Statistics with R

Introduction and Examples

Deepayan Sarkar

University of Wisconsin – Madison

Summer Institute for Training in Biostatistics (2005)

Our goal for the first two weeks

. . . is basically to get comfortable using R. We will learn

- to do some elementary statistics
- to use the documentation / help system
- about the language basics
- about data manipulation

We will learn about other specialized tools later when they are required.

What is R

- From the text: *R provides an environment in which you can perform statistical analysis and produce graphics. It is actually a complete programming language, although that is only marginally described in this book.*
- We will mostly use R as a toolbox for standard statistical techniques.
- Some knowledge of R programming essential to use it well.

More information available at the R Project homepage:
`http://www.r-project.org`

Interacting with R

R usually works interactively, using a question-and-answer model:

- Start R
- Type a command and press `Enter`
- R executes this command (often printing the result)
- R then waits for more input
- Type `q()` to exit

Simple Examples

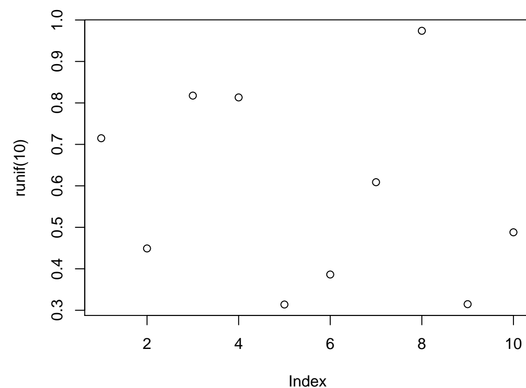
```
> 2 + 2
[1] 4
> exp(-2)
[1] 0.1353353
> log(100, base = 10)
[1] 2
> runif(10)
 [1] 0.46164277 0.84000664 0.34755931 0.77642092 0.37276779
 [6] 0.23124987 0.08789971 0.60120298 0.38273254 0.16632850
```

The last command generates 10 random numbers between 0 and 1. The printed result is a vector of 10 numbers. The bracketed numbers at the beginning of each line indicate the index of the first number on that line.

`exp`, `log` and `runif` are functions, indicated by the presence of parentheses. Most useful things in R are done by functions.

Simple Examples: plotting

```
> plot(runif(10))
```



Variables

Like most programming languages, R has symbolic variables which can be assigned values. The traditional way to do this in R is the `'<-'` operator. (The more common `'='` also works.)

```
> x <- 2
> yVar2 = x + 3
> s <- "this is a character string"
> x
[1] 2
> yVar2
[1] 5
> s
[1] "this is a character string"
> x + x
[1] 4
> x^yVar2
[1] 32
```

Variables

Possible variables names are very flexible. However, note that

- variable names cannot start with a digit
- names are case-sensitive
- some common names are already used by R, e.g., `c, q, t, C, D, F, I, T,` and should be avoided

Vectorized arithmetic

- The elementary data types in R are all vectors
- The `c(...)` construct can be used to create vectors:

```
> weight <- c(60, 72, 57, 90, 95, 72)
> weight
[1] 60 72 57 90 95 72
```

- To generate a vector of regularly spaced numbers, use

```
> seq(0, 1, length = 11)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
```

- Common arithmetic operations (including `+, -, *, /, ^`) and mathematical functions (e.g. `sin, cos, log`) work element-wise on vectors, and produce another vector:

Scalars from Vectors

Many functions summarize a data vector by producing a scalar from a vector. For example

```
> sum(weight)
[1] 446
> length(weight)
[1] 6
> avg.weight <- mean(weight)
> avg.weight
[1] 74.33333
```

Vectorized arithmetic

```
> height <- c(1.75, 1.8, 1.65, 1.9, 1.74, 1.91)
> height^2
[1] 3.0625 3.2400 2.7225 3.6100 3.0276 3.6481
> bmi <- weight/height^2
> bmi
[1] 19.59184 22.22222 20.93664 24.93075 31.37799 19.73630
> log(bmi)
[1] 2.975113 3.101093 3.041501 3.216102 3.446107 2.982460
```

