Basic R Matrix Operations

Joseph M. Hilbe
Jet Propulsion Laboratory & Arizona State University
Hilbe@asu.edu : 12 Feb, 2011

A matrix is a rectangular array of items, formatted in terms of rows (*i*) and columns (*j*). A matrix is defined by the values of *i* and *j*, which are called the *dimensions* of the matrix. Therefore, a matrix with 3 rows and 4 columns is a (3,4), or 3x4, dimension matrix. When the dimensions of a matrix are identical, for example 2x2 , it is known as a *square* matrix.

R treats matrices in a rather unique manner. An R matrix is a long vector of items wrapped into columns and rows, in that order. Because a matrix is stored as a single vector, its values must be stored in the same mode. For example, the values must all be numeric, all character, or all logical. For statistical purposes, matrices are thought of as numeric. Henceforth we only address numeric matrices and vectors in this monograph.

1: CREATION OF A MATRIX

One may create a matrix from existing rows or columns. Typically they are made by combining columns. For example, let us define 4 vectors of five numbers with the following values:

```
> c1 <- c(3,4,6,8,5)
> c2 <- c(4,8,4,7,1)
> c3 <- c(2,2,5,4,6)
> c4 <- c(4,7,5,2,5)
```

We may combine the four columns into a matrix using the code,

```
> matrix1 <- cbind(c1, c2, c3, c4)
```

which is now a matrix. We show this using the class() function.

```
> class(matrix1)
[1] "matrix"
```

Typing *matrix1* displays the various elements in matrix format. Note that the columns are labeled using the names of the objects that were used for *cbind()*. The rows have no names.

```
> matrix1
     c1 c2 c3 c4
[1,]  3  4  2  4
[2,]  4  8  2  7
[3,]  6  4  5  5
[4,]  8  7  4  2
[5,]  5  1  6  5

> colnames(matrix1)
[1] "c1" "c2" "c3" "c4"

> rownames(matrix1)
NULL
```

The rows can be named, eg. A, B, and so forth, by using the *rownames()* function. Column names can be renamed to other labels using the *colnames()* function.

```
> rownames(matrix1) <- c("A", "B", "C", "D", "E")
> matrix1
  c1 c2 c3 c4
A  3  4  2  4
B  4  8  2  7
C  6  4  5  5
D  8  7  4  2
E  5  1  6  5
```

A seeming matrix can be tested to determine if it is a true matrix by using the function *is.matrix()*. Applied on *matrix1*, we have

```
> is.matrix(matrix1)
[1] TRUE
```

We know that *c1*, *c2*, *c3*, and *c4* are all vectors, which can be tested as

```
> is.vector(c1)
[1] TRUE
```

and are not matrices:

```
> is.matrix(c1)
[1] FALSE
```

It is also the case that *matrix1* is not a vector

```
> is.vector(matrix1)
[1] FALSE
```

matrix1 may be converted into an R matrix that shows rows as [#,] and columns as [,#]

```
> matrix2 <-matrix(matrix1, ncol=4)
> matrix2
```

```
     [,1] [,2] [,3] [,4]
[1,]    3    4    2    4
[2,]    4    8    2    7
[3,]    6    4    5    5
[4,]    8    7    4    2
[5,]    5    1    6    5
```

Matrix operations of addition, subtraction, multiplication, inversion, etc, may now be done using *matrix2*.

One may also create the same matrix as *matrix2* by constructing a long array of numbers, partitioning it into four equal columns, using the code:

```
> matrix3 <- matrix(c(3,4,6,8,5,4,8,4,7,1,2,2,5,4,6,4,7,5,2,5), ncol=4)
> matrix3
     [,1] [,2] [,3] [,4]
[1,]    3    4    2    4
[2,]    4    8    2    7
[3,]    6    4    5    5
[4,]    8    7    4    2
[5,]    5    1    6    5
```

Or you can partition the data into 5 equal rows, as:

```
> matrix4 <- matrix(c(3,4,6,8,5,4,8,4,7,1,2,2,5,4,6,4,7,5,2,5), nrow=5)
> matrix4
     [,1] [,2] [,3] [,4]
[1,]    3    4    2    4
[2,]    4    8    2    7
[3,]    6    4    5    5
[4,]    8    7    4    2
[5,]    5    1    6    5
```

If we had not partitioned the elements into columns or rows, a single array of numbers running down each column, across columns, would be displayed

```
> matrixx <- matrix(c(3,4,6,8,5,4,8,4,7,1,2,2,5,4,6,4,7,5,2,5))
> matrixx
      [,1]
 [1,]    3
 [2,]    4
 [3,]    6
 [4,]    8
 [5,]    5
 [6,]    4
 [7,]    8
 [8,]    4
 [9,]    7
[10,]    1
[11,]    2
[12,]    2
[13,]    5
[14,]    4
[15,]    6
[16,]    4
[17,]    7
[18,]    5
```

```
[19,]     2
[20,]     5
```

matrixx is nevertheless a true matrix, evidenced by

```
> is.matrix(matrixx)
[1] TRUE
```

A small matrix can be defined by combining a vector of numbers and partitioning them by row and column numbers within the same function. For example, *matrixA* is a 2x3 matrix. If we had specified *matrix(c(..), 3,2)* instead, a 3x2 matrix would have been created.

```
> matrixA <-  matrix(c (3 ,4 ,2 ,4 ,8 ,2) ,2 ,3)
> matrixA
     [,1] [,2] [,3]
[1,]    3    2    8
[2,]    4    4    2
```

DIMENSIONS

The dimensions of a matrix may be determined by using the *dim* function. Therefore,

```
> dim(matrix4)
[1] 5 4

> dim(matrixx)
[1] 20  1
```

which tells us that *matrix4* is a 5x4 matrix and *matrixx* is a 20x1 dimension matrix.

