Chapter 10
Linear Algebra with Programming Applications

The R language has several built-in matrix algebra commands. This proves useful for analysts who wish to write their own estimators or have other problems in linear algebra that they wish to compute using software. In some instances, it is easier to apply a formula for predictions, standard errors, or some other quantity directly rather than searching for a canned program to compute the quantity, if one exists. Matrix algebra makes it straightforward to compute these quantities yourself. This chapter introduces the syntax and available commands for conducting matrix algebra in R.

The chapter proceeds by first describing how a user can input original data by hand, as a means of creating vectors, matrices, and data frames. Then it presents several of the commands associated with linear algebra. Finally, we work through an applied example in which we estimate a linear model with ordinary least squares by programming our own estimator.

As working data throughout the chapter, we consider a simple example from the 2010 US congressional election. We model the Republican candidate's share of the two-party vote in elections for the House of Representatives ($\mathbf{y}$) in 2010. The input variables are a constant ($\mathbf{x_1}$), Barack Obama's share of the two-party presidential vote in 2008 ($\mathbf{x_2}$), and the Republican candidate's financial standing relative to the Democrat in hundreds of thousands of dollars ($\mathbf{x_3}$). For simplicity, we model the nine House races in the state of Tennessee. The data are presented in Table 10.1.

Electronic supplementary material: The online version of this chapter (doi: 10.1007/978-3-319-23446-5_10) contains supplementary material, which is available to authorized users.

J.E. Monogan III, *Political Analysis Using R*, Use R!,
DOI 10.1007/978-3-319-23446-5_10

Table 10.1 Congressional election data from Tennessee in 2010

District	Republican ($\mathbf{y}$)	Constant ($\mathbf{x_1}$)	Obama ($\mathbf{x_2}$)	Funding ($\mathbf{x_3}$)
1	0.808	1	0.290	4.984
2	0.817	1	0.340	5.073
3	0.568	1	0.370	12.620
4	0.571	1	0.340	−6.443
5	0.421	1	0.560	−5.758
6	0.673	1	0.370	15.603
7	0.724	1	0.340	14.148
8	0.590	1	0.430	0.502
9	0.251	1	0.770	−9.048

Note: Data from Monogan (2013a)

10.1 Creating Vectors and Matrices

As a first task, when assigning values to a vector or matrix, we must use the traditional assignment command (`<-`). The command `c` **c**ombines several component elements into a *vector* object.[1] So to create a vector `a` with the specific values of 3, 4, and 5:

```
a <- c(3,4,5)
```

Within `c`, all we need to do is separate each element of the vector with a comma. As a more interesting example, suppose we wanted to input three of the variables from Table 10.1: Republican share of the two-party vote in 2010, Obama's share of the two-party vote in 2008, and Republican financial advantage, all as vectors. We would type:

```
Y<-c(.808,.817,.568,.571,.421,.673,.724,.590,.251)
X2<-c(.29,.34,.37,.34,.56,.37,.34,.43,.77)
X3<-c(4.984,5.073,12.620,-6.443,-5.758,15.603,14.148,0.502,
      -9.048)
```

Whenever entering data in vector form, we should generally make sure we have entered the correct number of observations. The `length` command returns how many observations are in a vector. To check all three of our vector entries, we type:

```
length(Y); length(X2); length(X3)
```

All three of our vectors should have length 9 if they were entered correctly. Observe here the use of the semicolon (`;`). If a user prefers to place multiple commands on a single line of text, the user can separate each command with a semicolon instead of putting each command on a separate line. This allows simple commands to be stored more compactly.

[1] Alternatively, when the combined elements are complex objects, `c` instead creates a `list` object.

If we want to create a vector that follows a **rep**eating pattern, we can use the `rep` command. For instance, in our model of Tennessee election returns, our constant term is simply a 9×1 vector of 1s:

```
X1 <- rep(1, 9)
```

The first term within `rep` is the term to be repeated, and the second term is the number of times it should be repeated.

Sequential vectors also are simple to create. A colon (`:`) prompts R to list sequential integers from the starting to the stopping point. For instance, to create an index for the congressional district of each observation, we would need a vector that contains values counting from 1 to 9:

```
index <- c(1:9)
```

As an aside, a more general command is the `seq` command, which allows us to define the intervals of a **seq**uence, as well as starting and ending values. For example, if for some reason we wanted to create a sequence from -2 to 1 in increments of 0.25, we would type:

```
e <- seq(-2, 1, by=0.25)
```

Any vector we create can be printed to the screen simply by typing the name of the vector. For instance, if we simply type `Y` into the command prompt, our vector of Republican vote share is printed in the output:

```
[1] 0.808 0.817 0.568 0.571 0.421 0.673 0.724
[8] 0.590 0.251
```

In this case, the nine values of the vector `Y` are printed. The number printed in the square braces at the start of each row offers the index of the first element in that row. For instance, the eighth observation of `Y` is the value 0.590. For longer vectors, this helps the user keep track of the indices of the elements within the vector.

