R Programming
Matrices

Matrices

A *matrix* is a rectangular arrangement of values which are all of the same basic type. For example,

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

arranges the numbers $1, \ldots, 6$ in three rows and two columns. This is a 3×2 matrix of numbers.

In R, the elements of a matrix are stored in a vector with the elements of the first column coming first followed by those of the second, and so on.

This is known as *column-major order* storage.

Creating Matrices

The matrix on the previous slide can be created as follows

```
> matrix(1:6, nrow = 3, ncol = 2)
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

An optional argument by`row=TRUE` can be used to arrange the vector of values by row.

```
> matrix(1:6, nrow = 3, ncol = 2, byrow = TRUE)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

Matrix Elements and Recycling

The first argument to the `matrix` function contains the elements which are to form the matrix. If there are not enough elements in the argument to create the matrix, the recycling rule is applied to obtain more.

A 2×3 matrix filled with 1s can be obtained as follows.

```
> matrix(1, nrow = 2, ncol = 3)
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
```

Optional Dimension Specifications

Often R can work out the number of rows in a matrix given the number of columns and the elements, or the the number of columns in a matrix given the number of rows and the elements. In such cases it is not necessary to specify both the number of rows and the number of columns.

```
> matrix(1:6, nrow = 3)
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Determining Matrix Dimensions

The number of rows and columns or a matrix can be obtained with the functions `nrow` and `ncol`. Or obtained together with the function `dim` which returns a vector containing the number of rows as the first element can the number of columns as its second.

```
> x = matrix(1:6, nc = 2)
> nrow(x)
[1] 3
> ncol(x)
[1] 2
> dim(x)
[1] 3 2
```

Creating Matrices from Rows and Columns

Matrices can be created by gluing together rows with `rbind` of gluing together columns with `cbind`.

```
> cbind(1:3, 4:6)
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> rbind(1:3, 4:6)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Binding Rows and Columns; Recycling

The arguments to `rbind` and `cbind` are not required to be the same length length. When they are not, a matrix is created which is big enough to accommodate the largest argument, and the others have the recycling rule applied to them to supply additional arguments. Apparent mismatches produce warnings.

```
> rbind(1:2, 1:3)
     [,1] [,2] [,3]
[1,]    1    2    1
[2,]    1    2    3
Warning message:
In rbind(1:2, 1:3) :
  number of columns of result is not
  a multiple of vector length (arg 1)
```

Matrices and Naming

It is possible to attach row and column labels to matrices. Rownames can be attached with `rownames`, column names with `colnames`, and both can be attached simultaneously with `dimnames`.

```
> x = matrix(1:6, nrow = 2)
> dimnames(x) = list(c("First", "Second"),
                     c("A", "B", "C"))
> x
       A B C
First  1 3 5
Second 2 4 6
```

Extracting Names

Names can also be extracted with `dimnames`, `rownames` and `colnames`.

```
> dimnames(x)
[[1]]
[1] "First"  "Second"

[[2]]
[1] "A" "B" "C"

> rownames(x)
[1] "First"  "Second"
> colnames(x)
[1] "A" "B" "C"
```

Extracting Matrix Elements

The `ij`-th element of a matrix `x` can be extracted with the expression `x[i, j]`. This means, for example, that we could sum all the elements in a matrix `x` with the following code.

```
> s = 0
> for(i in 1:nrow(x))
      for(j in 1:ncol(x))
          s = s + x[i, j]
```

Because matrices are just vectors with additional dimensioning information it is actually more efficient to use

```
> s = sum(x)
```

Matrix Subsets

Expressions of the form `x[i, j]` can also be used to extract more general subsets of the elements of a matrix by specifying vector subscripts.

```
> x = matrix(1:12, nrow = 3, ncol = 4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> x[1:2, c(2, 4)]
     [,1] [,2]
[1,]    4   10
[2,]    5   11
```

Assigning to Matrix Subsets

It is also possible to assign to subsets of matrices.

```
> x[1:2, c(2, 4)] = 21:24
> x
     [,1] [,2] [,3] [,4]
[1,]    1   21    7   23
[2,]    2   22    8   24
[3,]    3    6    9   12
> x[2:1, c(2, 4)] = 21:24
> x
     [,1] [,2] [,3] [,4]
[1,]    1   22    7   24
[2,]    2   21    8   23
[3,]    3    6    9   12
```

Specifying Entire Rows and Columns

When a subscript is omitted, it is taken to correspond to all possible values. This works for extracting value from matrices and for assigning to them.

```
> x = matrix(1:12, nrow = 3, ncol = 4)
> x[1,] = 100
> x
     [,1] [,2] [,3] [,4]
[1,]  100  100  100  100
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Subsetting Matrices as Vectors

Because matrices are just vectors with additional dimensioning information they can be treated as vectors.