It is interesting to note that a simple numeric vector such as *c1* has no dimension in R. However, as we shall later observe, operations on such a vector can result in a matrix.

```
> dim(c1)
NULL
```

FREQUENCY TABLE

We may determine the count of elements in a matrix using the table function. For instance, if we wish to know how many times the value of 1 appears in the matrix, how many 2's, and so forth, we can:

```
> table(matrix4)
matrix4
1 2 3 4 5 6 7 8
1 3 1 5 4 2 2 2
```

There are 5 4's, which is the mode of the elements.

SUM, MEAN AND MEDIAN

If we wish to know the sum, mean, and median of all the elements of the matrix, we can

```
> sum(matrix4)
[1] 92

> mean(matrix4)
[1] 4.6


> median(matrix4)
[1] 4.5
```

However, if there are missing values in the matrix, we need to provide the function with the option: `na.rm=TRUE`.

ROW AND COLUMN SUMS AND MEANS

```
> matrix4
     [,1] [,2] [,3] [,4]
[1,]    3    4    2    4
[2,]    4    8    2    7
[3,]    6    4    5    5
[4,]    8    7    4    2
[5,]    5    1    6    5
```

The *rowSums()* and *rowMeans()* functions are used to obtain the sum and mean of the values for each row in a matrix,

```
> rowSums(matrix4)
[1] 13 21 20 21 17

> rowMeans(matrix4)
[1] 3.25 5.25 5.00 5.25 4.25
```

Checking the sum of each row in *matrix4* by sight, it is clear that the vector of sum values are correct. Recall that each vector can be named, for example,

```
> rsum <- rowSums(matrix4)

> rsum
[1] 13 21 20 21 17

> is.vector(rsum)
[1] TRUE
```

It is also simple to verify that the row means are calculated correctly. Using the second row we have

```
> (4 + 8 + 2 + 7)/4
[1] 5.25
```

Which is identical to the second element in the vector of row means above.
Column sums and means have the same logic as for rows. Using the *colSums()* and *colMeans()* functions we have,

```
> colSums(matrix4)
[1] 26 24 19 23

> colMeans(matrix4)
[1] 5.2 4.8 3.8 4.6
```

2: SUBSETS OF MATRICES

We may select a particular element from *matrix4*, defined by its row and column, using the code:

matrix4[row, column]

For instance, if we want to display the element (4,2) we could:

```
> matrix4[4,2]
[1] 7
```

Referring to the matrix output for *matrix4* we find that the (4,2) element is indeed 7.
We may also select a block subset of elements from *matrix4*. For instance, suppose that we wish to create a new matrix, *matrix5*, that consists or the first two columns of *matrix4*. We select rows 1 through 5 and columns 1 through 2, shown as:

```
> matrix5 <- matrix4[1:5,1:2]
> matrix5
     [,1] [,2]
[1,]    3    4
[2,]    4    8
[3,]    6    4
[4,]    8    7
[5,]    5    1

> dim(matrix5)
[1] 5 2
```

I provided a test by issuing the *dim()* function for *matrix5*, demonstrating the it is in fact a 5x2 matrix.
A shortcut can be given when selecting an entire row of column. For *matrix5* we selected all of the rows for two columns. We therefore do not have to specifically show the row numbers. For example, we could have created *matrix5* using the code:

```
> matrix5 <- matrix4[,1:2]
> matrix5
     [,1] [,2]
[1,]    3    4
[2,]    4    8
[3,]    6    4
[4,]    8    7
[5,]    5    1
```

Let us create *matrix6* as the two rightmost columns, 3 and 4.

```
> matrix6 <- matrix4[1:5,3:4]
> matrix6
     [,1] [,2]
[1,]    2    4
[2,]    2    7
[3,]    5    5
[4,]    4    2
[5,]    6    5
```

or,

```
> matrix6 <- matrix4[,3:4]
> matrix6
     [,1] [,2]
[1,]    2    4
[2,]    2    7
[3,]    5    5
[4,]    4    2
[5,]    6    5
```

Finally, we can abstract any block from a matrix using the correct numeric designations. For example, suppose that we want to display the inner elements of *matrix4*. Viewing *matrix4* again,

```
> matrix4
     [,1] [,2] [,3] [,4]
[1,]    3    4    2    4
[2,]    4    8    2    7
[3,]    6    4    5    5
[4,]    8    7    4    2
[5,]    5    1    6    5
```

We want to show the block that consists of the ***bold-italic*** numbers in the middle of *matrix4* above. We select rows 2 through 4 and columns 2 and 3. A new matrix can be created with 3x2 dimensions.

```
> matrix4a <- matrix4[2:4, 2:3]
> matrix4a
     [,1] [,2]
[1,]    8    2
[2,]    4    5
[3,]    7    4
```

3: APPENDING ROWS AND CONCATENATION OF COLUMNS

Appending rows of two or more different but structurally similar matrices together, and concatenating columns of two of more different, but structurally similar matrices together is a common matrix operation.