10.1.1 Creating Matrices

Turning to matrices, we can create an object of class *matrix* in several possible ways. First, we could use the `matrix` command: In the simplest possible case, suppose we wanted to create a matrix with all of the values being the same. To create a 4×4 matrix `b` with every value equaling 3:

```
b <- matrix(3, ncol=4, nrow=4, byrow=FALSE)
```

The syntax of the `matrix` command first calls for the elements that will define the matrix: In this case, we listed a single scalar, so this number was repeated for all matrix cells. Alternatively, we could list a vector instead that includes enough entries to fill the entire matrix. We then need to specify the number of columns (`ncol`) and rows (`nrow`). Lastly, the `byrow` argument is set to `FALSE` by default. With the `FALSE` setting, R fills the matrix column-by-column. A `TRUE` setting fills

the matrix row-by-row instead. As a rule, whenever creating a matrix, type the name of the matrix (b in this case) into the command console to see if the way R input the data matches what you intended.

A second simple option is to create a matrix from vectors that have already been entered. We can **bind** the vectors together as **c**olumn vectors using the cbind command and as **r**ow vectors using the rbind command. For example, in our model of Tennessee election returns, we will need to create a matrix of all input variables in which variables define the columns and observations define the rows. Since we have defined our three variable vectors (and each vector is ordered by observation), we can simply create such a matrix using cbind:

```
X<-cbind(1,X2,X3)
X
```

The cbind command treats our vectors as columns in a matrix. This is what we want since a predictor matrix defines rows with observations and columns with variables. The 1 in the cbind command ensures that all elements of the first column are equal to the constant 1. (Of course, the way we designed X1, we also could have included that vector.) When typing X in the console, we get the printout:

```
          X2      X3
 [1,] 1 0.29  4.984
 [2,] 1 0.34  5.073
 [3,] 1 0.37 12.620
 [4,] 1 0.34 -6.443
 [5,] 1 0.56 -5.758
 [6,] 1 0.37 15.603
 [7,] 1 0.34 14.148
 [8,] 1 0.43  0.502
 [9,] 1 0.77 -9.048
```

These results match the data we have in Table 10.1, so our covariate matrix should be ready when we are ready to estimate our model.

Just to illustrate the rbind command, R easily would combine the vectors as rows as follows:

```
T<-rbind(1,X2,X3)
T
```

We do not format data like this, but to see how the results look, typing T in the console results in the printout:

```
     [,1]   [,2]   [,3]    [,4]    [,5]    [,6]    [,7]
   1.000 1.000  1.00  1.000  1.000  1.000  1.000
X2 0.290 0.340  0.37  0.340  0.560  0.370  0.340
X3 4.984 5.073 12.62 -6.443 -5.758 15.603 14.148
     [,8]    [,9]
   1.000  1.000
X2 0.430  0.770
X3 0.502 -9.048
```

Hence, we see that each variable vector makes a row and each observation makes a column. In the printout, when R lacks the space to print an entire row on a single line, it wraps all rows at once, thus presenting columns 8 and 9 later.

Third, we could create a matrix by using subscripting. (Additional details on vector and matrix subscripts are presented in Sect. 10.1.3.) Sometimes when creating a matrix, the user will know all of the values up front, but on other occasions a user must create a matrix and then fill in the values later. In the latter case, it is a good idea to create "blanks" to fill in by designating every cell as missing (or NA). The nice feature of this is that a user can easily identify a cell that was never filled-in. By contrast, if a matrix is created with some default numeric value, say 0, then later on it is impossible to distinguish a cell that has a default 0 from one with a true value of 0. So if we wanted to create a 3×3 matrix named `blank` to fill in later, we would write:

```
blank <- matrix(NA, ncol=3, nrow=3)
```

If we then wanted to assign the value of 8 to the first row, third column element, we would write:

```
blank[1,3] <- 8
```

If we then wanted to insert the value π $(= 3.141592\ldots)$ into the second row, first column entry, we would write:

```
blank[2,1] <- pi
```

If we wanted to use our previously defined vector $\mathbf{a} = (3, 4, 5)'$ to define the second column, we would write:

```
blank[,2] <- a
```

We then could check our progress simply by typing `blank` into the command prompt, which would print:

```
         [,1] [,2] [,3]
[1,]       NA    3    8
[2,] 3.141593    4   NA
[3,]       NA    5   NA
```

To the left of the matrix, the row terms are defined. At the top of the matrix, the column terms are defined. Notice that four elements are still coded NA because a replacement value was never offered.