```
> x = matrix(1:6, nrow = 2, ncol = 3)
> x[7]
[1] NA
```

The functions `row` and `col` return matrices indicating the row and column of each element. This can be used to extract or change submatrices.

```
> x[row(x) < col(x)] = 0
> x
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    2    4    0
```

An Example

This is how to create a 4×4 tridiagonal matrix with diagonal values being 2 and the off-diagonal elements being 1.

```
> x = matrix(0, nrow = 4, ncol = 4)
> x[row(x) == col(x)] = 2
> x[abs(row(x) - col(x)) == 1] = 1
> x
     [,1] [,2] [,3] [,4]
[1,]    2    1    0    0
[2,]    1    2    1    0
[3,]    0    1    2    1
[4,]    0    0    1    2
```

Simple Computations with Matrices

The basic mathematical manipulations on matrices are defined by using the fact that the operations are defined for the underlying vectors.

```
> x = matrix(1:4, nrow = 2, ncol = 2)

> x
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> x + x^2
     [,1] [,2]
[1,]    2   12
[2,]    6   20
```

Combining Vectors and Matrices

Some checks are carried out to try to make sure that operations are sensible. This can produce warnings.

```
> x + 1:3
     [,1] [,2]
[1,]    2    6
[2,]    4    5
Warning message:
In x + 1:3 :
  longer object length is not a multiple of shorter
        object length
```

Note that it is an error to try to combine a matrix with a vector which has more elements than the matrix.

The "Apply" Mechanism

Because this is such a common kind of task, R has a special way of carrying out this kind of operation. The function `apply` can be used to compute row or column summaries. The expression

`apply(`*matrix*`, 1, `*summary*`)`

computes row summaries of the type specified by *summary* for the matrix specified by *matrix*. Similarly,

`apply(`*matrix*`, 2, `*summary*`)`

computes column summaries.

Apply and Anonymous Functions

In the preceding examples, we've used a function name to specify which summary function the `apply` mechanism should use. It is also possible to specify the function directly by using its definition as the argument to apply. Here is how we can compute the column sums of squares for `x`.

```
> apply(x, 2, function(x) sum(x^2))
[1]  5 25
```

Computing the sums of squares about the column means is just as easy.

```
> apply(x, 2, function(x) sum((x - mean(x))^2))
[1] 0.5 0.5
```

Combining Vectors and Matrices

When a vector is added to a matrix, the recycling rule is used to expand the vector so that it matches the number of elements in the matrix.

```
> x + 1
     [,1] [,2]
[1,]    2    4
[2,]    3    5

> x + 1:2
     [,1] [,2]
[1,]    2    4
[2,]    4    6
```

Row and Column Summaries

A common thing to want to do with matrices is to obtain a vector containing a numerical summary for each row (or each column). It is possible to to this by using `for` loops. As a simple example, we'll look at computing row (or column) means. We can do this as follows:

```
> rm = numeric(nrow(x))
> for(i in 1:nrow(x))
      rm[i] = mean(x[i,])
```

Notice that there is nothing special about using the `mean` function here. The same code will work for any numerical summary.

Computing Row and Column Means

Row and column means of `x` can be computed as follows:

```
> apply(x, 1, mean)
[1] 2 3

> apply(x, 2, mean)
[1] 1.5 3.5
```

There is no difficulty in substituting any other summary function in place of `mean`. For example, the row standard deviations can be computed as follows.

```
> apply(x, 1, sd)
[1] 1.414214 1.414214
```

More Complex Summaries

The first subscript of the result ranges over the set of values returned by the summary function while the second specifies which row or column the summary function is being computed for.