To show how to append rows of two matrices, we use *matrixA* which we constructed earlier, and will create another matrix, called *matrixB*, which shall be appended to *matrixA*.

```
> matrixA
     [,1] [,2] [,3]
[1,]    3    2    8
[2,]    4    4    2

> matrixB <-  matrix(c (6 ,4 ,5 ,8 ,7 ,4) ,2 ,3)
> matrixB
     [,1] [,2] [,3]
[1,]    6    5    7
[2,]    4    8    4

> matrixC <-  rbind(matrixA, matrixB)
> matrixC
     [,1] [,2] [,3]
[1,]    3    2    8
[2,]    4    4    2
[3,]    6    5    7
[4,]    4    8    4
```

We may invert the order so that the elements of *matrixB* are above those of *matrixA*.

```
> matrixCi <-  rbind(matrixB, matrixA)
> matrixCi
     [,1] [,2] [,3]
[1,]    6    5    7
[2,]    4    8    4
[3,]    3    2    8
[4,]    4    4    2
```

We only needed to reverse the order in which matrices declared. If there are a number of matrices which we intend to append, their order corresponds to the order in which they are declared in the *rbind()* function.

The two matrices may also be concatenated, that is, their columns may be joined, creating a new matrix.

```
> matrixD <-  cbind(matrixA, matrixB)
> matrixD
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    3    2    8    6    5    7
[2,]    4    4    2    4    8    4
```

In a similar manner to appending rows, concatenation is done by the order in which columns are given to the `cbind()` function.

```
> matrixDi <-  cbind(matrixB, matrixA)
> matrixDi
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    6    5    7    3    2    8
[2,]    4    8    4    4    4    2
```

It should be apparent from what we have said that rows and columns may be duplicated using the *rbind()* and *cbind()* functions. To duplicate the rows of *matrixA* we simply specify *matrixA* twice. Any number of duplications can be made.

```
> rbind(matrixA, matrixA)
     [,1] [,2] [,3]
[1,]    3    2    8
[2,]    4    4    2
[3,]    3    2    8
[4,]    4    4    2
```

The *cbind()* function can also be used to insert a column between others of the same length. First we create a matrix called *matrixE*, with all 1's in the first column and the numbers 1 through 7 in the second.

```
> matrixE <- cbind(1, 1:7)
> matrixE
     [,1] [,2]
[1,]    1    1
[2,]    1    2
[3,]    1    3
[4,]    1    4
[5,]    1    5
[6,]    1    6
[7,]    1    7
```

Next we create a vector of even numbers from 2 through 14, calling it *new*. We now insert *new* into the matrix, first as the third column, and then rearranging it to be placed in the middle between the previous two columns.

```
> new <- c(2,4,6,8,10,12,14)
> matrixF <- cbind(matrixE, new)[, c(1, 3, 2)]


> matrixF
        new
[1,] 1   2 1
[2,] 1   4 2
[3,] 1   6 3
[4,] 1   8 4
[5,] 1  10 5
[6,] 1  12 6
[7,] 1  14 7
```

4: MATRIX ADDITION

Matrix addition is straight forward, adding element by element from two similar matrices; ie from matrices with identical dimensions.

Suppose *matrix5* and *matrix6*, which were created as subsets of *matrix4*.

```
> matrix5
     [,1] [,2]
[1,]    3    4
[2,]    4    8
[3,]    6    4
[4,]    8    7
[5,]    5    1
```

```
> matrix6
     [,1] [,2]
[1,]    2    4
[2,]    2    7
[3,]    5    5
[4,]    4    2
[5,]    6    5
```

Since the above matrices are true matrices, we simply add the elements of each matrix as:

```
> matrix5 + matrix6
     [,1] [,2]
[1,]    5    8
[2,]    6   15
[3,]   11    9
[4,]   12    9
[5,]   11    6
```

To check, we may add elements (1,1) from each matrix

```
> matrix5[1,1] + matrix6[1,1]
[1] 5
```

and elements (2,2), which are [8,7],

```
> matrix5[2,2] + matrix6[2,2]
[1] 15
```

It can be observed that we may add elements – any elements – between two or more matrices, if done individually, and we can add whole matrices elementwise in a straightforward manner.

5: MATRIX SUBTRACTION

We may subtract matrices in exactly the same manner as we added them. Therefore,

```
> matrix5 - matrix6
     [,1] [,2]
[1,]    1    0
[2,]    2    1
[3,]    1   -1
[4,]    4    5
[5,]   -1   -4
```

Again, subtracting the (1,1) elements of *matrix6* from *matrix5*, we have

```
> matrix5[1,1] - matrix6[1,1]
[1] 1
```

and subtracting elements (4,2), which is 7-2 = 5.

```
> matrix5[4,2] - matrix6[4,2]
[1] 5
```

5 in indeed the resultant value of the (4,2) element subtraction.

6: MATRIX TRANSPOSITION

Transposition is an important matrix operation. Essentially, we transpose a matrix if we switch rows and columns. For example, I show both *matrix5* and its transpose.

```
> matrix5
     [,1] [,2]
[1,]    3    4
[2,]    4    8
[3,]    6    4
[4,]    8    7
[5,]    5    1

> t(matrix5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    3    4    6    8    5
[2,]    4    8    4    7    1
```

The value 5 in *matrix5* is element (5,1). Its transposed position is (1,5).
What happens if we transpose both *matrix5* and *matrix6* and add them?

```
> t(matrix5) + t(matrix6)
     [,1] [,2] [,3] [,4] [,5]
[1,]    5    6   11   12   11
[2,]    8   15    9    9    6
```

It is indeed the transpose of *matrix5 + matrix6*. Likewise, the subtraction of the transpose of *matrix6* from the transpose of *matrix5* is the same as the transpose of *matrix5 – matrix6*.

```
> t(matrix5) - t(matrix6)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    1    4   -1
[2,]    0    1   -1    5   -4
```

The transpose of a matrix, e.g., X, is traditionally symbolized as X', or X-prime. We shall discuss transposed matrices later in section 10.