Fourth, in contrast to filling-in matrices after creation, we also may know the values we want at the time of creating the matrix. As a simple example, to create a 2×2 matrix $\mathbf{W}$ in which we list the value of each cell column-by-column:

```
W <- matrix(c(1,2,3,4), ncol=2, nrow=2)
```

Notice that our first argument is now a vector because we want to provide unique elements for each of the cells in the matrix. With four elements in the vector, we have the correct number of entries for a 2×2 matrix. Also, in this case, we ignored the `byrow` argument because the default is to fill-in by columns. By contrast, if we wanted to list the cell elements row-by-row in matrix $\mathbf{Z}$, we would simply set `byrow` to `TRUE`:

```
Z <- matrix(c(1,2,3,4), ncol=2, nrow=2, byrow=TRUE)
```

Type `W` and `Z` into the console and observe how the same vector of cell entries has been reorganized going from one cell to another.

Alternatively, suppose we wanted to create a 10×10 matrix $\mathbf{N}$ where every cell entry was a **r**andom draw from a **norm**al distribution with mean 10 and standard deviation 2, or $\mathcal{N}(10, 4)$:

```
N <- matrix(rnorm(100, mean=10, sd=2), nrow=10, ncol=10)
```

Because `rnorm` returns an object of the vector class, we are again listing a vector to create our cell entries. The `rnorm` command is drawing from the normal distribution with our specifications 100 times, which provides the number of cell entries we need for a 10×10 matrix.

Fifth, on many occasions, we will want to create a *diagonal matrix* that contains only elements on its main **diag**onal (from the top left to the bottom right), with zeros in all other cells. The command `diag` makes it easy to create such a matrix:

```
D <- diag(c(1:4))
```

By typing `D` into the console, we now see how this diagonal matrix appears:

```
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    2    0    0
[3,]    0    0    3    0
[4,]    0    0    0    4
```

Additionally, if one inserts a square matrix into the `diag` command, it will return a vector from the matrix's diagonal elements. For example, in a variance-covariance matrix, the diagonal elements are variances and can be extracted quickly in this way.

10.1.2 Converting Matrices and Data Frames

A final means of creating a matrix object is with the `as.matrix` command. This command can take an object of the *data frame* class and convert it to an object of the *matrix* class. For a data frame called `mydata`, for example, the command `mydata.2 <- as.matrix(mydata)` would coerce the data into matrix form, making all of the ensuing matrix operations applicable to the data. Additionally, typing `mydata.3 <- as.matrix(mydata[,4:8])` would only take columns four through eight of the data frame and coerce those into a matrix, allowing the user to subset the data while creating a matrix object.

Similarly, suppose we wanted to take an object of the *matrix* class and create an object of the *data frame* class, perhaps our Tennessee electoral data from Table 10.1. In this case, the `as.data.frame` command will work. If we simply wanted to create a data frame from our covariate matrix $\mathbf{X}$, we could type:

```
X.df <- as.data.frame(X)
```

If we wanted to create a data frame using all of the variables from Table 10.1, including the dependent variable and index, we could type:

```
tennessee <- as.data.frame(cbind(index,Y,X1,X2,X3))
```

In this case, the `cbind` command embedded in `as.data.frame` defines a matrix, which is immediately converted to a data frame. In all, then, if a user wishes to input his or her own data into R, the most straightforward way will be to define variable vectors, bind them together as columns, and convert the matrix to a data frame.

10.1.3 Subscripting

As touched on in the matrix creation section, calling elements of vectors and matrices by their subscripts can be useful for either extracting necessary information or for making assignments. To call a specific value, we can index a vector **y** by the *n*th element, using `Y[n]`. So the third element of vector **y**, or y_3, is called by:

```
Y[3]
```

To index an $n \times k$ matrix **X** for the value X_{ij}, where i represents the row and j represents the column, use the syntax `X[i,j]`. If we want to select all values of the jth column of **X**, we can use `X[,j]`. For example, to return the second column of matrix **X**, type:

```
X[,2]
```

Alternatively, if a column has a name, then the name (in quotations) can be used to call the column as well. For example:

```
X[,"X2"]
```

Similarly, if we wish to select all of the elements of the ith row of **X**, we can use `X[i,]`. For the first row of matrix **X**, type:

```
X[1,]
```

Alternatively, we could use a row name as well, though the matrix **X** does not have names assigned to the rows.