```
> apply(x, 1, range)
     [,1] [,2]
[1,]    1    2
[2,]    3    4

> apply(x, 2, range)
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Additional Arguments to Summaries

It is also possible to specify additional arguments for the *summary* function as extra arguments to the `apply` call. To see an example, let's return to the `mean` function, which has an optional argument `trim`, which specifies a fraction of the observations to trimmed from its argument before the mean is computed. The largest and smallest `trim/2` fraction of the observations are trimmed before the mean is computed. A value of `trim=.1` could be specified in a call to `apply` as follows.

```
apply(matrix, margin, mean, trim = .1)
```

Sweeping Out Summaries

Once summaries have been computed, it is common to want to subtract out those summaries to obtain residual values. As with computing the summaries themselves, this can be done by looping over the rows or columns, as appropriate. Because this is quite a common task, there is a special function called `sweep` which can be used to sweep out computed summaries. To subtract out row summaries from the rows of a matrix use a call of the form

```
sweep(matrix, 1, apply(matrix, 1, summary))
```

and to subtract out column summaries use a call of the form

```
sweep(matrix, 2, apply(matrix, 2, summary))
```

Other Forms of Sweeping

By default, sweep will subtract out the summaries, but it is possible to use any other binary operation instead. For example, the following statement forms will divide through by the summary, rather than subtracting.

```
sweep(matrix, 1, apply(matrix, 1, summary), "/")
sweep(matrix, 2, apply(matrix, 2, summary), "/")
```

As a concrete example, here is how to divide the columns of a matrix by their column means.

```
> x = cbind(1:3, 9:11)
> sweep(x, 2, apply(x, 2, mean), "/")
     [,1] [,2]
[1,]  0.5  0.9
[2,]  1.0  1.0
[3,]  1.5  1.1
```

A Statistical Example

Consider the following data showing child mortality in regions of the United States, all races, 1964–1966 (deaths per 1000 live births). This is typical of data encountered in statistical data analysis.

```
> mortality
               Father's Education (Years)
Region           <=8 9-11   12 13-15 >=16
  Northeast     25.3 25.3 18.2  18.3 16.3
  North Central 32.1 29.0 18.8  24.3 19.0
  South         38.8 31.0 19.3  15.7 16.8
  West          25.4 21.1 20.3  24.0 17.5
```

Creating The Matrix

```
> mortality =
    matrix(c(25.3, 25.3, 18.2, 18.3, 16.3,
             32.1, 29.0, 18.8, 24.3, 19.0,
             38.8, 31.0, 19.3, 15.7, 16.8,
             25.4, 21.1, 20.3, 24.0, 17.5),
           nrow = 4, byrow = TRUE,
           dimnames = list(
             Region =
               c("Northeast", "North Central",
                 "South", "West"),
             "Father's Education (Years)" =
               c("<=8", "9-11", "12",
                 "13-15", ">=16")))
```

An "Overall + Rows + Columns" Model

A common way of dealing with a table of values like this is to try to explain the pattern in the table with in terms of some simpler underlying structure. In the case of the mortality data, we will look at how to fit an overall plus row-effect plus column-effect model to describe these values. This means that we will seek to represent the values y_{ij} in the matrix by

$$y_{ij} = \mu + \alpha_i + \beta_j + \varepsilon_{ij}$$

where μ represents the overall level of the data, α_i is an effect common to all the values in the i-th row, β_j is a value common to all values in the j-th column and the ε_{ij} are (small) residuals which describe the deviations between what the model predicts and the observed values.