7: DIAGONAL, IDENTITY, AND SYMMETRICAL MATRICES

DIAGONAL MATRIX

First, we shall construct two matrices taken from *matrix4*, *matrix7* consists of the upper-left 2x2 block, and *matrix8* is the upper-left 3x3 block. Each is a square matrix.

```
> matrix7 <- matrix4[1:2,1:2]
> matrix7
     [,1] [,2]
[1,]    3    4
[2,]    4    8
>
> matrix8 <- matrix4[1:3,1:3]
> matrix8
```

```
     [,1] [,2] [,3]
[1,]    3    4    2
[2,]    4    8    2
[3,]    6    4    5
```

The diagonal of *matrix8* consists of the vector of values running from the upper-left to lower-right; i.e., 3, 8, 5

```
> diag3 <- diag(matrix8)
> diag3
[1] 3 8 5
```

It is clear that the diagonal of *matrix7* is the vector 3, 8, and 5.

TRACE

The sum of the diagonal elements of a square matrix is called the *trace*. The trace of *matrix8* is 3+8+5 = 16. In R the trace can be calculated using the code,

```
> sum(diag(matrix8))
[1] 16
```

IDENTITY

An identity matrix is a matrix with 1's on the diagonal and 0 otherwise. It is commonly used in statistical operations. A matrix multiplied by an identity matrix remains unchanged.

```
> I <- diag(c(1,1,1))
> I
[,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

SYMMETRY

The transpose of a symmetric matrix is the same as the non-transposed matrix. Symmetric matrices are square. We first create a 3x3 square matrix called *Sy*.

```
> Sy <- matrix(c(2,1,3,1,2,1,3,1,0),3,3)
> Sy
     [,1] [,2] [,3]
[1,]    2    1    3
[2,]    1    2    1
[3,]    3    1    0
```

We transpose *Sy*, and find that the elements are identical in the two matrices. The matrix *Sy* is therefore symmetric.

```
> t(Sy)
     [,1] [,2] [,3]
[1,]    2    1    3
[2,]    1    2    1
[3,]    3    1    0
```

8: MATRIX DETERMINANT

Determinants require square matrices. Let us create a simple 2x2 matrix appearing as 1, 0, 1, 1.

```
> matrix9 <- matrix(c(0,1,1,1), nrow=2)
> matrix9
     [,1] [,2]
[1,]    0    1
[2,]    1    1
```

For such a matrix, we multiply the (1,1) and (2,2) elements and subtract the product of (2,1) and (1,2). This gives us 0 – 1 = -1.

```
> det(matrix9)
[1] -1
```

The determinant of *matrix7*, shown below, is calculated as:

```
> matrix7
     [,1] [,2]
[1,]    3    4
[2,]    4    8

> det(matrix7)
[1] 8

> (3*8) - (4*4)
[1] 8
```

3x3 matrices are more difficult to solve. For example, we shall work with *matrix8*.

```
     [,1] [,2] [,3]
[1,]    3    4    2
[2,]    4    8    2
[3,]    6    4    5
```

There are a number of methods by which third-order determinants may be solved. The order is based on the number of rows or columns in the matrix.

One method is based on the following schemata of the matrix

```
     [,1] ,2] [,3]
[1,]  a11 a12  a13
[2,]  a21 a22  a23
[3,]  a31 a32  a33
```

The determinant of *matrix7*, symbolized as | *matrix7* |, is given from the following:

$$|D| = a11(+1)\begin{vmatrix} a22 & a23 \\ a32 & a33 \end{vmatrix} + a12\,(-1)\begin{vmatrix} a21 & a23 \\ a31 & a33 \end{vmatrix} + a13(+1)\begin{vmatrix} a21 & a22 \\ a31 & a32 \end{vmatrix}$$

= a11*a22*a33 – a11*a23*a32 – a12*a21*a33 + a12*a23*a31 + a13*a21*a32 – a13*a22*a31

```
> 3*8*5 - 3*2*4 - 4*4*5 + 4*2*6 + 2*4*4 - 2*8*6
[1] 0

> det(matrix8)
[1] 5.921479e-15
```

Note the rounding error in the calculated value. For most all purposes this value is considered to be equal to 0.

We may also calculate the determinant as

```
> 3*(40-8) - 4*(20-12) +  2*(16-48)
[1] 0
```

In any case, it is clear that the determinant is 0. Using the same method it is simple to calculate the third order determinant of matrix10.

```
> matrix10 <- matrix(c(1,1,0,1,0,1,0,1,1), nrow=3)
> matrix10
     [,1] [,2] [,3]
[1,]    1    1    0
[2,]    1    0    1
[3,]    0    1    1

> det(matrix10)
[1] -2
```

Determinants have some interesting properties. Another way to designate a determinant is *det(A)* where A is a square matrix. If A and B are square matrices, and their product is C = AB, then *det(C) = det(AB)* , as well as *det(A)det(B).* There are other properties which we mention later in this monograph.

9: MATRIX MULTIPLICATION

Except for multiplication of the elements of a matrix by a scalar, matrix multiplication is more difficult than the previous operations. The logic is not difficult. It is just that care must be taken when dealing with the various columns of data.

MULTIPLICATION BY A SCALAR

Suppose that we wish to multiply *matrix7* by 4. We do this by multiplying each element of the matrix by 4. The same holds for any matrix when multiplied by a scalar.

