

Description of R

- ▶ An environment for statistical computing and graphics
- ▶ R was originally created by Ross Ihaka and Robert Gentleman (hence the name R) at the University of Auckland, New Zealand.
- ▶ Operates in Windows, Mac, or Unix. We'll focus on Windows.
- ▶ It's free! The main website is <http://www.r-project.org/>.

Getting help

- ▶ If you need more information about a function, say “lm”, enter

```
>?lm
```

- ▶ To search for help on a specific topic use R's search page www.r-project.org/search.html.
- ▶ For more general information use R's manuals, www.cran.r-project.org/manuals.html.

Running R in Windows

- ▶ Code can be entered one-line-at-a-time at the command prompt.
- ▶ Batches of code can also be saved in a file, say `code.R`, and run all at once.
- ▶ Click File/Source R Code to browse for the code
- ▶ Or, if you know the code's location, at the command prompt enter the code

```
> source("K : /code.R")
```

Saving output

- ▶ To view a list of objects in the current workspace, enter

```
> ls()
```

- ▶ To save these objects, click File/Save workspace, and enter a name for the workspace.

- ▶ Or at the command prompt enter

```
> save.image("K : /mywork.RData")
```

- ▶ Workspaces can be large. It's often wise to delete individual objects that are no longer needed (e.g., the matrix X) before saving the workspace, i.e.,

```
> rm(X)
```

Data forms - Scalars

- ▶ Scalar variables are created like

$$a < -6$$

- ▶ Usual calculator commands apply: $+$, $-$, $\sin()$, $\log()$, $\exp()$, $\sqrt{}$, etc.
- ▶ To assign a missing value, enter

$$a < -NA$$

- ▶ To check if an observation is missing, enter

$$is.na(a)$$

- ▶ Rounding (2 is the number of decimal places, “;” separates commands)

$$> a < -pi; round(a, 2)$$
$$[1] 3.14$$

Data forms - vectors

Here are some vector commands and R output (“;” separates two commands).

```
> b <- -c(1, 6, 3); b
```

```
[1] 1 6 3
```

```
> b <- -1 : 7; b
```

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

```
> b <- -rep(1, 4); b
```

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

```
> b <- -seq(0, 1, length = 4); b
```

```
[1] 0.0000000 0.3333333 0.6666667 1.0000000
```

```
> b[2]; b[c(2, 4)]
```

```
[1] 0.3333333
```

```
[1] 0.3333333 1.0000000
```

```
> c <- matrix(1 : 6, 2, 3); c
     [, 1] [, 2] [, 3]
[1,]  1  3  5
[2,]  2  4  6
> c[2, 2]; c[2, 3]; dim(c)
[1] 4
[1] 6
[1] 2 3
```

- ▶ `> diag(3)` creates a 3×3 identity matrix.
- ▶ `> diag(1 : 3)` creates a 3×3 diagonal matrix with diagonal elements 1, 2, and 3.
- ▶ `> cbind(a, b)` creates a two-column matrix with vector *a* in

Data forms - arrays

An array is a matrix with more than two dimensions, e.g.,

```
> d <- array(1 : 12, c(2, 2, 3))
```

```
> d[2, 1, 2]
```

```
[1] 6
```

```
> dim(d)
```

```
[1] 2 2 3
```

Data forms - lists

A list is a collection of scalars, matrices, or arrays.

```
> a <- 1:3
> b <- "hi, my name is joe"
> l <- list(part1 = a, textpart = b)
> l$part1
[1] 1 2 3
> l$part1[3]
[1] 3
> l$textpart
[1] "hi, my name is joe"
```

> *names(l)* gives the names of the elements of list *l*.

Matrix manipulation

```
d <- -matrix(1 : 4, 2, 2); d
```

```
[1,] 1 3
```

```
[2,] 2 4
```

- ▶ Scalar multiplication, $> 2 * d$

```
[1,] 2 6
```

```
[2,] 4 8
```

- ▶ Operations are applied to each element, $> exp(d)$

```
[1,] 2.718282 20.08554
```

```
[2,] 7.389056 54.59815
```

Matrix manipulations

- ▶ Matrix addition (assuming $\dim(X1) = \dim(X2)$): `> X1 + X2`
- ▶ Matrix multiplication (assuming it's well-defined):
`> X1% * %X2`
- ▶ Element-by-element multiplication (assuming $\dim(X1) = \dim(X2)$): `> X1 * X2`
- ▶ Matrix inversion (assuming X is square): `> solve(X)`.
- ▶ Transpose: `> t(X)`.
- ▶ Spectral decomposition: `> eigen(X)`.

Probability distributions

- ▶ To generate the scalar $X \sim N(6, 2)$,

> X < -rnorm(1, 6, 2)

- ▶ To generate the vectors $X_1, \dots, X_{20} \stackrel{iid}{\sim} N(6, 2)$,

> X < -rnorm(20, 6, 2)

- ▶ To generate $X_{ij} \stackrel{iid}{\sim} N(6, 2)$, $i = 1, \dots, 10$, $j = 1, \dots, 20$,

> X < -matrix(rnorm(200, 6, 2), 10, 20)

- ▶ There are many other distributions: *rt()*, *rgamma()*, *rexp()*, etc.

- ▶ *dnorm*, *pnorm*, *qnorm* give the density, distribution, and quantile function, respectively, for Gaussian variables.

