result)<-"game.simulation"
    invisible(result)
}
```

As a general tip, when writing a long function, it is best to test code outside of the `function` wrapper. For bonus points and to get a full sense of this code, you may want to break this function into its components and see how each piece works. As the comments throughout show, each set of commands does something we normally might do in a function. Notice that when the arguments are defined, both `m.1` and `theta` are given default values. This means that in future uses, if we do not specify the values of these inputs, the function will use these defaults, but if we do specify them, then the defaults will be overridden. Turning to the arguments within the function, it starts off by defining internal parameters and setting their attributes. Every set of commands thereafter is a loop within a loop to repeat certain commands and fill output vectors and matrices.

The key addition here that is unique to anything discussed before is in the definition of the `result` term at the end of the function. Notice that in doing this, we first define the output as a `list`, in which each component is named. In this case, we already named our objects the same as the way we want to label them in the output list, but that may not always be the case.[9] We then use the `class` command to declare that our output is of the `game.simulation` class, a concept we are now creating. The `class` command formats this as an object in the S3 family. By typing `invisible(result)` at the end of the function, we know that our function will return this `game.simulation`-class object.[10]

Now that this function is defined, we can put it to use. Referring to the terms in Eq. (11.1), suppose that $V = 0.1$, $\delta = 0$, $m_1 = 0.7$, and $m_2 = -0.1$. In the code below, we create a new object called `treatment.1` using the `simulate` function when the parameters take these values, and then we ask R to print the output:

```
treatment.1<-simulate(v=0.1,delta=0.0,m.2=-0.1)
treatment.1
```

Notice that we did not specify `m.1` because 0.7 already is the default value for that parameter. The output from the printout of `treatment.1` is too lengthy to reproduce here, but on your own screen you will see that `treatment.1` is an object of the `game.simulation` class, and the printout reports the values for each attribute.

If we are only interested in a particular feature from this result, we can ask R to return only the value of a specific slot. For an S3 object, which is how we saved our result, we can call an attribute by naming the object, using the `$` symbol, and then naming the attribute we want to use. (This contrasts from S4 objects, which use the `@` symbol to call slots.) For example, if we only wanted to see what the equilibrium choices for parties *A* and *D* were, we could simply type:

```
treatment.1$equilibriumA
treatment.1$equilibriumD
```

Each of these commands returns the same output:

```
[1] "0.7"
```

So, substantively, we can conclude that the equilibrium for the game under these parameters is $\theta_A = \theta_D = 0.7$, which is the ideal point of the median voter in the first

[9]For example, in the phrase `outcomeA=outcomeA`, the term to the left of the equals sign states that this term in the list will be named `outcomeA`, and the term to the right calls the object of this name from the function to fill this spot.

[10]The code would differ slightly for the S4 object system. If we defined our `simulation` class as the earlier footnote describes, here we would replace the definition of `result` as follows: `result<-new("simulation",outcomeA=outcomeA, outcomeD=outcomeD,bestResponseA=bestResponseA, bestResponseD=bestResponseD, equilibriumA=equilibriumA, equilibriumD=equilibriumD)`. Replacing the `list` command with the `new` command assigns the output to our `simulation` class and fills the various slots all in one step. This means we can skip the extra step of using the `class` command, which we needed when using the S3 system.

election. Substantively, this should make sense because setting $\delta = 0$ is a special case when the parties are not at all concerned about winning the second election, so they position themselves strictly for the first election.[11]

To draw a contrast, what if we conducted a second simulation, but this time increased the value of δ to 0.1? We also could use finer-grained values of the positions parties could take, establishing their position out to the hundredths decimal place, rather than tenths. We could do this as follows:

```
treatment.2<-simulate(v=0.1,delta=0.1,m.2=-0.1,
     theta=seq(-1,1,.01))
treatment.2$equilibriumA
treatment.2$equilibriumD
```

We accomplished the finer precision by substituting our own vector in for `theta`. Our output values of the two equilibrium slots are again the same:

```
[1] "0.63"
```

So we know under this second treatment, the equilibrium for the game is $\theta_A = \theta_D = 0.63$. Substantively, what happened here is we increased the value of winning the second election to the parties. As a result, the parties moved their issue position a little closer to the median voter's issue preference in the second election.

11.7 Monte Carlo Analysis: An Applied Example

As an applied example that synthesizes several of the tools developed in this chapter, we now conduct a Monte Carlo analysis. The basic logic of Monte Carlo analysis is that the researcher generates data knowing the true population model. The researcher then asks whether a certain method's sample estimates have good properties, given what the population quantities are. Common treatments in a Monte Carlo experiment include: choice of estimator, sample size, distribution of predictors, variance of errors, and whether the model has the right functional form (e.g., ignoring a nonlinear relationship or omitting a predictor). By trying various treatments, a user can get a comparative sense of how well an estimator works.

In this case, we will introduce Signorino's (1999) strategic multinomial probit model as the estimator we will use in our Monte Carlo experiment. The idea behind this approach is that, when a recurring political situation can be represented with a game (such as militarized international disputes), we can develop an empirical model of the outcomes that can occur (such as war) based on the strategy of the game. Each possible outcome has a *utility function* for each player, whether the player be an individual, a country, or another actor. The utility represents how much

[11]If we had instead saved `treatment.1` as an S4 `simulation` object as the prior footnotes described, the command to call a specific attribute would instead be: `treatment.1@equilibriumA`.

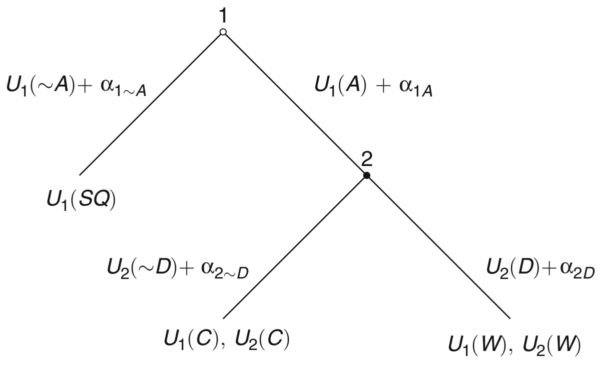

Fig. 11.5 Strategic deterrence model

benefit the player gets from the outcome. In this setup, we use observable predictors to model the utilities for each possible outcome. This kind of model is interesting because the population data-generating process typically has nonlinearities in it that are not captured by standard approaches. Hence, we will have to program our own likelihood function and use `optim` to estimate it.[12]

To motivate how the strategic multinomial probit is set up, consider a substantive model of behavior between two nations, an aggressor (1) and a target (2). This is a two-stage game: First, the aggressor decides whether to attack (A) or not (∼A); then, the target decides whether to defend (D) or not (∼D). Three observable consequences could occur: if the aggressor decides not to attack, the **status quo** holds; if the aggressor attacks, the target may choose to **capitulate**; finally, if the target defends, we have **war**. The game tree is presented in Fig. 11.5. At the end of each branch are the utilities to players (1) and (2) for **status quo** (SQ), **capitulation** (C), and **war** (W), respectively. We make the assumption that these players are rational and therefore choose the actions that maximize the resulting payoff. For example, if the target is weak, the payoff of war $U_2(W)$ would be terribly negative, and probably lower than the payoff for capitulation $U_2(C)$. If the aggressor knows this, it can deduce that in the case of attack, the target would capitulate. The aggressor therefore would know that if it chooses to attack, it will receive the payoff $U_1(C)$ (and not $U_1(W)$). If the payoff of capitulation $U_1(C)$ is bigger than the payoff of the status quo $U_1(\text{SQ})$ for the aggressor, the rational decision is to attack (and the target's rational decision is to capitulate). Of course, the goal is to get a sense of what will happen as circumstances change across different dyads of nations.