```
> matrix7
     [,1] [,2]
[1,]    3    4
[2,]    4    8

> 4*matrix7
     [,1] [,2]
[1,]   12   16
[2,]   16   32
```

or,

```
> Sc <- 4
> Sc*matrix7
     [,1] [,2]
[1,]   12   16
[2,]   16   32
```

TWO MATRICES

Matrices must be *conformable* in order to be multiplied. The number of columns in the first matrix must equal the number of rows of the second, e.g. (1x4) X (4x2). The inner terms of the two matrices must be equal. The dimensions of the product equals the outer terms of the two matrices, e.g. (1x4) X (4x2) = (1x2).

To give a simple example, we multiply a row matrix by a column matrix. Suppose we have a 1x4 row matrix defined as

```
> matM <- matrix(c(3,4,6,8), nrow=1)
> matM
     [,1] [,2] [,3] [,4]
[1,]    3    4    6    8
```

and a 4x1 column matrix defined as

```
> matN <- matrix(c(2,1,4,2), ncol=1)
> matN
     [,1]
[1,]    2
[2,]    1
[3,]    4
[4,]    2
```

Since the number of columns of *matM* equals the number of rows in *matN*, we may multiply the matrices using the *%*%* operator as:

```
> matO <- matM %*% matN
> matO
     [,1]
[1,]   50
```

The product is a single value. That is, 1x4 X 4x1 results in a 1x1 matrix.

The product is determined by summing the products of the corresponding elements in each matrix. We only need perform this operation once when multiplying a row matrix by a column matrix. Here we have (3x2)+(4x1)+6x4)+(8x2) = 50.

Suppose that we wish to multiply *matN* by *matP*, which we define as:

```
> matP
     [,1] [,2] [,3] [,4]
[1,]    3    4    2    4
[2,]    4    8    2    7
[3,]    6    4    5    5
[4,]    8    7    4    2
```

The matrices are conformable. The product is the result of multiplying *matM* by each column of *matP*. Since we are multiplying a 1x4 matrix with a 4x4, we expect to have a 1x4 matrix as the product.

```
> matQ <- matM %*% matP
> matQ
     [,1] [,2] [,3] [,4]
[1,]  125  124   76   86
```

The matrix multiplication of two 2x2 matrices is based on the same logic as described above. However, each row of the first matrix is multiplied by each column of the second, resulting in a 2x2 product. We shall demonstrate by taking two 2x2 matrices from *matrix4*. *matrix7* has previously been constructed from matrix4 as:

```
> matrix7
     [,1] [,2]
[1,]    3    4
[2,]    4    8
```

We now create *matrix11*, which is the upper-right 2x2 block in *matrix4*.

```
> matrix11 <- matrix4[1:2,3:4]
> matrix11
     [,1] [,2]
[1,]    2    4
[2,]    2    7
```

Multiplication of *matrix7* and *matrix11* appears as,

```
> matrix7 %*% matrix11
     [,1] [,2]
[1,]   14   40
[2,]   24   72
```

with (3x2)+(4x2) = 14 as the upper-left element of the product, (3x4)+(4x7) = 40 as the upper-right element, (4x2)+(8x2) = 24 as the lower-left element, and (4x4)+(8x7)=72 as the lower-right element.

The *crossprod()* can also be used to multiply matrices. Many statisticians prefer the *crossprod()* function to the %*% operator in that there are built-in efficiencies to the function.

```
> crossprod(matrix7, matrix11)
     [,1] [,2]
[1,]   14   40
[2,]   24   72
```

Next, consider multiplying *matrixA* by *matrixB*. Both are 2x3 matrices. We may multiply them if we transpose *matrixB* to become a 3x2 matrix, which we shall call *matrixF*.

```
> matrixA
     [,1] [,2] [,3]
[1,]    3    2    8
[2,]    4    4    2
```

```
> matrixB
     [,1] [,2] [,3]
[1,]    6    5    7
[2,]    4    8    4

> matrixF <- t(matrixB)
> matrixF
     [,1] [,2]
[1,]    6    4
[2,]    5    8
[3,]    7    4

> matrixA %*% matrixF
     [,1] [,2]
[1,]   84   60
[2,]   58   56
```

or

```
> matrixA %*% t(matrixB)
     [,1] [,2]
[1,]   84   60
[2,]   58   56
```

We may also use R's *tcrossprod()*, which internally transposes the matrix.

```
> tcrossprod(matrixA, matrixB)
     [,1] [,2]
[1,]   84   60
[2,]   58   56
```

Finally, consider the 2x5 *matR*, defined as

```
> matR <- t(matrix5)
> matR
     [,1] [,2] [,3] [,4] [,5]
[1,]    3    4    6    8    5
[2,]    4    8    4    7    1
```

We may multiply it with a 5x*q* matrix, where *q* represents any value. Suppose the 5x4 *matrix4*.