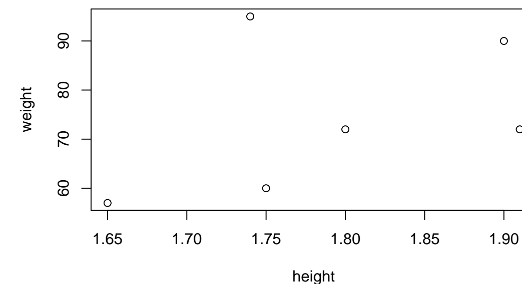
When two vectors are not of equal length, the shorter one is recycled. The following adds 0 to all the odd elements and 2 to all the even elements of `1:10`:

```
> 1:10 + c(0, 2)
 [1]  1  4  3  6  5  8  7 10  9 12
```

Graphics

The simplest way to produce R graphics output is to use the `plot` function:

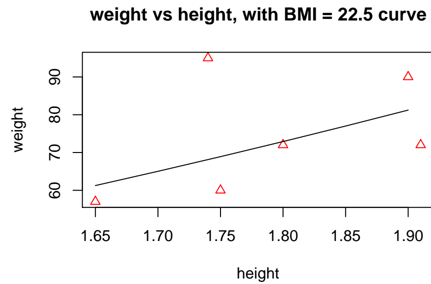
```
> plot(x = height, y = weight)
```



Graphics

There are many options that can control the details of what the plot looks like. Once created, plots can also be enhanced:

```
> plot(x = height, y = weight, pch = 2, col = "red")
> hh <- c(1.65, 1.7, 1.75, 1.8, 1.85, 1.9)
> lines(x = hh, y = 22.5 * hh^2)
> title(main = "weight vs height, with BMI = 22.5 curve")
```



Descriptive Statistics

Simple summary statistics: mean, median, standard deviation, variance

```
> x <- rnorm(100)
> mean(x)
[1] -0.2881313
> sd(x)
[1] 0.983495
> var(x)
[1] 0.9672625
> median(x)
[1] -0.3380632
```

Descriptive Statistics (contd)

Simple summary statistics: quantiles, inter-quartile range

```
> xquants <- quantile(x)
> xquants
        0%        25%        50%        75%       100%
-2.3011444 -0.9792947 -0.3380632  0.4119879  2.8260118
> xquants[4] - xquants[2]
     75%
1.391283
> IQR(x)
[1] 1.391283
> quantile(x, probs = seq(0, 1, length = 11))
         0%         10%         20%         30%         40%
-2.30114444 -1.55916781 -1.12134696 -0.77464565 -0.61983638
        50%         60%         70%         80%         90%
-0.33806323 -0.01043648  0.16151194  0.45262367  0.82814147
       100%
 2.82601184
```

The `summary` function

When applied to a numeric vector, `summary` produces a nice summary display:

```
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-2.3010 -0.9793 -0.3381 -0.2881  0.4120  2.8260
```

The output of `summary` can be different when applied to other objects.

The Iris Data Set

Let's look at a real example: The Iris data set is one of many already available in R (type `data()` for a full list).

```
> str(iris)
'data.frame':        150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1
```

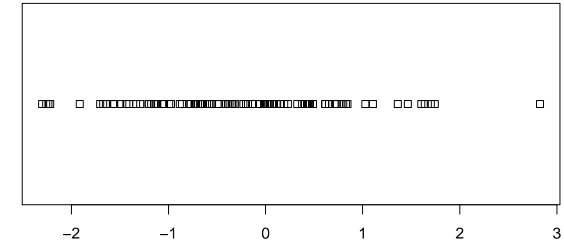
This data set contains measurements on 150 flowers, 50 each from 3 species: *Iris setosa*, *versicolor* and *virginica*.

It is typically used to illustrate the problem of classification—given the four measurements for a new flower, can we predict its Species?

Graphical display: Strip Plots

The simplest plot of numeric data is a strip plot (often called a dot plot)

```
> stripchart(x)
```



The summary function revisited

```
> summary(iris)
  Sepal.Length    Sepal.Width     Petal.Length
 Min.   :4.300   Min.   :2.000   Min.   :1.000
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600
 Median :5.800   Median :3.000   Median :4.350
 Mean   :5.843   Mean   :3.057   Mean   :3.758
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100
 Max.   :7.900   Max.   :4.400   Max.   :6.900
  Petal.Width          Species
 Min.   :0.100   setosa    :50
 1st Qu.:0.300   versicolor:50
 Median :1.300   virginica :50
 Mean   :1.199
 3rd Qu.:1.800
 Max.   :2.500
```