If we wish, though, we can create **row names** for matrices. For example:

```
rownames(X)<-c("Dist. 1","Dist. 2","Dist. 3","Dist. 4",
     "Dist. 5","Dist. 6","Dist. 7","Dist. 8","Dist. 9")
```

Similarly, the command `colnames` allows the user to define **column names**. To simply type `rownames(X)` or `colnames(X)` without making an assignment, R will print the row or column names saved for a matrix.

10.2 Vector and Matrix Commands

Now that we have a sense of creating vectors and matrices, we turn to commands that either extract information from these objects or allow us to conduct linear algebra with them. As was mentioned before, after entering a vector or matrix into R, in addition to printing the object onto the screen for visual inspection, it is also a good idea to check and make sure the dimensions of the object are correct. For instance, to obtain the **length** of a vector **a**:

```
length(a)
```

Similarly, to obtain the **dim**ensions of a matrix, type:

```
dim(X)
```

The `dim` command first prints the number of rows, then the number of columns. Checking these dimensions offers an extra assurance that the data in a matrix have been entered correctly.

For vectors, the elements can be treated as data, and summary quantities can be extracted. For instance, to add up the **sum** of a vector's elements:

```
sum(X2)
```

Similarly, to take the **mean** of a vector's elements (in this example, the mean of Obama's 2008 vote by district):

```
mean(X2)
```

And to take the **var**iance of a vector:

```
var(X2)
```

Another option we have for matrices and vectors is that we can sample from a given object with the command `sample`. Suppose we want a sample of ten numbers from **N**, our 10×10 matrix of random draws from a normal distribution:

```
set.seed(271828183)
N <- matrix(rnorm(100, mean=10, sd=2), nrow=10, ncol=10)
s <- sample(N,10)
```

The first command, `set.seed` makes this simulation more replicable. (See Chap. 11 for more detail about this command.) This gives us a vector named **s** of ten random elements from **N**. We also have the option of applying the `sample` command to vectors.[2]

The `apply` command is often the most efficient way to do vectorized calculations. For example, to calculate the means for all the columns in our Tennessee data matrix **X**:

```
apply(X, 2, mean)
```

[2]For readers interested in bootstrapping, which is one of the most common applications of sampling from data, the most efficient approach will be to install the `boot` package and try some of the examples that library offers.

In this case, the first argument lists the matrix to analyze. The second argument, `2` tells R to apply a mathematical function along the *columns* of the matrix. The third argument, `mean` is the function we want to apply to each column. If we wanted the mean of the *rows* instead of the columns, we could use a `1` as the second argument instead of a 2. Any function defined in R can be used with the `apply` command.

10.2.1 Matrix Algebra

Commands that are more specific to matrix algebra are also available. Whenever the user wants to save the output of a vector or matrix operation, the assignment command must be used. For example, if for some reason we needed the difference between each Republican share of the two-party vote and Obama's share of the two-party vote, we could assign vector **m** to the difference of the two vectors by typing:

```
m <- Y - X2
```

With arithmetic operations, R is actually pretty flexible, but a word on how the commands work may avoid future confusion. For addition (`+`) and subtraction (`-`): If the arguments are two vectors of the same length (as is the case in computing vector **m**), then R computes regular vector addition or subtraction where each element is added to or subtracted from its corresponding element in the other vector. If the arguments are two matrices of the same size, matrix addition or subtraction applies where each entry is combined with its corresponding entry in the other matrix. If one argument is a scalar, then the scalar will be added to each element of the vector or matrix.

Note: If two vectors of uneven length or two matrices of different size are added, an error message results, as these arguments are non-conformable. To illustrate this, if we attempted to add our sample of ten numbers from before, **s**, to our vector of Obama's 2008 share of the vote, $\mathbf{x_2}$, as follows:

```
s + X2
```

In this case we would receive an output that we should ignore, followed by an error message in red:

```
[1] 11.743450  8.307068 11.438161 14.251645 10.828459
[6] 10.336895  9.900118 10.092051 12.556688  9.775185
Warning message:
In s + X2 : longer object length is not a multiple of shorter
object length
```

This warning message tells us the output is nonsense. Since the vector `s` has length 10, while the vector `X2` has length 9, what R has done here is added the *tenth* element of `s` to the *first* element of `X2` and used this as the tenth element of the output. The error message that follows serves as a reminder that a garbage input that breaks the rules of linear algebra produces a garbage output that no one should use.