In general, we assume the utility functions respond to observable specific circumstances (the value of the disputed resources, the expected diplomatic price due to sanctions in case of aggression, the military might, and so forth). We

[12]Readers interested in doing research like this in their own work should read about the `games` package, which was developed to estimate this kind of model.

will also assume that the utilities for each country's choice are stochastic. This introduces uncertainty into our payoff model, representing the fact that this model is incomplete, and probably excludes some relevant parameters from the evaluation. In this exercise, we will consider only four predictors X_i to model the utility functions, and assume that the payoffs are linear in these variables.

In particular, we consider the following parametric form for the payoffs:

$$\begin{aligned}
U_1(\mathrm{SQ}) &= 0 \\
U_1(C) &= X_1\beta_1 \\
U_1(W) &= X_2\beta_2 \\
U_2(C) &= 0 \\
U_2(W) &= X_3\beta_3 + X_4\beta_4 \\
\alpha &\sim \mathscr{N}(0, 0.5)
\end{aligned} \tag{11.4}$$

Each nation makes its choice based on which decision will give it a bigger utility. This is based on the known information, plus an unknown private disturbance (α) for each choice. This private disturbance adds a random element to actors' decisions. We as researchers can look at past conflicts, measure the predictors (X), observe the outcome, and would like to infer: How important are X_1, X_2, X_3, and X_4 in the behavior of nation-dyads? We have to determine this based on whether each data point resulted in **status quo**, **capitulation**, or **war**.

11.7.1 Strategic Deterrence Log-Likelihood Function

We can determine estimates of each $\hat{\beta}$ using maximum likelihood estimation. First, we have to determine the probability (p) the target (2) will defend if attacked:

$$\begin{aligned}
p = P(D) &= P(U_2(D) + \alpha_{2D} > U_2(\sim D) + \alpha_{2\sim D}) \\
&= P(U_2(D) - U_2(\sim D) > \alpha_{2\sim D} - \alpha_{2D}) \\
&= \Phi(X_3\beta_3 + X_4\beta_4)
\end{aligned} \tag{11.5}$$

Where Φ is the cumulative distribution function for a standard normal distribution. If we know p, we can determine the probability (q) that the aggressor (1) will attack:

$$\begin{aligned}
q = P(A) &= P(U_1(A) + \alpha_{1A} > U_a(\sim A) + \alpha_{1\sim A}) \\
&= P(U_1(A) - U_1(\sim A) > \alpha_{1\sim A} - \alpha_{1A}) \\
&= \Phi(pX_2\beta_2 + (1-p)X_1\beta_1)
\end{aligned} \tag{11.6}$$

Notice that p is in the equation for q. This nonlinear feature of the model is not accommodated by standard canned models.

Knowing the formulae for p and q, we know the probabilities of status quo, capitulation, and war. Therefore, the likelihood function is simply the product of the probabilities of each event, raised to a dummy of whether the event happened, multiplied across all observations:

$$L(\boldsymbol{\beta}|\mathbf{y},\mathbf{X}) = \prod_{i=1}^{n}(1-q)^{D_{\text{SQ}}} \times (q(1-p))^{D_C} \times (pq)^{D_W} \tag{11.7}$$

Where D_{SQ}, D_C, and D_W are dummy variables equal to 1 if the case is status quo, capitulation, or war, respectively, and 0 otherwise. The log-likelihood function is:

$$\ell(\boldsymbol{\beta}|\mathbf{y},\mathbf{X}) = \sum_{i=1}^{n} D_{\text{SQ}}\log(1-q) + D_C \log q(1-p) + D_W \log(pq) \tag{11.8}$$

We are now ready to begin programming this into R. We begin by cleaning up and then defining our log-likelihood function as `llik`, which again includes comments to describe parts of the function's body:

```
rm(list=ls())

llik=function(B,X,Y) {
  #Separate data matrices to individual variables:
  sq=as.matrix(Y[,1])
  cap=as.matrix(Y[,2])
  war=as.matrix(Y[,3])
  X13=as.matrix(X[,1])
  X14=as.matrix(X[,2])
  X24=as.matrix(X[,3:4])

  #Separate coefficients for each equation:
  B13=as.matrix(B[1])
  B14=as.matrix(B[2])
  B24=as.matrix(B[3:4])

  #Define utilities as variables times coefficients:
  U13=X13 %*% B13
  U14=X14 %*% B14
  U24=X24 %*% B24

  #Compute probability 2 will fight (P4) or not (P3):
  P4=pnorm(U24)
  P3=1-P4

  #Compute probability 1 will attack (P2) or not (P1):
  P2=pnorm((P3*U13+P4*U14))
  P1=1-P2

  #Define and return log-likelihood function:
  lnpsq=log(P1)
  lnpcap=log(P2*P3)
```

```
    lnpwar=log(P2*P4)
    llik=sq*lnpsq+cap*lnpcap+war*lnpwar
    return(sum(llik))
  }
```

While substantially longer than the likelihood function we defined in Sect. 11.5, the idea is the same. The function still accepts parameter values, independent variables, and dependent variables, and it still returns a log-likelihood value. With a more complex model, though, the function needs to be broken into component steps. First, our data are now matrices, so the first batch of code separates the variables by equation from the model. Second, our parameters are all coefficients stored in the argument `B`, so we need to separate the coefficients by equation from the model. Third, we matrix multiply the variables by the coefficients to create the three utility terms. Fourth, we use that information to compute the probabilities that the target will defend or not. Fifth, we use the utilities, plus the probability of the target's actions, to determine the probability the aggressor will attack or not. Lastly, the probabilities of all outcomes are used to create the log-likelihood function.

11.7.2 *Evaluating the Estimator*

Now that we have defined our likelihood function, we can use it to simulate data and fit the model over our simulated data. With any Monte Carlo experiment, we need to start by defining the number of experiments we will run for a given treatment and the number of simulated data points in each experiment. We also need to define empty spaces for the outputs of each experiment. We type:

```
set.seed(3141593)
i<-100      #number of experiments
n<-1000     #number of cases per experiment
beta.qre<-matrix(NA,i,4)
stder.qre<-matrix(NA,i,4)
```

We start by using `set.seed` to make our Monte Carlo results more replicable. Here we let `i` be our number of experiments, which we set to be 100. (Though normally we might prefer a bigger number.) We use `n` as our number of cases. This allows us to define `beta.qre` and `stder.qre` as the output matrices for our estimates of the coefficients and standard errors from our models, respectively.

With this in place, we can now run a big loop that will repeatedly simulate a dataset, estimate our strategic multinomial probit model, and then record the results. The loop is as follows:

```
for(j in 1:i){
     #Simulate Causal Variables
     x1<-rnorm(n)
     x2<-rnorm(n)
     x3<-rnorm(n)
     x4<-rnorm(n)
```

```
    #Create Utilities and Error Terms
    u11<-rnorm(n,sd=sqrt(.5))
    u13<-x1
    u23<-rnorm(n,sd=sqrt(.5))
    u14<-x2
    u24<-x3+x4+rnorm(n,sd=sqrt(.5))
    pR<-pnorm(x3+x4)
    uA<-(pR*u14)+((1-pR)*u13)+rnorm(n,sd=sqrt(.5))

    #Create Dependent Variables
    sq<-rep(0,n)
    capit<-rep(0,n)
    war<-rep(0,n)
    sq[u11>=uA]<-1
    capit[u11<uA & u23>=u24]<-1
    war[u11<uA & u23<u24]<-1
    Nsq<-abs(1-sq)

    #Matrices for Input
    stval<-rep(.1,4)
    depvar<-cbind(sq,capit,war)
    indvar<-cbind(x1,x2,x3,x4)

    #Fit Model
    strat.mle<-optim(stval,llik,hessian=TRUE,method="BFGS",
         control=list(maxit=2000,fnscale=-1,trace=1),
         X=indvar,Y=depvar)