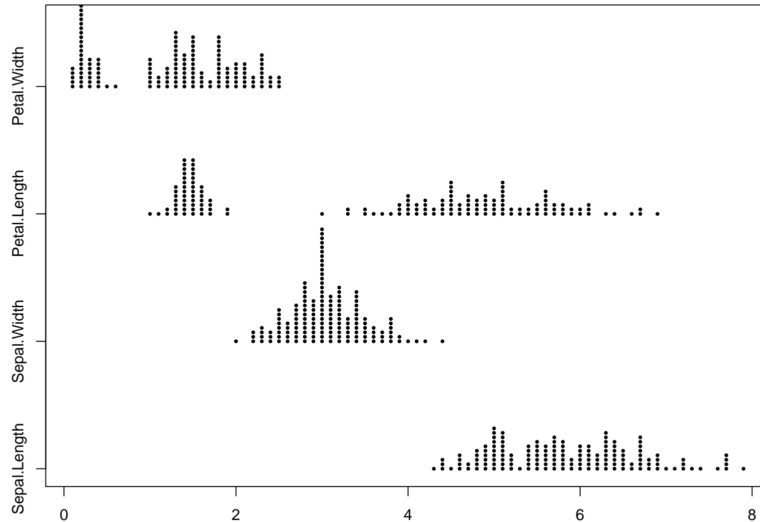
Note the different format of the output. `Species` is summarized differently because it is a categorical variable (more commonly called factor in R).

Strip Plots for the Iris data

We can produce a similar plot with the Iris data. Unfortunately, it's not possible to indicate which points came from which Species.

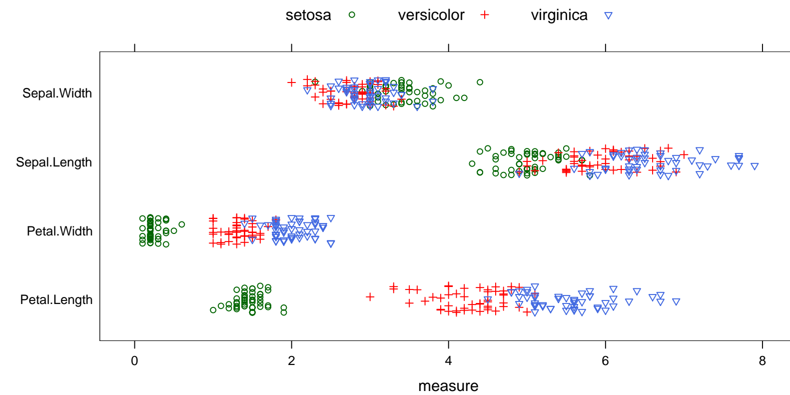
```
> stripchart(iris[, 1:4], method = "stack", pch = 16,
+     cex = 0.4, offset = 0.6)
```

Strip Plots for the Iris data



Grouped Display

```
> library(package = "lattice")
> stripplot(type ~ measure, data = iris2, groups = Species,
+     jitter = TRUE, auto.key = list(columns = 3))
```



Grouped Display

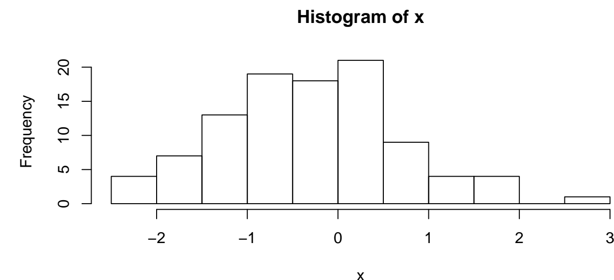
A more suitable plotting function is available in an add-on package called lattice, but that needs the data to be in a slightly different structure:

```
> iris2 <- reshape(iris, varying = list(names(iris)[1:4]),
+     v.names = "measure", timevar = "type", times = names(iris)[1:4],
+     direction = "long")
> str(iris2, give.attr = FALSE)
'data.frame':       600 obs. of  4 variables:
 $ Species: Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ type   : chr  "Sepal.Length" "Sepal.Length" "Sepal.Length" "Sepal.Length" ...
 $ measure: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ id     : int  1 2 3 4 5 6 7 8 9 10 ...
```