By contrast, `*`, `/`, `^` (exponents), `exp` (exponential function), and `log` all apply the relevant operation to each scalar entry in the vector or matrix. For our matrix of predictors from Tennessee, for example, try the command:

```
X.sq <- X^2
```

This would return a matrix that squares each separate cell in the matrix **X**. By a similar token, the simple multiplication operation (`*`) performs multiplication element by element. Hence, if two vectors are of the same length or two matrices of the same size, the output will be a vector or matrix of the same size where each element is the product of the corresponding elements. Just as an illustration, let us multiply Obama's share of the two-party vote by Republican financial advantage in a few ways. (The quantities may be silly, but this offers an example of how the code works.) Try for example:

```
x2.x3 <- X2*X3
```

The output vector is:

```
[1]   1.44536   1.72482   4.66940 -2.19062 -3.22448
[6]   5.77311   4.81032   0.21586 -6.96696
```

This corresponds to element-by-element scalar multiplication.

More often, the user actually will want to conduct proper matrix multiplication. In R, matrix multiplication is conducted by the `%*%` operation. So if we had instead multiplied our two vectors of Obama vote share and Republican financial advantage in this way:

```
x2.x3.inner <- X2%*%X3
```

R would now return the *inner product*, or dot product ($\mathbf{x_2} \cdot \mathbf{x_3}$). This equals 6.25681 in our example. Another useful quantity for vector multiplication is the *outer product*, or tensor product ($\mathbf{x_2} \otimes \mathbf{x_3}$). We can obtain this by *transposing* the second vector in our code:

```
x2.x3.outer <- X2%*%t(X3)
```

The output of this command is a 9×9 matrix. To obtain this quantity, the **t**ranspose command (`t`) was used.[3] In matrix algebra, we may need to turn row vectors to column vectors or vice versa, and the **t**ranspose operation accomplishes this. Similarly, when applied to a matrix, every row becomes a column and every column becomes a row.

As one more example of matrix multiplication, the input variable matrix **X** has size 9×3, and the matrix **T** that lets variables define rows is a 3×9 matrix. In this case, we could create the matrix $\mathbf{P} = \mathbf{TX}$ because the number of columns of **T** is the same as the number of rows of **X**. We therefore can say that the matrices are *conformable for multiplication* and will result in a matrix of size 3×3. We can compute this as:

[3] An alternate syntax would have been `X2%o%X3`.

```
P <- T%*%X
```

Note that if our matrices are not conformable for multiplication, then R will return an error message.[4]

Beyond matrix multiplication and the transpose operation, another important quantity that is unique to matrix algebra is the **det**erminant of a square matrix (or a matrix that has the same number of rows as columns). Our matrix **P** is square because it has three rows and three columns. One reason to calculate the determinant is that a square matrix has an inverse only if the determinant is nonzero.[5] To compute the determinant of **P**, we type:

```
det(P)
```

This yields a value of 691.3339. We therefore know that **P** has an inverse.

For a square matrix that can be inverted, the *inverse* is a matrix that can be multiplied by the original to produce the identity matrix. With the `solve` command, R will either **solve** for the inverse of the matrix or convey that the matrix is noninvertible. To try this concept, type:

```
invP <- solve(P)
invP%*%P
```

On the first line, we created $\mathbf{P}^{-1}$, which is the inverse of **P**. On the second line, we multiply $\mathbf{P}^{-1}\mathbf{P}$ and the printout is:

```
                              X2            X3
   1.000000e+00 -6.106227e-16 -6.394885e-14
X2 1.421085e-14  1.000000e+00  1.278977e-13
X3 2.775558e-17 -2.255141e-17  1.000000e+00
```

This is the basic form of the identity matrix, with values of 1 along the main diagonal (running from the top left to bottom right), and values of 0 off the diagonal. While the off-diagonal elements are not listed exactly as zero, this can be attributed to rounding error on R's part. The scientific notation for the second row, first column element for example means that the first 13 digits after the decimal place are zero, followed by a 1 in the 14th digit.

[4] A unique variety of matrix multiplication is called the Kronecker product ($\mathbf{H} \otimes \mathbf{L}$). The Kronecker product has useful applications in the analysis of panel data. See the `kronecker` command in R for more information.

[5] As another application in statistics, the likelihood function for a multivariate normal distribution also calls on the determinant of the covariance matrix.

10.3 Applied Example: Programming OLS Regression

To illustrate the various matrix algebra operations that R has available, in this section we will work an applied example by computing the ordinary least squares (OLS) estimator with our own program using real data.

10.3.1 Calculating OLS by Hand

First, to motivate the background to the problem, consider the formulation of the model and how we would estimate this by hand. Our population linear regression model for vote shares in Tennessee is: $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \mathbf{u}$. In this model, $\mathbf{y}$ is a vector of the Republican vote share in each district, $\mathbf{X}$ is a matrix of predictors (including a constant, Obama's share of the vote in 2008, and the Republican's financial advantage relative the Democrat), $\boldsymbol{\beta}$ consists of the partial coefficient for each predictor, and $\mathbf{u}$ is a vector of disturbances. We estimate $\hat{\boldsymbol{\beta}} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$, yielding the sample regression function $\hat{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\beta}}$.