```
> matrix4
     [,1] [,2] [,3] [,4]
[1,]    3    4    2    4
[2,]    4    8    2    7
[3,]    6    4    5    5
[4,]    8    7    4    2
[5,]    5    1    6    5
```

Here we have a relationship of (2x5) X (5x4), and should expect a 2x4 product. Indeed, we get

```
> matR %*% matrix4
     [,1] [,2] [,3] [,4]
[1,]  150  129  106  111
[2,]  129  146   78  111
```

DIRECT PRODUCT

It should be mentioned that at times it is important to multiply the elements of two similar matrices, e.g. *matrixA* and *matrixB*. Both are 2x3 matrices. This type of result is called a *direct product*, and is accomplished by using the standard `*' operator. Each of the similar elements of the two matrices is multiplied with one another. For example, directly multiplying *matrixA* and *matrixB*, element 1,1 of the two matrices, 3,6, produces 18. Multiplying elements 1,2, with values of 2,5 results in 10, and so forth.

```
> matrixA * matrixB
     [,1] [,2] [,3]
[1,]   18   10   56
[2,]   16   32    8
```

KRONECKER PRODUCT

The *Kronecker product* of two matrices is a bit more complicated, with each element of the first matrix being scalar multiplied with the whole of the second matrix, arrayed as the same structure as the first.

Suppose we wish to find the Kronecker product of *matrix7* and *matrix11*.

```
> matrix7
     [,1] [,2]
[1,]    3    4
[2,]    4    8

> matrix11
     [,1] [,2]
[1,]    2    4
[2,]    2    7
```

Element 1,1 of *matrix7* is multiplied as a scalar with *matrix11*, producing a 2x2 matrix

```
> 3*matrix11
     [,1] [,2]
[1,]    6   12
[2,]    6   21

Or

> matrix7[1,1]*matrix11
     [,1] [,2]
[1,]    6   12
[2,]    6   21
```

This 2x2 matrix is element 1,1 of the resultant Kronecker product. To obtain element 2,1 of the Kronecker product we multiply element 2,1 of *matrix7* (4) with *matrix11*, producing,

```
> matrix7[2,1]*matrix11
     [,1] [,2]
[1,]    8   16
[2,]    8   28
```

The other two elements of the Kronecker product of *matrix7* and *matrix11* are found in the same manner. The Kronecker product can be found directly in R by,

```
> kronecker(matrix7, matrix11)
     [,1] [,2] [,3] [,4]
[1,]    6   12    8   16
[2,]    6   21    8   28
[3,]    8   16   16   32
[4,]    8   28   16   56
```

Note that the upper left 2x2 block and lower left 2x2 block of the above result is the same as the two scalar products we calculated above. The product each of the 4 elements of *matrix7* by *matrix11* results in the combined 4-block matrix calculated above.

```
    [,1] [,2]  [,3] [,4]
[1,]  6   12 |   8   16
[2,]  6   21 |   8   28
      -----------------
[3,]  8   16 |  16   32
[4,]  8   28 |  16   56
```

The matrix terms used to calculate a Kronecker product do not have to be similar since the resultant product takes the format of the first matrix.

VECTOR MULTIPLICATION: SQUARING

In R, vectors are technically dimensionless; they have only length. Recalling the vector, *c1*, that we created in Section 1

```
> c1
[1] 3 4 6 8 5
```

It’s dimension can be checked by the *dim()* function

```
> dim(c1)
NULL
```

However, we can use the function *as.matrix()* to format *c1* as a matrix – a column matrix.

```
> dim(as.matrix(c1))
[1] 5 1
```

Interestingly, in R the transpose of a vector does have dimension – it is 2-dimensional,

```
> dim(t(c1))
[1] 1 5
```

which indicates that the transpose of a numeric non-dimensional vector is a 2-dimensional matrix. Of course, this is how R handles it – it is not a feature of matrices themselves.
We observe this relationship using the *class()* function.

```
> class(c1)
"numeric"
```

```
> class(t(c1))
“matrix”
```

A vector of squares is produced if the same vector is directly multiplied by itself. For example,

```
> d1 <- c1 * t(c1)
> d1
     [,1] [,2] [,3] [,4] [,5]
[1,]    9   16   36   64   25

> dim(d1)
[1] 1 5

> d2 <- c1 * c1
> d2
[1]  9 16 36 64 25

> dim(d2)
NULL

> class(d2)
[1] "matrix"
```

If we matrix multiply *c1* by the transpose of *c1*, we are multiplying a 5x1 by a 1x5 matrix, which results in a 5x5 matrix.

```
> d3 <- c1 %*% t(c1)
> d3
     [,1] [,2] [,3] [,4] [,5]
[1,]    9   12   18   24   15
[2,]   12   16   24   32   20
[3,]   18   24   36   48   30
[4,]   24   32   48   64   40
[5,]   15   20   30   40   25

> dim(d3)
[1] 5 5
```

We can obtain a vector of squares using matrix multiplication by using the diagonal, *diag()*, function

```
diag(c1 %*% t(c1))
[1]  9 16 36 64 25
```

The sum of squares can be obtained by reversing the order of matrices so that we are multiplying a 1x5 by a 5x1 matrix. The sum of squares, 9+16+36+64+25 = 150.