Linear regression using the “lm” function

- ▶ R also has a function to perform linear regression,

$$fit <- lm(y \sim x1 + x2)$$

- ▶ *fit* is a list with several outputs from the linear regression.
- ▶ `> names(fit)` lists the objects produced by `lm`.
- ▶ `> plot(fit)` gives some diagnostics
- ▶ `fit$residuals` is a vector of residuals
- ▶ `> summary(fit)` gives the results

	Estimate	Std Error	t value	Pr(> t)
Intercept	0.9212	0.2216	4.157	6.96e-05
x1	1.9647	0.2113	9.299	4.38e-15
x2	-1.8709	0.2131	-8.779	5.78e-14

Importing/exporting data

- ▶ The function `read.table()` loads tab delimited files.
- ▶ I prefer `read.csv()` which loads comma delimited files.
- ▶ For example, if “datafile.csv” has 101 rows and 10 columns, and the first row consists of variable names,

```
> data <- read.csv("K : /datafile.csv", header = TRUE)
> dim(data)
[1] 100 10
```

- ▶ Export data using `write.table()` or `write.csv()`.

Summarizing data

- ▶ `> mean(x)` gives the mean of the object (vector or matrix) x .
- ▶ There are many other functions: `median()`, `sd()`, `quantile()`, etc.
- ▶ `> table(x)` gives the frequency of each unique value of x , similar to `proc freq` in SAS.
- ▶ If x is a matrix, `mean(x[, 1])` gives the mean of the first column and `mean(x[1,])` gives the mean of the first row.
- ▶ If x is a matrix, `apply(x, 2, mean)` produces a vector of column means and `apply(x, 1, mean)` produces a vector of row means (mean can be replaced with `sd`, `median`,...).

Plotting functions

- ▶ `> plot(x, y)` makes a scatterplot of x and y
- ▶ `lines` and `points` add lines and points to a pre-existing plot
- ▶ `> boxplot()` makes a boxplot.
- ▶ `> hist()` makes a histogram.
- ▶ `> contour()` makes a contour plots.

Loops

For-loop

- ▶ $x < -rep(0, 5)$
 - ▶ $for(j \text{ in } 1 : 5)\{$
 - ▶ $x[j] < -j + 2$
 - ▶ $\}$
 - ▶ x
- ```
[1] 3 4 5 6 7
```

## while-loop

- ▶  $count < -0$
  - ▶  $while(count < 5)\{$
  - ▶  $print(count)$
  - ▶  $count < -count + 2$
  - ▶  $\}$
- ```
[1] 0 2 4
```

Loops are slow!

Compare the run time with and with the loop

- ▶ $n < -1000000$
- ▶ $y < -rep(0, n)$
- ▶ $for(i in 1 : n)\{y[i] < -rnorm(1)\}$
- ▶ $y < -rnorm(n)$ is much faster
- ▶ Avoid loops whenever possible!

The ifelse command

Syntax: `ifelse(comparison, result if true, result if false)`

- ▶ `ifelse(2 < 5, 1, 0)`
[1] 1
- ▶ `> ifelse(1 : 5 == 4, 1, 0)` (“`x==y`” means “`x equals y`”)
[1] 0 0 0 1 0
- ▶ `> ifelse(1 : 5 != 5, 6 : 10, 99)` (“`!=`” means “not equal to”)
[1] 6 7 8 9 99
- ▶ `x < -c(1, 2, NA); ifelse(is.na(x), 0, x)`
[1] 1 2 0
- ▶ `TF < -c(T, T, F); ifelse(TF, 1, 0)`
[1] 1 1 0

if/else statements

syntax: `if(condition){do if true} else{do if false}`.

Example 1:

- ▶ `x < -2`
- ▶ `if(x == 2){print("x is two")}`
- ▶ `else{print("x is not two")}`

Example 2:

- ▶ `x < -rnorm(2)`
- ▶ `if(x[1] > x[2]){`
- ▶ `x < -x[2 : 1]`
- ▶ `print("x is now sorted")`
- ▶ `}`

Creating your own functions

```
syntax: name <- function(inputs){  
  code...  
  output}
```

Example: This function computes $x^+ = \max(0, x)$

- ▶ `pospart <- function(x){`
- ▶ `xplus <- ifelse(x > 0, x, 0)`
- ▶ `xplus}`
- ▶ `y <- seq(-5, 5, length = 11)`
- ▶ `y`
- ▶ `[1] -5 -4 -3 -2 -1 0 1 2 3 4 5`
- ▶ `pospart(y)`
- ▶ `[1] 0 0 0 0 0 0 1 2 3 4 5`

Another example

This function sets observations equal to “symbol” (default is -99) to NA.

- ▶ `clean <- function(data, symbol = -99){`
- ▶ `cleaned <- ifelse(data == symbol, NA, data)`
- ▶ `cleaned}`
- ▶ `x <- c(10, 0, -99)`

- ▶ `clean(x)`
[1] 10 0 NA
- ▶ `clean(x, symbol = -99)`
[1] 10 0 NA
- ▶ `clean(x, symbol = 0)`
[1] 10 NA -99

Packages

- ▶ There are two kinds of packages
- ▶ Base packages: the codes for these packages are automatically downloaded to your PC when you download R.
- ▶ Contributed packages: These must be downloaded from CRAN before they can be used. Click Packages/Install Package(s) and choose the package from a list.
- ▶ You must tell R to compile the code before using the functions
- ▶ To do this click Package/Load package or enter `library("package name")` at the command prompt.

For example, to perform Cox's proportional hazard's model,

- ▶ `library("survival")`
- ▶ `?coxph`