Fitting By Sweeping Out Effects

We will fit this model by progressively *sweeping out* the effects from a set of *working residuals*. To begin we will estimate the overall level as the average value and then subtracting away from the original values. This gives a set of deviations about the overall mean.

```
> r = mortality
> mu = mean(r)
> r = r - mu

> mu
[1] 22.825
```

Sweeping Out Row Effects

Next we compute the row effects as the row means of y and subtract them from y. The easy way to do this is using `apply` and `sweep`.

```
> alpha = apply(r, 1, mean)
> r = sweep(r, 1, alpha)

> alpha
    Northeast North Central         South
       -2.145         1.815         1.495
         West
       -1.165
```

Sweeping Out Column Effects

Finally, we extract the column effects as the column means of y and subtract them from y. Again, this can be done with `sweep`.

```
> beta = apply(r, 2, mean)
> r = sweep(r, 2, beta)

> beta
   <=8   9-11     12  13-15   >=16
 7.575  3.775 -3.675 -2.250 -5.425
```

The Residuals

After completing the process, we are left with a set of *residuals* from the fit.

```
> r
              Father's Education (Years)
Region            <=8   9-11     12 13-15   >=16
  Northeast     -2.955  0.845  1.195 -0.13  1.045
  North Central -0.115  0.585 -2.165  1.91 -0.215
  South          6.905  2.905 -1.345 -6.37 -2.095
  West          -3.835 -4.335  2.315  4.59  1.265
```

Remaining Effects?

To see that all the effects have been swept out of the residuals we can compute the row and column means of the residuals.

```
> round(apply(r, 1, mean), 4)
    Northeast North Central         South
            0             0             0
         West
            0

> round(apply(r, 2, mean), 4)
  <=8  9-11    12 13-15  >=16
    0     0     0     0     0
```

These are (essentially) zero.

Interaction?

Sometimes the residuals show patterns that indicate that a model does not fit as well as it might. There is a slight suggestion of an *interaction* effect in these residuals.

```
> r[order(alpha), order(beta)]
              Father's Education (Years)
Region            >=16     12 13-15   9-11    <=8
  Northeast      1.045  1.195 -0.13  0.845 -2.955
  West           1.265  2.315  4.59 -4.335 -3.835
  South         -2.095 -1.345 -6.37  2.905  6.905
  North Central -0.215 -2.165  1.91  0.585 -0.115
```

Packaging as a Function

```
> twoway =
      function(y)
      {
          mu = mean(y)
          y = y - mu
          alpha = apply(y, 1, mean)
          y = sweep(y, 1, alpha)
          beta = apply(y, 2, mean)
          y = sweep(y, 2, beta)
          list(overall = mu, rows = alpha,
               cols = beta, residuals = y)
      }
```

Commenting

I omitted comments from the function on the previous slide in order to save space. When writing computer code it is **vital** to include informative comments.

```
⋮

## ... Compute and remove the overall mean
mu = mean(y)
y = y - mu

## ... Compute and remove the row effects
alpha = apply(y, 1, mean)
y = sweep(y, 1, alpha)

⋮
```

(Generalized) Outer Products

The function `outer` provides another useful utility in connection with matrix calculations. In mathematics, the outer product of two vectors **x** and **y** is a matrix whose ij-th element is

$$x_i \times y_j.$$

`outer` generalizes this so that, when given a function f, the computed result is a matrix whose ij-th element is

$$f(x_i, y_j).$$

The default function value in R is $f(x,y) = x \times y$, which produces an ordinary outer product.

Example: A Multiplication Table

Here is a small multiplication table (truncated at 8 so that it fits on a slide.

```
> outer(1:8, 1:8)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    2    3    4    5    6    7    8
[2,]    2    4    6    8   10   12   14   16
[3,]    3    6    9   12   15   18   21   24
[4,]    4    8   12   16   20   24   28   32
[5,]    5   10   15   20   25   30   35   40
[6,]    6   12   18   24   30   36   42   48
[7,]    7   14   21   28   35   42   49   56
[8,]    8   16   24   32   40   48   56   64
```

Example: Generating Labels

```
> outer(LETTERS[1:8], 1:5,
        function(x, y) paste(x, y, sep=""))
     [,1] [,2] [,3] [,4] [,5]
[1,] "A1" "A2" "A3" "A4" "A5"
[2,] "B1" "B2" "B3" "B4" "B5"
[3,] "C1" "C2" "C3" "C4" "C5"
[4,] "D1" "D2" "D3" "D4" "D5"
[5,] "E1" "E2" "E3" "E4" "E5"
[6,] "F1" "F2" "F3" "F4" "F5"
[7,] "G1" "G2" "G3" "G4" "G5"
[8,] "H1" "H2" "H3" "H4" "H5"
```