```
> t(c1) %*% c1
     [,1]
[1,]  150
```

A vector of the products of corresponding terms in two different vectors can be obtained as,

```
> c1*c2
[1] 12 32 24 56  5
```

or

```
> c1*t(c2)
     [,1] [,2] [,3] [,4] [,5]
[1,]   12   32   24   56    5
```

10: MATRIX INVERSION

The *solve()* function is used to invert a square matrix. Inversion is in a sense the same as matrix division; there is no other type of matrix division. The traditional manner of indicating the inverse of a square matrix A is A^{-1}. Multiplying A and A^{-1}, i.e., AA^{-1}, results in an identity matrix, I. It should again be noted that elements of the inverted matrix are many times displayed in scientific notation due to rounding error in the calculations. Use of the *zapsmall()* function sometimes helps produce more readable values.

Recall that $[X'X]^{-1}$ is central to the estimation of least squares regression parameters, with *X* symbolizing the matrix of model data. If there is to be an intercept in the regression, the first column of *X* consists of all 1's. *X'* is the transpose of *X*.

Consider *matrix7*. We invert it, then multiply it by A, which produces an identity matrix. .

```
> matrix7
     [,1] [,2]
[1,]    3    4
[2,]    4    8
```

We may easily invert this 2x2 matrix by hand by multiplying each element of the matrix by the inverse of the determinant. If we symbolize *matrix7* as A, we may invert it by:

$$A^{-1} = \frac{1}{a_{11}a_{22}-a_{12}a_{21}}\begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix}$$

$$= \frac{1}{(3)(8)-(4)(4)}\begin{bmatrix} 8 & 4 \\ 4 & 3 \end{bmatrix}$$

$$= \frac{1}{8}\begin{bmatrix} 8 & 4 \\ 4 & 3 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & .5 \\ .5 & .375 \end{bmatrix}$$

Employing the *solve()* function on *matrix7* we have,

```
> solve(matrix7)
     [,1]   [,2]
[1,]  1.0 -0.500
[2,] -0.5  0.375
```

Checking to determine if we calculated correctly, multiplying *matrix7* by its inverse should result in an identity matrix.

```
> matrix7 %*% solve(matrix7)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

We observe that an identity matrix results.

Inverting 3x3 matrices is more difficult; inverting higher dimension matrices is substantially more tedious. It is best to let the computer handle such inversion tasks. Keep in mind though to assure that the elements of the matrix to be inverted are stored as doubles so that rounding error is minimized.

It should also be noted that the following relationships exist with respect to inversion.

$$A = (A^{-1})^{-1}$$
$$(A^{-1})' = (A')^{-1}$$
$$(AB)^{-1} = B^{-1}A^{-1}$$

The inverse of a symmetric matrix is also symmetric. $S^{-1} = (S^{-1})'$

ORTHOGONAL MATRICES

Orthogonal matrices are unique types of square matrices, where its transpose is also its inverse. If O is an orthogonal matrix, O'O = I = OO'. Statisticians determine if a square matrix is orthogonal by checking if O'O = OO' = I. Orthogonal matrices are important when calculating eigenvalues and eigenvectors, which we do not discuss in this monograph.

USING THE INVERSE OF MATRICES FOR SOLUTION OF EQUATIONS

Matrix inversion is an essential operation in regression analysis, and is the method used by many mathematicians to solve systems of equations. For example, consider the following system of two equations having two unknowns.

$$x + 3y = 18$$
$$2x + 3y = 24$$

Algebraically we would multiply each side of the first equation by 2, obtaining

$$2x + 6y = 36$$

Then subtract the second equation, resulting in

$$3y = 12$$
$$y = 4$$

Then substitute *y*=4 into either equation and solve for x. Thus

$$x + (3)(4) = 18$$
$$x = 6$$

An easier method is to use matrices.

$$\begin{bmatrix}1 & 3\\ 2 & 3\end{bmatrix}\begin{bmatrix}x\\ y\end{bmatrix} = \begin{bmatrix}18\\ 24\end{bmatrix}$$

Multiply each side by the inverse of the square matrix,

$$\begin{bmatrix}1 & 3\\ 2 & 3\end{bmatrix}^{-1}\begin{bmatrix}1 & 3\\ 2 & 3\end{bmatrix}\begin{bmatrix}x\\ y\end{bmatrix} = \begin{bmatrix}1 & 3\\ 2 & 3\end{bmatrix}^{-1}\begin{bmatrix}18\\ 24\end{bmatrix}$$

$$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}x\\ y\end{bmatrix} = \begin{bmatrix}-1 & 1\\ .667 & -.333\end{bmatrix}\begin{bmatrix}18\\ 24\end{bmatrix}$$

```
> matrixG <- matrix(c(1,2,3,3), ncol=2)
> matrixG
     [,1] [,2]
[1,]    1    3
[2,]    2    3

> matrixI <- solve(matrixG)
> matrixI
           [,1]       [,2]
[1,] -1.0000000  1.0000000
[2,]  0.6666667 -0.3333333
```

Solving matrix [*x*,*y*] by multiplying the inverse the square matrix of coefficients by the right hand side column matrix, we have

```
> matrixF <- matrix(c(18,24), ncol=1)
> matrixF
     [,1]
[1,]   18
[2,]   24

> matrixI %*% matrixH
     [,1]
[1,]    6
[2,]    4
```

which is

$$\begin{bmatrix}x\\ y\end{bmatrix} = \begin{bmatrix}6\\ 4\end{bmatrix}$$

The above matrix solution can be streamlined to reduce the steps, but more importantly the same format maintains for higher systems of equations. Whereas it would take quite a while to solve a 5-equation system, solving the matrix formulation is as easy as the 2x2 solution.

