Chapter 4

Matrices

Most all statistical techniques for multi-variate data analysis require some matrix algebra. This chapter covers R's matrix algebra features. Before we get to matrices, I will show you how to save and load your objects and code.

4.1 Maintaining Code & Objects

In the previous chapter we began using a text editor to keep track of our code. Whenever you use R for data analysis you will want to save your code in order to replicate your analysis. Open Notepad and write something like the following:

```
############################
# DATE: 12/29/2012         #
# AUTHOR: Peter Haschke    #
# INFO: test script for R  #
############################

# delete all objects from active memory

rm(list = ls())

# Create two vectors

X<-c(1,2,3,4,5)
Y<-c(5,4,3,2,1)

# Do some maths

Z <- X - Y
MEAN.Z <- mean(Z)
MIN.Z <- min(Z)     # This finds the smallest element
MAX.Z <- max(Z)     # This finds the largest element

# This concludes my test script for R
```

4.1.1 Scripting

All code should be saved with the `.r` or `.R` extension. So click 'File' and then 'Save as...' and save the file onto your Z-Drive as `"Z:\test.R"`. Now open the **R** console and type the following and press enter:

```
> source("z:/test.R")
>
```

If all went well, **R** just executed your entire script. All the code you wrote has been evaluated and you can access all the objects you created. It's magic, I know.

```
> ls() # verify if all the objects we created are there
[1] "MAX.Z"   "MEAN.Z" "MIN.Z"   "X"       "Y"       "Z"

> Z
[1] -4 -2  0  2  4
```

Using the `source()` function like this can be tedious especially while you are working on a program or some code. In those cases it may be more efficient to use the console interactively, by copying or pasting from your editor (or by using an editor that can directly communicate with **R**). Writing properly source-able files, however, is really important for sharing code, and writing programs that can be replicated on any computer and by anyone. I expect all problem sets to be sent to me in such a self-contained format.

4.1.2 Saving and Loading Objects

Whenever you are writing programs that take a while to execute (e.g. you are inverting some crazy matrix, etc), saving your code as a source-able script only is not a good idea. Luckily, you can save and load output or results. The easiest way to do this is via the `save()` and `load()` functions. Objects are saved in the `.Rdata` format via the `file` argument.

```
> save(X, Y, Z, MEAN.Z, file = "z:/results.Rdata")
>
```

Let's see if it works. Of course it does.

```
> rm(list = ls()) # remove all objects in active memory
> ls() # check if they are really gone
character(0)

>  # Above: R telling you it found no named objects
> load("z:/results.Rdata")
> ls()
[1] "MEAN.Z" "X" "Y" "Z"
```

4.2 Creating Matrices

To create matrices we will use the `matrix()` function. The `matrix()` function takes the following arguments:

- `data` an R object (this could be a vector)
- `nrow` the desired number of rows
- `ncol` the desired number of columns
- `byrow` a logical statement to populate the matrix by either row or by column

Example 1:
A 3 by 3 matrix filled with 1s

```
> Matrix1 <- matrix(data = 1, nrow = 3, ncol = 3)
> Matrix1
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
[3,]    1    1    1
```

Example 2:
As always we can drop the argument names (as long as you remember that the first argument asks for the data, the second for the rows, and the third for the columns). Let's create a rectangular matrix (3 by 7) and fill it with `NA`'s. `NA` is another useful special object existing in R. We have already seen `TRUE` and `FALSE`. `NA` is a kind of a special zero and most computations involving `NA` return NA (e.g. `NA + 1` evaluates to `NA`, and `NA == TRUE` returns `NA`). Also, say hello to the `dim()` function.

```
> Matrix2 <- matrix(NA, 3, 7)
> Matrix2
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]   NA   NA   NA   NA   NA   NA   NA
[2,]   NA   NA   NA   NA   NA   NA   NA
[3,]   NA   NA   NA   NA   NA   NA   NA

> dim(Matrix2) # this returns the dimensions of the matrix
[1] 3 7
```

Example 3:
Let's fill a matrix with a vector of values

```
> Vector8 <- 1:12
> Vector8
[1]  1  2  3  4  5  6  7  8  9 10 11 12

> Matrix3 <- matrix(data = Vector8, nrow = 4)
> Matrix3 # by default the matrix will be populated by column
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12

> Matrix4 <- matrix(data = Vector8, nrow = 4, byrow = TRUE)
> Matrix4 # now we populated it by row
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
```

Example 4:
You can also create matrices by pasting together vectors using the `rbind()` and `cbind()` functions.

```
> Vector9 <- 1:10
> Vector9
[1]  1  2  3  4  5  6  7  8  9 10

> Vector10 <- Vector9 ^ 2
> Vector10
 [1]   1   4   9  16  25  36  49  64  81 100

> Matrix5 <- rbind(Vector9, Vector10)
> Matrix5
         [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
Vector9     1    2    3    4    5    6    7    8    9    10
Vector10    1    4    9   16   25   36   49   64   81   100

> dim(Matrix5)
[1]  2 10
```