Example: Generating Labels (Cont)

```
> c(outer(LETTERS[1:8], 1:5,
          function(x, y) paste(x, y, sep="")))
 [1] "A1" "B1" "C1" "D1" "E1" "F1" "G1" "H1" "A2"
[10] "B2" "C2" "D2" "E2" "F2" "G2" "H2" "A3" "B3"
[19] "C3" "D3" "E3" "F3" "G3" "H3" "A4" "B4" "C4"
[28] "D4" "E4" "F4" "G4" "H4" "A5" "B5" "C5" "D5"
[37] "E5" "F5" "G5" "H5"

> c(outer(1:5, LETTERS[1:8],
          function(x, y) paste(y, x, sep="")))
 [1] "A1" "A2" "A3" "A4" "A5" "B1" "B2" "B3" "B4"
[10] "B5" "C1" "C2" "C3" "C4" "C5" "D1" "D2" "D3"
[19] "D4" "D5" "E1" "E2" "E3" "E4" "E5" "F1" "F2"
[28] "F3" "F4" "F5" "G1" "G2" "G3" "G4" "G5" "H1"
[37] "H2" "H3" "H4" "H5"
```

A More Complex Example

- Using `apply` and `outer` together provides a very powerful way of carrying out quite complex computations.
- In this example we'll consider the problem of finding the closest match for each value $y_1, \ldots, y_m$ in a set of values $x_1, \ldots, x_n$ (a kind of nearest neighbour regression).
- While the problem can be carried out using nested for loops, there is a one is a "one-liner" solution.

The `which` Function

- The `which` function accepts a vector of logical values and returns a vector that indicates which indices correspond to true values.

```
> x = c(-1, 0, 1)
> which(x >= 0)
[1] 2 3
> which(x == 0)
[1] 2
```

- The function is easy to implement.

```
> which = function(x) (1:length(x))[x]
```

Nearest Neighbour Matches

The algorithm is as follows:

1. Form the matrix D so that the ijth element of D is the distance between y_i and x_j (this can be done with `outer`.
2. Determine the column index of the smallest value in each row of D.
3. Return the x value corresponding to this index.

Nearest Neighbour Matches

```
> nearest =
      function(x, y)
      x[apply(outer(y, x,
                    function(y, x)
                    abs(y - x)),
              1,
              function(x)
              which(x == min(x))[1])]

> x = 0:10/10
> y = round(runif(4), 4)
> y
[1] 0.8734 0.6996 0.7339 0.5312
> nearest(x, y)
[1] 0.9 0.7 0.7 0.5
```

Matrix Transposes

Often it useful to be able to interchange the role of rows and columns in a matrix. This is refered to as taking the *transpose* of a matrix. R has a special purpose function called `t` which can be used to compute matrix transposes. It is very simple to use.

```
> x = cbind(1:3, 11:13)
> t(x)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]   11   12   13
```

Matrix Diagonals

- Given a matrix `A`, the function `diag` returns the diagonal elements of that matrix. Note that `diag(A)` is equivalent to `A[row(A)==col(A)]`.
- Given a vector `a`, the expression `diag(a)` returns a diagonal matrix with the elements of `a` on the diagonal.
- Given a positive integer `n` the expression `diag(n)` returns an $n \times n$ identity matrix.

Matrix Products

Matrices are often found in mathematical expressions involving *matrix multiplication*. Mathematically, the product of an $m \times n$ matrix A whose ij-th element is a_{ij} and an $n \times p$ matrix B whose ij-th element is b_{ij} is defined to be the $m \times p$ matrix whose ij-th element is

$$\sum_{k=1}^{n} a_{ik} b_{kj}$$

The product of matrices `A` and `B` is computed in R by the expression

```
A %*% B
```

Systems of Linear Equations

The set of equations

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \cdots & + & a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & \vdots & & \vdots & & \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \cdots & + & a_{mn}x_n & = & b_n \end{array}$$

can be written in the matrix form

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix},$$

or

$$\mathbf{Ax} = \mathbf{b}.$$

Solving Systems of Linear Equations

- Given a non-singular $n \times n$ matrix `A` and vector `b` of length n, the linear system $\mathbf{Ax} = \mathbf{b}$ can be solved with the R expression `solve(A, b)`.
- Solutions to multiple right-hand sides can be obtained by assembling them as the columns of a matrix `B` and using the expression. `solve(A, B)`.
- The expression `solve(A)` computes the inverse of `A` (because the default value for `b` is a suitable identity matrix).

