Statistical Computing Lab
Matrices and Linear Algebra in R

Giovanni Petris

Fall 2009

1 Matrices

To arrange values into a matrix, we use the `matrix()` function:

```
> a <- matrix(1 : 6, nrow = 2, ncol = 3)
> a

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

Individual entries can be referred to using a pair of indices. For example, the element in the second row, third column can be printed as

```
> a[2, 3]

[1] 6
```

An entire row or column can be retrieved by specifying its index and leaving the other index empty:

```
> a[2, ]

[1] 2 4 6

> a[ , 3]

[1] 5 6
```

Similarly to the way R works with vectors, you can use negative indices to exclude row or colums of a matrix.

```
> a[ -2, ]

[1] 1 3 5

> a[ , -3]

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

Note how the values 1,2,3,4,5,6 are used to create the matrix: the first two are used to fill the first column, then the next two to fill the second column, and so on. R allows matrices to be indexed by a single number, e.g.,

```
> a[5]

[1] 5
```

The fifth element of `a` is the fifth element of the vector obtained by "unrolling" the matrix, column by column. Sometimes you need to fill a matrix row by row, instead than column by column. You can change the default behavior as follows:

```
> a <- matrix(1 : 6, nrow = 2, ncol = 3, byrow = TRUE)
> a

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

Matrices can also be constructed by binding together smaller matrices or vectors. This is the purpose of the functions `rbind()` and `cbind()`, which bind together rows or columns, respectively.

```
> rbind(a, a)

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

> cbind(a, a)
```

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

For example, we can use `cbind()` to construct a 3×3 Hilbert matrix. The (i, j) entry is $1/(i + j - 1)$.

```
> H3 <- 1 / cbind(1 : 3, 2 : 4, 3 : 5)
> H3
```

```
      [,1]  [,2]  [,3]
[1,] 1.000 0.500 0.333
[2,] 0.500 0.333 0.250
[3,] 0.333 0.250 0.200
```

The diagonal of a matrix can be extracted using `diag()`.

```
> diag(H3)
```

```
[1] 1.000 0.333 0.200
```

The `diag()` function can also be used to turn a vector into a square matrix whose diagonal elements correspond to the entries of the given vector.

```
> diag(diag(H3))
```

```
     [,1]  [,2] [,3]
[1,]    1 0.000  0.0
[2,]    0 0.333  0.0
[3,]    0 0.000  0.2
```

The determinant of a square matrix can be obtained with `det()`.

1.0.1 Triangular matrices

The functions `lower.tri()` and `upper.tri()` can be used to obtain the lower and upper triangular parts of matrices. The output of the functions is a matrix of logical elements, with `TRUE` representing the relevant triangular elements. For example,

```
> lower.tri(H3)
```

```
      [,1]  [,2]  [,3]
[1,] FALSE FALSE FALSE
[2,]  TRUE FALSE FALSE
[3,]  TRUE  TRUE FALSE
```

A typical use of these functions is to set the upper or lower triangular part of a matrix to zero, thus constructing a triangular matrix. This is illustrated below.

```
> Htri <- H3
> Htri[lower.tri(Htri)] <- 0
> Htri

     [,1]  [,2]  [,3]
[1,]    1 0.500 0.333
[2,]    0 0.333 0.250
[3,]    0 0.000 0.200
```

2 Matrix multiplication and inversion

Matrix multiplication in R is performed by the operator `%*%`. For example,

```
> A <- matrix(c(1, 2, 0, 1), 2)
> B <- matrix(c(1, 3, 3, 1, 4, 5), nr = 2)
> A

     [,1] [,2]
[1,]    1    0
[2,]    2    1

> B

     [,1] [,2] [,3]
[1,]    1    3    4
[2,]    3    1    5

> A %*% B

     [,1] [,2] [,3]
[1,]    1    3    4
[2,]    5    7   13
```

The two matrices must *conform*, that is, the number of columns of the first must be the same as the number of rows of the second. While this is clearly the case in the previous example, you can obtain this information from R in several ways:

```
> dim(A)
```

```
[1] 2 2
```

```
> ncol(A) # or NCOL(A)
```

```
[1] 2
```

```
> nrow(B) # or NROW(B)
```

```
[1] 2
```

With `A` and `B` defined as above the matrix product `BA` in not defined, but the product `B'A` is (note that with `B'` I denote the transpose of `B`). The transpose of a matrix can be obtained with the function `t()`.

```
> t(B)
```

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

```
> t(B) %*% A
```

```
     [,1] [,2]
[1,]    7    3
[2,]    5    1
[3,]   14    5
```

A much more efficient way of computing the same matrix product is via the function `crossprod()`.