And cbind() ...

```
> Matrix6 <- cbind(Vector9, Vector10, Vector9)
> Matrix6
      Vector9 Vector10 Vector9
 [1,]       1        1       1
 [2,]       2        4       2
 [3,]       3        9       3
 [4,]       4       16       4
 [5,]       5       25       5
 [6,]       6       36       6
 [7,]       7       49       7
 [8,]       8       64       8
 [9,]       9       81       9
[10,]      10      100      10


> dim(Matrix6)
[1]  10 3
```

As you can see, the rbind() and cbind() functions automatically label row or column names. You can use the rownames() and colnames() functions to manipulate these.

```
> colnames(Matrix6)
[1] "Vector9"  "Vector10" "Vector9"

> rownames(Matrix6) # the rownames do not exist
NULL

> colnames(Matrix6) <- c("A", "B", "C")
> rownames(Matrix6) <- c("a","b","c","d","e","f","g","h","i",
   "j")

> Matrix6
   A   B  C
a  1   1  1
b  2   4  2
c  3   9  3
d  4  16  4
e  5  25  5
f  6  36  6
g  7  49  7
h  8  64  8
i  9  81  9
j 10 100 10
```

The diag() function is useful for creating the identity matrix

```
> Matrix7 <- diag(5) # creates a 5 by 5 identity matrix
> Matrix7
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1

> Vector11 <- c(1, 2, 3, 4, 5)
> Matrix8 <- diag(Vector11) # Vector11 across the diagonal
> Matrix8
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    2    0    0    0
[3,]    0    0    3    0    0
[4,]    0    0    0    4    0
[5,]    0    0    0    0    5

>  diag(Matrix7) # extracts the diagonal from the matrix
[1] 1 1 1 1 1
```

4.2.1 Indexing Matrices

It should be kind of obvious how indexing works with matrices from looking at the output **R** generates and knowing that vectors are indexed via [ ]. Using [i,j] will retrieve the *j*th element of the *i*th row.

```
> Matrix9 <- matrix(1:9, 3)
> Matrix9
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> Matrix9[1,1] # extracts the first element of the first row
[1] 1

> Matrix9[2,3] # extracts the third element of the second row

[1] 8
```

You can also extract entire rows as vectors by leaving the column entry blank, and vice versa.

```
> Matrix9
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> Matrix9[ ,1] # extracts the first column
[1] 1 2 3

> Matrix9[2, ] # extracts the second row
[1] 2 5 8

> Matrix9[1:2, ] # extracts the first and second row
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8

> Matrix9[Matrix9[ ,2] > 4, ] # extracts all rows that in
   their second column contain values greater than four
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    3    6    9
```

4.3 Mathematical Operations

R can do matrix arithmetic. Below is a list of some basic operations we can do.

→	`+ - * /`	standard scalar or by element operations
→	`%*%`	matrix multiplication
→	`t()`	transpose
→	`solve()`	inverse
→	`det()`	determinant
→	`chol()`	cholesky decomposition
→	`eigen()`	eigenvalues and eigenvectors
→	`crossprod()`	cross product
→	`%x%`	kronecker product

4.3.1 Matrix Math-Examples

```
> X <- matrix(1:4, nrow = 2)
> X
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> Y <- diag(2)
> Y
     [,1] [,2]
[1,]    1    0
[2,]    0    1

> X * Y # by element multiplication
     [,1] [,2]
[1,]    1    0
[2,]    0    4

> X %*% Y # matrix multiplication
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> t(X) # transpose
     [,1] [,2]
[1,]    1    2
[2,]    3    4

> solve(X) # inverse
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5

> sum(diag(X)) # trace
[1] 5

> det(X) # determinant
[1] 5-2
```

To compute eigenvalues and vectors you can use the eigen() function. Unlike most functions we have encountered so far, it does not return a vector or a matrix but a list.

```
> eigen(X)
$values
[1]  5.3722813 -0.3722813

$vectors
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736
```

As mentioned in before in Chapter 3.1, lists can be a collection of different types of *objects*. The eigen() function returns a list containing a vector and a matrix. You can extract named elements from a list with the $ symbol. Alternatively you can extract element with double [[ ]].

```
> Eigen.List <- eigen(X)
> names(Eigen.List) # prints the names of objects of the list
[1] "values"  "vectors"

> Eigen.List$vectors # returns the eigenvectors
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736

> Eigen.List$values # returns the eigenvalues
[1]  5.3722813 -0.3722813
```

4.3.2 Bonus Example

Last but not least, try this:

```
> Constant <- rep(1, times = 10)
> Variable <- c(1,2,3,1,2,3,5,6,7,8)
> X <- cbind(Constant, Variable)
> Y <- seq(1, 10)
> Beta.Hats <- solve(t(X)%*%X)%*%t(X)%*%Y
> colnames(Beta.Hats) <- "Estimate"
> Beta.Hats
         Estimate
Constant  1.34375
Variable  1.09375

> OMG!!!!
Error: unexpected '!' in "OMG!"
```