Example: Formulae for Sums of Powers

There is a well known formula for the sum of the first n positive integers.

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Let's examine the more general problem of finding a formula for the sum of the kth powers of the first n positive integers.

$$S_{nk} = 1^k + 2^k + \cdots + n^k$$

Note that for all n

$$S_{nk} \le \underbrace{n^k + n^k + \cdots + n^k}_{n \text{ tems}} = n \times n^k = n^{k+1}$$

so that S_{nk} is probably a polynomial of degree $\le k+1$.

A System of Equations

Let's suppose that

$$S_{nk} = c_0 + c_1 n + c_2 n^2 + \cdots c_{n+1} n^{k+1}$$

By considering what happens for $n = 1, 2, \ldots, k+2$ we get a series of simultaneous equations which can be solved for $c_0, \ldots c_{k+1}$.

$$\begin{array}{rcl} S_{1k} & = & c_0 + c_1 1 + c_2 1^2 + c_{k+1} 1^{k+1} \\ S_{2k} & = & c_0 + c_1 2 + c_2 2^2 + c_{k+1} 2^{k+1} \\ & \vdots & \\ S_{k+2,k} & = & c_0 + c_1 (k+2) + c_2 (k+2)^2 + c_{k+1} (k+2)^{k+1} \end{array}$$

A Matrix System

The equations can be written in matrix form as

$$\begin{pmatrix} 1 & 1 & 1^2 & \cdots & 1^{k+1} \\ 1 & 2 & 2^2 & \cdots & 2^{k+1} \\ \vdots & & & & \vdots \\ 1 & k+2 & (k+2)^2 & \cdots & (k+2)^{k+1} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{k+1} \end{pmatrix} = \begin{pmatrix} S_{1k} \\ S_{2k} \\ \vdots \\ S_{k+2,k} \end{pmatrix}.$$

To determine the values of $c_0, \ldots c_{k+1}$, we only need to create the coefficient matrix and vector of right-hand sides, and then to use `solve`.

System Components

The coefficient matrix can be obtained in a number of ways. One obvious way is to create the matrix and then fill the elements in one by one.

```
> A = matrix(1, nr = k + 2, nc = k + 2)
> for(i in 1:(k+2))
    for(j in 1:(k+2))
      A[i, j] = i^(j-1)
```

This produces the right answer, but a much cleaner implementation can be obtained by using the `outer` function.

```
> A = outer(1:(k+2), 0:(k+1), "^")
```

The right-hand sides can generated quite simply using the `cumsum` function.

```
> b = cumsum(seq(k+2)^k)
```

A Function

Using the statements on the previous slide, it is very easy to create a function which returns the coefficients $c_0, \ldots c_{k+1}$.

```
> sumpow =
    function(k)
    solve(outer(1:(k+2), 0:(k+1), "^"),
          cumsum(seq(k+2)^k))
```

For $k = 1$ the result is

```
> sumpow(1)
[1] 0.0 0.5 0.5
```

which corresponds to

$$S_{n1} = \frac{n}{2} + \frac{n^2}{2} = \frac{n(n+1)}{2}.$$

The Formula for $k = 2$

For $k = 2$,

```
> sumpow(2)
[1] 1.360023e-15 1.666667e-01 5.000000e-01
[4] 3.333333e-01
```

The first coeffient is very close to 0, the deviation being due to small numerical errors which occured during the calculation. The other coefficients are very close to simple fractions. (We could use the functions `cf.expand` and `rat.approx` developed earlier to work out the values.)
The formula is

$$S_{n2} = \frac{n}{6} + \frac{n^2}{2} + \frac{n^3}{3} = \frac{n + 3n^2 + 2n^3}{6} = \frac{n(n+1)(2n+1)}{6}.$$