Histograms

Strip plots are pretty much useless for large data sets. The most popular (mostly because they are easy to draw by hand) graphical summary for numeric data is the histogram

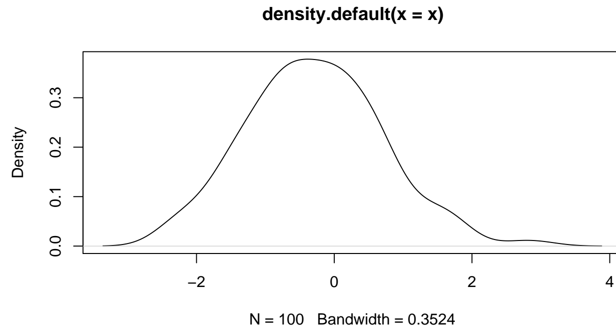
```
> hist(x)
```



Density Plots

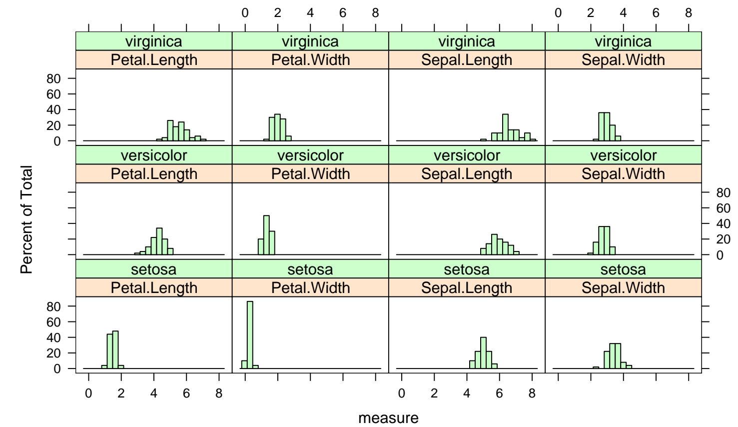
Density plots are generalized histograms

```
> plot(density(x))
```



Grouped Display (Histogram)

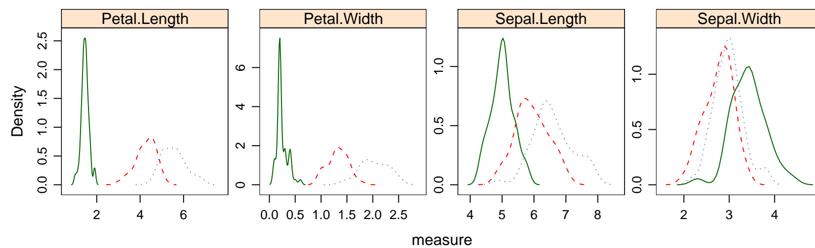
```
> histogram(~measure | type * Species, iris2, nint = 25)
```



Grouped Display (Density Plot)

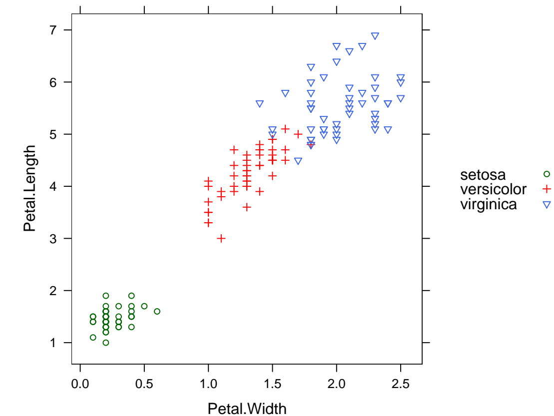
Again, for grouped data, the analogous `lattice` functions are more suitable.

```
> densityplot(~measure | type, data = iris2, groups = Species,
+     scales = "free", plot.points = FALSE)
```



Grouped Scatter Plot

```
> xyplot(Petal.Length ~ Petal.Width, iris, groups = Species,
+     aspect = 1, auto.key = list(side = "right"))
```



Categorical Data

We have already seen one example:

```
> summary(iris$Species)
    setosa versicolor  virginica
        50         50         50
```

Let's try to predict the Species using other measurements.

- What's the best measure to use?
- What are good thresholds?