```
> all.equal(crossprod(B, A), t(B) %*% A)
```

```
[1] TRUE
```

A similar function, `tcrossprod()`, computes the matrix product `AB'` whenever the product is well defined. Consider the matrix

```
> X <- cbind(1, seq(-1, 1, length = 11))
> X
```

```
      [,1] [,2]
 [1,]    1 -1.0
 [2,]    1 -0.8
 [3,]    1 -0.6
 [4,]    1 -0.4
 [5,]    1 -0.2
 [6,]    1  0.0
 [7,]    1  0.2
 [8,]    1  0.4
 [9,]    1  0.6
[10,]    1  0.8
[11,]    1  1.0
```

Use `crossprod()` and `tcrossprod()` to find `X'X` and `XX'`. Note the dimensions of the resulting matrices.

Do it now!

2.0.2 Matrix inversion

The inverse of a matrix can be found using `solve()`:

```
> Ainv <- solve(A)
> Ainv
```

```
     [,1] [,2]
[1,]    1    0
[2,]   -2    1
```

```
> all.equal(Ainv %*% A, diag(2)) # check
```

```
[1] TRUE
```

The default method to find the inverse uses the QR decomposition of `A`, as discussed in class. Another point that I stressed in class is that if you only need the inverse to solve a linear system, you don't really need it.

```
> b <- c(5, 3)
> solve(A, b) # use this
```

```
[1]  5 -7
```

```
> Ainv %*% b  # don't use this
```

```
     [,1]
[1,]    5
[2,]   -7
```

In fact, `solve()` can be used for solving simultaneously a set of linear systems for different right-hand sides "b". Computing the inverse corresponds to solving n linear systems, where n is the dimension of A.

```
> solve(A, diag(nrow(A)))
```

```
     [,1] [,2]
[1,]    1    0
[2,]   -2    1
```

The QR decomposition of a matrix can be obtained using the function `qr()`. The output of the function is not easy to interpret. In particular, the Q and R matrices are stored in a compact form. To recover them, call `qr.Q()` and `qr.R()` on the output of `qr()`.

```
> H.qr <- qr(H3)
> H.qr
```

```
$qr
       [,1]   [,2]    [,3]
[1,] -1.167 -0.643 -0.4500
[2,]  0.429 -0.102 -0.1053
[3,]  0.286  0.729  0.0039

$rank
[1] 3

$qraux
[1] 1.8571 1.6842 0.0039

$pivot
[1] 1 2 3

attr(,"class")
[1] "qr"
```

```
> Q <- qr.Q(H.qr)
> Q

       [,1]   [,2]   [,3]
[1,] -0.857  0.502  0.117
[2,] -0.429 -0.568 -0.702
[3,] -0.286 -0.652  0.702

> R <- qr.R(H.qr)
> R

      [,1]   [,2]    [,3]
[1,] -1.17 -0.643 -0.4500
[2,]  0.00 -0.102 -0.1053
[3,]  0.00  0.000  0.0039
```

Verify, using `lower.tri()` and logical operators, that `R` is upper triangular. Verify also that `Q` has orthonormal columns, i.e., that `Q'Q` is an identity matrix.

Do it now!

3 Some comparisons

It may be interesting to compare execution times for the different methods of solving the Least Squares problem when the number of observations is large. Let us start by making up our own data, at random.

```
> set.seed(123)
> n <- 5e5
> p <- 20
> X <- matrix(rnorm(n * p, mean = 1 : p, sd = 10), nr = n, nc = p, byrow = TRUE)
> y <- rowSums(X) + rnorm(n)
```

The naive approach to solve the LS problem is to compute $\hat{\beta} = (X'X)^{-1}X'y$.

```
> system.time(bHat1 <- solve(t(X) %*% X) %*% t(X) %*% y)

   user  system elapsed
  2.668   0.224   2.896
```

```
> drop(bHat1)

 [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

A way more efficient way to compute the same quantity is put `crossprod()` and `solve()` to good use.

```
> system.time(bHat2 <- solve(crossprod(X), crossprod(X, y)))

   user  system elapsed
  1.035   0.004   1.116

> all.equal(bHat1, bHat2)

[1] TRUE
```

A different approach is to use Choleski factorization. Note that in R the function `chol()` returns the *upper* factor.

```
> system.time(bHat3 <- {
+     R <- chol(crossprod(X))
+     backsolve(R, forwardsolve(R, crossprod(X, y),
+                               upper.tri = TRUE, transpose = TRUE))})

   user  system elapsed
  1.040   0.001   1.040

> all.equal(bHat1, bHat3)

[1] TRUE
```

Finally, the QR decomposition can be used to solve the LS problem.

```
> system.time(bHat4 <- {
+     qrX <- qr(X)
+     R <- qr.R(qrX)
+     backsolve(R, crossprod(qr.Q(qrX), y))})

   user  system elapsed
   9.73    1.98   32.07

> all.equal(bHat1, bHat4)

[1] TRUE
```

Although relatively slow, the approach based on the QR decomposition is the recommended one – and the one used by R itself in fitting linear models – because of its numerical stability properties.

4 Arrays

A generalization of a matrix is an *array*. Arrays have multiple indices and can be created using the function `array()`.

```
> b <- array(1 : 24, dim = c(3, 4, 2))
> b

, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

The dimensions in the example above are specified by the vector `c(3, 4, 2)`. When inserting data, the first index varies fastest; when it has run through its full range, the second index changes, etc.