The Formula for $k = 3$

For $k = 3$,

```
> sumpow(3)
[1] 0.00 0.00 0.25 0.50 0.25
```

The formula is

$$S_{n3} = \frac{n^2}{4} + \frac{n^3}{2} + \frac{n^4}{4} = \frac{n^2(1 + 2n + n^2)}{4} = \frac{n^2(n+1)^2}{4}.$$

Regression Analysis

The general linear model can be written in matrix form as

$$\mathbf{y} = \mathbf{X}\beta + \varepsilon.$$

The least-squares estimates of the parameters are given by

$$\widehat{\beta} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$$

and the residuals by

$$\widehat{\varepsilon} = \mathbf{y} - \mathbf{X}\widehat{\beta}.$$

The estimated dispersion matrix of $\widehat{\beta}$ is

$$\mathscr{D}[\widehat{\beta}] = \frac{\widehat{\varepsilon}'\widehat{\varepsilon}}{n - p}(\mathbf{X}'\mathbf{X})^{-1}.$$

Regression In R

The equations for regression analysis can be translated directly into R statements.

```
> n = nrow(x)
> p = ncol(x)
> betahat = solve(t(x) %*% x, t(x) %*% y)
> epsilonhat = y - x %*% betahat
> sigmahat2 = sum(epsilonhat^2) / (n - p)
> D = sigmahat2 * solve(t(x) %*% x)
```

Warning: This is a direct coding of the mathematical equations associated with regression analysis. It is **not** the best way of computing the results.

Least Squares

The least square problem is to determine the value of β that minimizes the norm of the residuals

$$\mathbf{r} = \mathbf{y} - \mathbf{X}\beta.$$

This must occur when $\mathbf{r}$ is orthogonal to the columns of $\mathbf{X}$. In other words when

$$\mathbf{X}'(\mathbf{y} - \mathbf{X}\beta) = \mathbf{0}.$$

The QR Decomposition

Any $n \times p$ matrix $\mathbf{X}$ can be written in the form

$$\mathbf{X} = \mathbf{QR}$$

where $\mathbf{Q}$ is an $n \times p$ matrix whose columns are orthogonal and $\mathbf{R}$ is an upper triangular $p \times p$ matrix.

Since $\mathbf{Q}$ is orthogonal $\mathbf{Q}'\mathbf{Q} = \mathbf{D}$ where $\mathbf{D}$ is diagonal.

If $\mathbf{X}$ is non-singular then so is $\mathbf{R}$.

Using the QR Decomposition

Remember that the least-squares solution β satisfies

$$\mathbf{0} = \mathbf{X}'(\mathbf{y} - \mathbf{X}\beta).$$

Substituting for $\mathbf{X}$ in this

$$\begin{aligned}\mathbf{0} &= \mathbf{R}'\mathbf{Q}'(\mathbf{y} - \mathbf{QR}\beta) \\ &= \mathbf{R}'(\mathbf{Q}'\mathbf{y} - \mathbf{Q}'\mathbf{QR}\beta) \\ &= \mathbf{R}'(\mathbf{Q}'\mathbf{y} - \mathbf{DR}\beta).\end{aligned}$$

Since if we assume that $\mathbf{X}$ and hence $\mathbf{R}$ is non-singular, this means

$$\mathbf{0} = (\mathbf{Q}'\mathbf{y} - \mathbf{DR}\beta)$$

or

$$\mathbf{R}\beta = \mathbf{D}^{-1}\mathbf{Q}'\mathbf{y}.$$

The Least Squares Solution

We know that

$$\mathbf{R}\beta = \mathbf{D}^{-1}\mathbf{Q}'\mathbf{y}.$$

This is very easy to solve for β.

1. We form the vector $\mathbf{Q}'\mathbf{y}$ and divide its elements through by the diagonal elements of $\mathbf{D}$ to obtain a vector $\mathbf{z}$.
2. We solve the $p \times p$ matrix system

   $$\mathbf{R}\beta = \mathbf{z}$$

   for β. This is easy because $\mathbf{R}$ is triangular and so the system can be solved by back-substition.

This recipe provides a way of solving least-squares problems which is both fast and accurate. It is much more accurate than solving the normal equations.

R Functions

- The R function `qr` computes the QR decomposition.
- The functions `qr.coef`, `qr.qy`, `qr.qty`, `qr.resid` and `qr.fitted` compute useful quantities directly from the QR decomposition.
- The function `lsfit` carries out regression computations on a matrix `X` and a vector `y`.