To start computing by hand, we have to define $\mathbf{X}$. Note that we must include a vector of 1s in order to estimate an intercept term. In scalar form, our population model is: $y_i = \beta_1 x_{1i} + \beta_2 x_{2i} + \beta_3 x_{3i} + u_i$, where $x_{1i} = 1$ for all i. This gives us the predictor matrix:

$$\mathbf{X} = \begin{bmatrix} 1 & 0.29 & 4.984 \\ 1 & 0.34 & 5.073 \\ 1 & 0.37 & 12.620 \\ 1 & 0.34 & -6.443 \\ 1 & 0.56 & -5.758 \\ 1 & 0.37 & 15.603 \\ 1 & 0.34 & 14.148 \\ 1 & 0.43 & 0.502 \\ 1 & 0.77 & -9.048 \end{bmatrix}$$

Next, premultiply $\mathbf{X}$ by its transpose:

$$\mathbf{X}'\mathbf{X} = \begin{bmatrix} 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 \\ 0.290 & 0.340 & 0.370 & 0.340 & 0.560 & 0.370 & 0.340 & 0.430 & 0.770 \\ 4.984 & 5.073 & 12.620 & -6.443 & -5.758 & 15.603 & 14.148 & 0.502 & -9.048 \end{bmatrix} \begin{bmatrix} 1 & 0.29 & 4.984 \\ 1 & 0.34 & 5.073 \\ 1 & 0.37 & 12.620 \\ 1 & 0.34 & -6.443 \\ 1 & 0.56 & -5.758 \\ 1 & 0.37 & 15.603 \\ 1 & 0.34 & 14.148 \\ 1 & 0.43 & 0.502 \\ 1 & 0.77 & -9.048 \end{bmatrix}$$

which works out to:

$$\mathbf{X}'\mathbf{X} = \begin{bmatrix} 9.00000 & 3.81000 & 31.68100 \\ 3.81000 & 1.79610 & 6.25681 \\ 31.68100 & 6.25681 & 810.24462 \end{bmatrix}$$

We also need $\mathbf{X}'\mathbf{y}$:

$$\mathbf{X}'\mathbf{y} = \begin{bmatrix} 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 \\ 0.290 & 0.340 & 0.370 & 0.340 & 0.560 & 0.370 & 0.340 & 0.430 & 0.770 \\ 4.984 & 5.073 & 12.620 & -6.443 & -5.758 & 15.603 & 14.148 & 0.502 & -9.048 \end{bmatrix} \begin{bmatrix} 0.808 \\ 0.817 \\ 0.568 \\ 0.571 \\ 0.421 \\ 0.673 \\ 0.724 \\ 0.590 \\ 0.251 \end{bmatrix}$$

which works out to:

$$\mathbf{X}'\mathbf{y} = \begin{bmatrix} 5.4230 \\ 2.0943 \\ 28.0059 \end{bmatrix}$$

Inverse by Hand

The last quantity we need is the inverse of $\mathbf{X}'\mathbf{X}$. Doing this by hand, we can solve by Gauss-Jordan elimination:

$$[\mathbf{X}'\mathbf{X}|\mathbf{I}] = \left[\begin{array}{ccc|ccc} 9.00000 & 3.81000 & 31.68100 & 1.00000 & 0.00000 & 0.00000 \\ 3.81000 & 1.79610 & 6.25681 & 0.00000 & 1.00000 & 0.00000 \\ 31.68100 & 6.25681 & 810.24462 & 0.00000 & 0.00000 & 1.00000 \end{array}\right]$$

Divide row 1 by 9:

$$\left[\begin{array}{ccc|ccc} 1.00000 & 0.42333 & 3.52011 & 0.11111 & 0.00000 & 0.00000 \\ 3.81000 & 1.79610 & 6.25681 & 0.00000 & 1.00000 & 0.00000 \\ 31.68100 & 6.25681 & 810.24462 & 0.00000 & 0.00000 & 1.00000 \end{array}\right]$$

Subtract 3.81 times row 1 from row 2:

$$\left[\begin{array}{ccc|ccc} 1.00000 & 0.42333 & 3.52011 & 0.11111 & 0.00000 & 0.00000 \\ 0.00000 & 0.18320 & -7.15481 & -0.42333 & 1.00000 & 0.00000 \\ 31.68100 & 6.25681 & 810.24462 & 0.00000 & 0.00000 & 1.00000 \end{array}\right]$$