For example, consider the 5 unknown system of equations:

$$\begin{array}{rcrcrcrcrcr}
a & + & 2b & - & 1.5c & + & 1.25d & - & .5e & = & 2\\
-.25a & + & 1.2b & + & .6c & - & .33d & + & .75e & = & 1.4\\
.4a & + & .7b & + & 3c & + & d & - & 2e & = & 3\\
a & - & 1.25b & + & 3c & + & 1.2d & - & 1.5e & = & 1.75\\
2a & + & 3b & - & c & + & 2d & - & 1.5e & = & 1.5
\end{array}$$

After setting up the matrix of *X* and *y* values, with *y* being the response or dependent variable, we may solve for the 5 parameters, *a*, *b*, *c*, *d* and *e* by *solve(X) %*% y*.

```
> X <- matrix(c(1, 2, -1.5, 1.25, -.5, -.25, 1.2, .6, -.33, .75, .4, .7, 3,
1, -2, 1,-1.25, 3, 1.2, -1.5, 2, 3, -1, 2, -1.5), ncol=5)

> y <- matrix(c(2, 1.4, 3, 1.75, 1.5), ncol=1)
> solve(X) %*% y
           [,1]
[1,]  1.7746925
[2,]  3.2314060
[3,] -1.6239844
[4,]  2.6961499
[5,] -0.5066986
```

11: LINEAR REGRESSION

Recall that we earlier mentioned that $[X'X]^{-1}$ is central to regression. In fact, linear regression models are solved by matrices as:

$$\beta = [X'X]^{-1} X'y$$

where *X* is the matrix of data values in the model, *y* is the response or dependent variable, and β is the vector of model coefficients, or parameters, to be estimated. This is based on the well known relationship

$$\hat{y} = \beta_0 + \beta_0 X_1 + \ldots + \beta_n X_n$$

The product of *X* and its transpose is a square matrix, which inverted and multiplied by the product of *X* and the dependent variable, *y*, produces the parameter estimates, β.

I show how this logic can be implemented in R to solve for linear regression coefficients, which are given as beta. First we must create some data. Suppose we create a 14 row, 2 column matrix as

```
> X <- matrix(c(1,4,6,0,5,4,1,4,7,1,2,2,1,4,6,0,7,5,1,5,1,1,2,5,0,3,4,4),
    ncol=2)
```

If the regression is to have an intercept, we need to place a column of 1's as the first column, making a 14x3 matrix. We can so this in several ways. First, we can use the matrix function we employed earlier in this monograph,

```
> int <- matrix(rep(1,14), ncol=1)
```

However, it is better to employ a more generic value than 14 – a function that picks up the number of rows in the data. The *dim()* does this. But to use it, we must use the repetition, or *rep()*, function. I call the vector of 14 1's *int*, and bind it to the columns of X as the first column. We then have an enhanced the 14.x 3 data matrix, X.

```
> int <- rep(1,dim(X)[1])
> X <- cbind(int, X)
```

Next create a column matrix consisting of the values of the response or dependent variable, y.

```
> y <- matrix(c(10,9,7,4,8,12,11,7,3,5,3,12,9,10), ncol=1)
```

Finally, we construct the various terms appropriate for a linear regression, *X'X*, *XX*$^{-1}$, and *X'y*.

```
> XX <- t(X) %*% X
> XX1 <- solve(XX)
> Xy <- t(X) %*% y
```

A vector or parameter estimates can be obtained as [*X'X*] $^{-1}$ *X'y*. Note that I defined *XX1* as the inverse of *X'X*.

```
> beta <- XX1 %*% Xy
> beta
        [,1]
int  7.7903040
    -0.1430014
     0.1577683
```

Combining the above commands and functions and running from the R script editor, we have

```
> X <- matrix(c(1,4,6,0,5,4,1,4,7,1,2,2,1,4,6,0,7,5,1,5,1,1,2,5,0,3,4,4),
+     ncol=2)
> int <- rep(1,dim(X)[1])
> X <- cbind(int, X)
> y <- matrix(c(10,9,7,4,8,12,11,7,3,5,3,12,9,10), ncol=1)
> XX <- t(X) %*% X
> XX1 <- solve(XX)
> Xy <- t(X) %*% y
> beta <- XX1 %*% Xy
> beta
           [,1]
int  7.7903040
    -0.1430014
     0.1577683
```

The final terms of the above algorithm can be combined so that beta (β) is solved by one line of code. I have formatted the output to 4 places. The entire code group is then,

```
> X <- matrix(c(1,4,6,0,5,4,1,4,7,1,2,2,1,4,6,0,7,5,1,5,1,1,2,5,0,3,4,4),
    ncol=2)
> int <- rep(1,dim(X)[1])
> X <- cbind(int, X)
> y <- matrix(c(10,9,7,4,8,12,11,7,3,5,3,12,9,10), ncol=1)
> beta <- solve(t(X) %*% X) %*% (t(X) %*% y)
> round(beta, 4)
        [,1]
int  7.7903
    -0.1430
     0.1578
```

with the intercept value of 7.79, and the coefficients for the two columns as -0.143 and 0.158 respectively. We can use the *lm()* function in R to solve a linear regression. But first *X* must be partitioned into two separate variables, *x1* and *x2*.

```
> x1 <- X[,2]      # create x1 as the second column of X
> x2 <- X[,3]      # create x2 as the third column of X
> lm(y ~ x1+x2)    # linear regression

Call:
lm(formula = y ~ x1 + x2)

Coefficients:
(Intercept)           x1           x2
     7.7903      -0.1430       0.1578
```

In this monograph we have covered most of the matrix operations related basic statistical analysis using R. There are many more complex matrix operations, to be sure. The operations described here, however, should serve as a solid foundation upon which other more complicated matrix functions and operations can be handled.