The Singular Value Decomposition

Any $n \times p$ matrix $\mathbf{X}$ (with $n > p$) can be written as

$$\mathbf{X} = \mathbf{UDV}',$$

where $\mathbf{U}$ is an $n \times p$ orthogonal matrix, $\mathbf{D}$ is a $p \times p$ diagonal matrix and $\mathbf{V}$ is an $p \times p$ orthogonal matrix.

This decomposition is known as the *singular value decomposition* of $\mathbf{X}$ and the diagonal elements $d_1, \ldots, d_p$ of $\mathbf{D}$ are known as the *singular values* of $\mathbf{X}$.

The singular value decomposition is very important in statistics, particularly in regression and multivariate analysis.

Generalized Inverses

Suppose that $\mathbf{A}$ is an $m \times n$ matrix. A matrix $\mathbf{A}^-$ is called the Moore-Penrose pseudoinverse of $\mathbf{A}$ if

1. $\mathbf{AA}^-\mathbf{A} = \mathbf{A}$.
2. $\mathbf{A}^-\mathbf{AA}^- = \mathbf{A}^-$.
3. $\mathbf{AA}^-$ is symmetric.
4. $\mathbf{A}^-\mathbf{A}$ is symmetric.

Pseudoinverses can often play the role of inverses in cases where inverses do not exist. For example, when the design matrix $\mathbf{X}$ in a regression problem is singular, the least-squares coefficients can still be obtained as

$$(\mathbf{X}'\mathbf{X})^-\mathbf{X}'\mathbf{y}.$$

Computing Generalized Inverses

Suppose that $\mathbf{X}$ has the singular decomposition $\mathbf{UDV}'$. The Moore-Penrose pseudoinverse is then

$$\mathbf{VD}^-\mathbf{U}'$$

Where $\mathbf{D}^-$ is the pseudoinverse of $\mathbf{D}$. For a diagonal matrix such as $\mathbf{D}$, we get the pseudoinverse by taking the reciprocal of each non-zero elements on the diagonal, and leaving the zeros in place.

(In floating-point computations the test for exact zeros is usually replaced by a test of whether the values fall below some small tolerance in absolute value.)

A Generalised Inverse Function

A highly simplified `ginv` function could be written as follows.

```
> ginv =
      function(x, tol = 1e-7) {
          z = svd(x)
          dminus = ifelse(z$d > tol, 1/z$d, 0)
          z$v %*% diag(dminus) %*% t(x$u)
      }
```

A more robust function (which also deals with the $n < p$ case) can be found in the MASS library.

Regression Via the SVD

The regression model

$$\mathbf{y} = \mathbf{X}\beta + \boldsymbol{\varepsilon}.$$

can be fitted using the SVD. This method works even when $\mathbf{X}$ is singular.

Let $\mathbf{UDV}'$ denote the singular value decomposition of $\mathbf{X}$. Remember that we are seeking to minimise the norm of

$$\mathbf{y} - \mathbf{X}\beta = \mathbf{y} - \mathbf{XVV}'\beta.$$

Since orthogonal matrices preserve norms, this is the same as

$$\mathbf{U}'(\mathbf{y} - \mathbf{XVV}'\beta).$$

Regression Via the SVD (Cont)

$$\begin{aligned}\mathbf{U}'(\mathbf{y} - \mathbf{XVV}'\beta) &= \mathbf{U}'\mathbf{y} - \mathbf{U}'\mathbf{UDV}'\mathbf{VV}'\beta \\ &= \mathbf{U}'\mathbf{y} - \mathbf{DV}'\beta \\ &= \mathbf{z} - \mathbf{D}\gamma.\end{aligned}$$

The minimum possible value is obtained when

$$\gamma_j = \begin{cases} z_j/d_j & \text{if } d_j \neq 0, \\ \text{anything} & \text{if } d_j = 0. \end{cases}$$

To make the solution unique, it is common to set $\gamma_j = 0$ when $d_j = 0$.

To obtain β, use the fact that

$$\beta = \mathbf{V}\gamma.$$