Subtract 31.681 times row 1 from row 3:

$$\left[\begin{array}{ccc|ccc} 1.00000 & 0.42333 & 3.52011 & 0.11111 & 0.00000 & 0.00000 \\ 0.00000 & 0.18320 & -7.15481 & -0.42333 & 1.00000 & 0.00000 \\ 0.00000 & -7.15481 & 698.72402 & -3.52011 & 0.00000 & 1.00000 \end{array}\right]$$

Divide row 2 by 0.1832:

$$\left[\begin{array}{ccc|ccc} 1.00000 & 0.42333 & 3.52011 & 0.11111 & 0.00000 & 0.00000 \\ 0.00000 & 1.00000 & -39.05464 & -2.31077 & 5.45852 & 0.00000 \\ 0.00000 & -7.15481 & 698.72402 & -3.52011 & 0.00000 & 1.00000 \end{array}\right]$$

Add 7.15481 times row 2 to row 3:

$$\left[\begin{array}{ccc|ccc} 1.00000 & 0.42333 & 3.52011 & 0.11111 & 0.00000 & 0.00000 \\ 0.00000 & 1.00000 & -39.05464 & -2.31077 & 5.45852 & 0.00000 \\ 0.00000 & 0.00000 & 419.29549 & -20.05323 & 39.05467 & 1.00000 \end{array}\right]$$

Divide row 3 by 419.29549:

$$\left[\begin{array}{ccc|ccc} 1.00000 & 0.42333 & 3.52011 & 0.11111 & 0.00000 & 0.00000 \\ 0.00000 & 1.00000 & -39.05464 & -2.31077 & 5.45852 & 0.00000 \\ 0.00000 & 0.00000 & 1.00000 & -0.04783 & 0.09314 & 0.00239 \end{array}\right]$$

Add 39.05464 times row 3 to row 2:

$$\left[\begin{array}{ccc|ccc} 1.00000 & 0.42333 & 3.52011 & 0.11111 & 0.00000 & 0.00000 \\ 0.00000 & 1.00000 & 0.00000 & -4.17859 & 9.09607 & 0.09334 \\ 0.00000 & 0.00000 & 1.00000 & -0.04783 & 0.09314 & 0.00239 \end{array}\right]$$

Subtract 3.52011 times row 3 from row 1:

$$\left[\begin{array}{ccc|ccc} 1.00000 & 0.42333 & 0.00000 & 0.27946 & -0.32786 & -0.00841 \\ 0.00000 & 1.00000 & 0.00000 & -4.17859 & 9.09607 & 0.09334 \\ 0.00000 & 0.00000 & 1.00000 & -0.04783 & 0.09314 & 0.00239 \end{array}\right]$$

Subtract .42333 times row 2 from row 1:

$$\left[\begin{array}{ccc|ccc} 1.00000 & 0.00000 & 0.00000 & 2.04838 & -4.17849 & -0.04792 \\ 0.00000 & 1.00000 & 0.00000 & -4.17859 & 9.09607 & 0.09334 \\ 0.00000 & 0.00000 & 1.00000 & -0.04783 & 0.09314 & 0.00239 \end{array}\right] = [\mathbf{I}|(\mathbf{X}'\mathbf{X})^{-1}]$$

As a slight wrinkle in these hand calculations, we can see that $(\mathbf{X}'\mathbf{X})^{-1}$ is a little off due to rounding error. It should actually be a symmetric matrix.

Final Answer by Hand

If we postmultiply $(\mathbf{X}'\mathbf{X})^{-1}$ by $\mathbf{X}'\mathbf{y}$ we get:

$$(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y} = \begin{bmatrix} 2.04838 & -4.17849 & -0.04792 \\ -4.17859 & 9.09607 & 0.09334 \\ -0.04783 & 0.09314 & 0.00239 \end{bmatrix} \begin{bmatrix} 5.4230 \\ 2.0943 \\ 28.0059 \end{bmatrix} = \begin{bmatrix} 1.0178 \\ -1.0018 \\ 0.0025 \end{bmatrix} = \hat{\boldsymbol{\beta}}$$

Or in scalar form: $\hat{y}_i = 1.0178 - 1.0018x_{2i} + 0.0025x_{3i}$.

10.3.2 Writing An OLS Estimator in R

Since inverting the matrix took so many steps and even ran into some rounding error, it will be easier to have R do some of the heavy lifting for us. (As the number of observations or variables rises, we will find the computational assistance even more valuable.) In order to program our own OLS estimator in R, the key commands we require are:

- Matrix multiplication: `%*%`
- Transpose: `t`
- Matrix inverse: `solve`

Knowing this, it is easy to program an estimator for $\hat{\boldsymbol{\beta}} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$.