    #Save Results
    beta.qre[j,]<-strat.mle$par
    stder.qre[j,]<-sqrt(diag(solve(-strat.mle$hessian)))
}
```

In this model, we set $\beta = 1$ for every coefficient in the population model. Otherwise, Eq. (11.4) completely defines our population model. In the loop the first three batches of code all generate data according to this model. First, we generate four independent variables, each with a standard normal distribution. Second, we define the utilities according to Eq. (11.4), adding in the random disturbance terms (α) as indicated in Fig. 11.5. Third, we create the values of the dependent variables based on the utilities and disturbances. After this, the fourth step is to clean our simulated data; we define starting values for `optim` and bind the dependent and independent variables into matrices. Fifth, we use `optim` to actually estimate our model, naming the within-iteration output `strat.mle`. Lastly, the results from the model are written to the matrices `beta.qre` and `stder.qre`.

After running this loop, we can get a quick look at the average value of our estimated coefficients ($\hat{\beta}$) by typing:

```
apply(beta.qre,2,mean)
```

In this call to `apply`, we study our matrix of regression coefficients in which each row represents one of the 100 Monte Carlo experiments, and each column represents one of the four regression coefficients. We are interested in the coefficients, so we type 2 to study the columns, and then take the mean of the columns. While your results may differ somewhat from what is printed here, particularly if you did not set the seed to be the same, the output should look something like this:

```
[1] 1.0037491 1.0115165 1.0069188 0.9985754
```

In short, all four estimated coefficients are close to 1, which is good because 1 is the population value for each. If we wanted to automate our results a little more to tell us the bias in the parameters, we could type something like this:

```
deviate <- sweep(beta.qre, 2, c(1,1,1,1))
colMeans(deviate)
```

On the first line, we use the sweep command, which **sweep**s out (or subtracts) the summary statistic of our choice from a matrix. In our case, beta.qre is our matrix of coefficient estimates across 100 simulations. The 2 argument we enter separately indicates to apply the statistic by column (e.g., by coefficient) instead of by row (which would have been by experiment). Lastly, instead of listing an empirical statistic we want to subtract away, we simply list the true population parameters to subtract away. On the second line, we calculate the bias by taking the **Mean** deviation by **col**umn, or by parameter. Again, the output can vary with different seeds and number generators, but in general it should indicate that the average values are not far from the true population values. All of the differences are small:

```
[1]  0.003749060  0.011516459  0.006918824 -0.001424579
```

Another worthwhile quantity is the mean absolute error, which tells us how much an estimate differs from the population value on average. This is a bit different in that overestimates and underestimates cannot wash out (as they could with the bias calculation). An unbiased estimator still can have a large error variance and a large mean absolute error. Since we already defined deviate before, now we need only type:

```
colMeans(abs(deviate))
```

Our output shows small average absolute errors:

```
[1] 0.07875179 0.08059979 0.07169820 0.07127819
```

To get a sense of how good or bad this performance is, we should rerun this Monte Carlo experiment using another treatment for comparison. For a simple comparison, we can ask how well this model does if we increase the sample size. Presumably, our mean absolute error will decrease with a larger sample, so in each experiment we can simulate a sample of 5000 instead of 1000. To do this, rerun all of the code from this section of the chapter, but replace a single line. When defining the number of cases—the third line after the start of Sect. 11.7.2—instead of typing n<-1000, type instead: n<-5000. At the end of the program, when the mean absolute errors are reported, you will see that they are smaller. Again, they will vary from session to session, but the final output should look something like this:

```
[1] 0.03306102 0.02934981 0.03535597 0.02974488
```

As you can see, the errors are smaller than they were with 1000 observations. Our sample size is considerably bigger, so we would hope that this would be the case. This gives us a nice comparison.

With the completion of this chapter, you now should have the toolkit necessary to program in R whenever your research has unique needs that cannot be addressed by standard commands, or even user-contributed packages. With the completion of this book, you should have a sense of the broad scope of what R can do for political analysts, from data management, to basic models, to advanced programming. A true strength of R is the flexibility it offers users to address whatever complexities and original problems they may face. As you proceed to use R in your own research, be sure to continue to consult with online resources to discover new and promising capabilities as they emerge.

11.8 Practice Problems

1. Probability distributions: Calculate the probability for each of the following events:
   a. A standard normally distributed variable is larger than 3.
   b. A normally distributed variable with mean 35 and standard deviation 6 is larger than 42.
   c. $X < 0.9$ when x has the standard uniform distribution.
2. Loops: Let $h(x,n) = 1 + x + x^2 + \cdots + x^n = \sum_{i=0}^{n} x^i$. Write an R program to calculate $h(0.9, 25)$ using a `for` loop.
3. Functions: In the previous question, you wrote a program to calculate $h(x,n) = \sum_{i=0}^{n} x^i$ for $x = 0.9$ and $n = 25$. Turn this program into a more general function that takes two arguments, x and n, and returns $h(x,n)$. Using the function, determine the values of $h(0.8, 30)$, $h(0.7, 50)$, and $h(0.95, 20)$.
4. Maximum likelihood estimation. Consider an applied example of Signorino's (1999) strategic multinomial probit method. Download a subset of nineteenth century militarized interstate disputes, the Stata-formatted file `war1800.dta`, from the Dataverse (see page vii) or this chapter's online content (see page 205). These data draw from sources such as EUGene (Bueno de Mesquita and Lalman 1992) and the Correlates of War Project (Jones et al. 1996). Program a likelihood function for a model like the one shown in Fig. 11.5 and estimate the model for these real data. The three outcomes are: **war** (coded 1 if the countries went to war), **sq** (coded 1 if the status quo held), and **capit** (coded 1 if the target country capitulated). Assume that $U_1(\text{SQ})$ is driven by **peaceyrs** (number of years since the dyad was last in conflict) and **s_wt_re1** (S score for political similarity of states, weighted for aggressor's region). $U_1(W)$ is a function of **balanc** (the aggressor's military capacity relative to the combined capacity of

the dyad). $U_2(W)$ is a function of a constant and **balanc**. $U_1(C)$ is a constant, and $U_2(C)$ is zero.

a. Report your estimates and the corresponding standard errors.
b. Which coefficients are statistically distinguishable from zero?
c. Bonus: Draw the predicted probability of **war** if **balanc** is manipulated from its minimum of 0 to its maximum of 1, while **peaceyrs** and **s_wt_re1** are held at their theoretical minima of 0 and -1, respectively.

   - Just for fun: If you draw the same graph of predicted probabilities, but allow **balanc** to go all the way up to 5, you can really illustrate the kind of non-monotonic relationship that this model allows. Remember, though, that you do not want to interpret results outside of the range of your data. This is just a fun illustration for extra practice.

5. Monte Carlo analysis and optimization: Replicate the experiment on Signorino's (1999) strategic multinomial probit from Sect. 11.7. Is there much difference between your average estimated values and the population values? Do you get similar mean absolute errors to those reported in that section? How do your results compare when you try the following treatments?

   a. Suppose you decrease the standard deviation of `x1`, `x2`, `x3`, and `x4` to 0.5 (as opposed to the original treatment which assumed a standard deviation of 1). Do your estimates improve or get worse? What happens if you cut the standard deviation of these four predictors even further, to 0.25? What general pattern do you see, and why do you see it? (*Remember:* Everything else about these treatments, such as the sample size and number of experiments, needs to be the same as the original control experiment. Otherwise all else is not being held equal.)
   b. Bonus: What happens if you have an omitted variable in the model? Change your log-likelihood function to exclude `x4` from your estimation procedure, but continue to include `x4` when you simulate the data in the `for` loop. Since you only estimate the coefficients for `x1`, `x2`, and `x3`, how do your estimates in this treatment compare to the original treatment in terms of bias and mean absolute error?