First we must enter our variable vectors using the data from Table 10.1 and combine the input variables into a matrix. If you have not yet entered these, type:

```
Y<-c(.808,.817,.568,.571,.421,.673,.724,.590,.251)
X1 <- rep(1, 9)
X2<-c(.29,.34,.37,.34,.56,.37,.34,.43,.77)
X3<-c(4.984,5.073,12.620,-6.443,-5.758,15.603,14.148,0.502,
     -9.048)
X<-cbind(X1,X2,X3)
```

To estimate OLS with our own program, we simply need to translate the estimator $(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$ into R syntax:

```
beta.hat<-solve(t(X)%*%X)%*%t(X)%*%Y
beta.hat
```

Breaking this down a bit: The `solve` command leads off because the first quantity is an inverse, $(\mathbf{X}'\mathbf{X})^{-1}$. Within the call to `solve`, the first argument must be transposed (hence the use of `t`) and then it is postmultiplied by the non-transposed covariate matrix (hence the use of `%*%`). We follow up by postmultiplying the transpose of $\mathbf{X}$, then postmultiplying the vector of outcomes ($\mathbf{y}$). When we print our results, we get:

```
             [,1]
    1.017845630
X2 -1.001809341
X3  0.002502538
```

These are the same results we obtained by hand, despite the rounding discrepancies we encountered when inverting on our own. Writing the program in R, however, simultaneously gave us full control over the estimator, while being much quicker than the hand operations.

Of course an even faster option would be to use the canned `lm` command we used in Chap. 6:

```
tennessee <- as.data.frame(cbind(Y,X2,X3))
lm(Y~X2+X3, data=tennessee)
```

This also yields the exact same results. In practice, whenever computing OLS estimates, this is almost always the approach we will want to take. However, this procedure has allowed us to verify the usefulness of R's matrix algebra commands. If the user finds the need to program a more complex estimator for which there is not a canned command, this should offer relevant tools with which to accomplish this.

10.3.3 Other Applications

With these basic tools in hand, users should now be able to begin programming with matrices and vectors. Some intermediate applications of this include: computing standard errors from models estimated with `optim` (see Chap. 11) and predicted values for complicated functional forms (see the code that produced Fig. 9.3). Some of the more advanced applications that can benefit from using these tools include: feasible generalized least squares (including weighted least squares), optimizing likelihood functions for multivariate normal distributions, and generating correlated data using **Chol**esky decomposition (see the `chol` command). Very few programs offer the estimation and programming flexibility that R does whenever matrix algebra is essential to the process.

10.4 Practice Problems

For these practice problems, consider the data on congressional elections in Arizona in 2010, presented below in Table 10.2:

Table 10.2 Congressional election data from Arizona in 2010

District	Republican ($\mathbf{y}$)	Constant ($\mathbf{x_1}$)	Obama ($\mathbf{x_2}$)	Funding ($\mathbf{x_3}$)
1	0.50	1	0.44	−9.11
2	0.65	1	0.38	8.31
3	0.52	1	0.42	8.00
4	0.28	1	0.66	−8.50
5	0.52	1	0.47	−7.17
6	0.67	1	0.38	5.10
7	0.45	1	0.57	−5.32
8	0.47	1	0.46	−18.64

Note: Data from Monogan (2013a)

1. Create a vector consisting of the Republican share of the two-party vote ($\mathbf{y}$) in Arizona's eight congressional districts. Using any means you prefer, create a matrix, $\mathbf{X}$, in which the eight rows represent Arizona's eight districts, and the three columns represent a constant vector of ones ($\mathbf{x_1}$), Obama's 2008 share of the vote ($\mathbf{x_2}$), and the Republican's financial balance ($\mathbf{x_3}$). Print out your vector and matrix, and ask R to compute the vector's length and matrix's dimensions. Are all of these outputs consistent with the data you see in Table 10.2?
2. Using your matrix, $\mathbf{X}$, compute $\mathbf{X}'\mathbf{X}$ using R's commands. What is your result?
3. Using R's commands, find the inverse of your previous answer. That is, compute $(\mathbf{X}'\mathbf{X})^{-1}$. What is your result?
4. Using the data in Table 10.2, consider the regression model $y_i = \beta_1 x_{1i} + \beta_2 x_{2i} + \beta_3 x_{3i} + u_i$. Using R's matrix algebra commands, compute $\hat{\boldsymbol{\beta}} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$. What are your estimates of the regression coefficients, $\hat{\beta}_1$, $\hat{\beta}_2$, and $\hat{\beta}_3$? How do your results compare if you compute these using the `lm` command from Chap. 6?