Tables

Association between categorical variables can be summarized by tables:

```
> PL.tab <- xtabs(~PL.disc + Species, iris)
> SL.tab <- with(iris, table(SL.disc, Species))
> PL.tab
            Species
PL.disc       setosa versicolor virginica
  (0,2.5]         50          0         0
  (2.5,4.75]       0         44         1
  (4.75,7]         0          6        49
> SL.tab
            Species
SL.disc       setosa versicolor virginica
  (0,5.5]         47         11         1
  (5.5,6.25]       3         25        12
  (6.25,8]         0         14        37
```

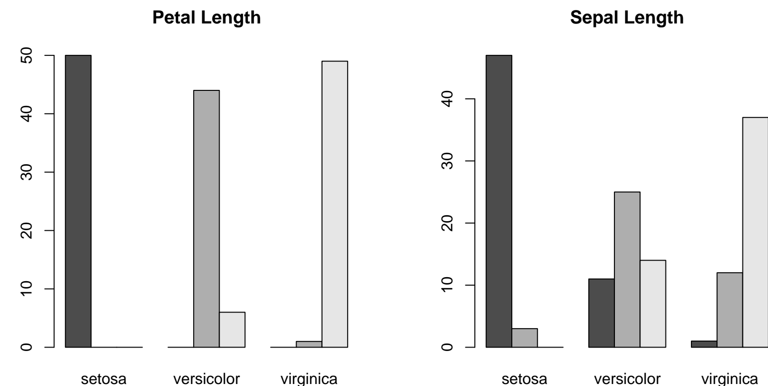
Discretizing

A continuous measure can be converted into a categorical one using the `cut` function:

```
> iris$PL.disc <- cut(iris$Petal.Length, breaks = c(0,
+     2.5, 4.75, 7))
> iris$SL.disc <- cut(iris$Sepal.Length, breaks = c(0,
+     5.5, 6.25, 8))
> str(iris)
`data.frame':       150 obs. of  7 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1
 $ PL.disc     : Factor w/ 3 levels "(0,2.5]","(2.5,4.75]",..: 1 1 1 1 1 1 1 1 1 1
 $ SL.disc     : Factor w/ 3 levels "(0,5.5]","(5.5,6.25]",..: 1 1 1 1 1 1 1 1 1 1
```

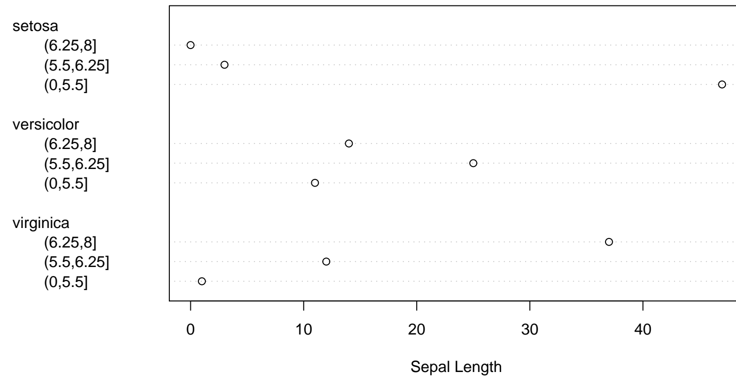
Graphical Display of Tables: Bar chart

```
> par(mfrow = c(1, 2))
> barplot(PL.tab, beside = TRUE, main = "Petal Length")
> barplot(SL.tab, beside = TRUE, main = "Sepal Length")
```



Graphical Display of Tables: Dot chart

```
> dotchart(SL.tab, xlab = "Sepal Length")
```



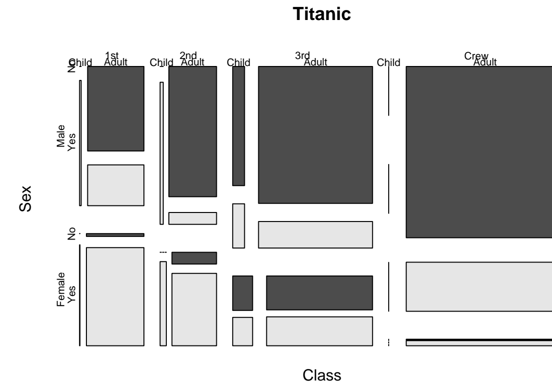
Higher Dimensional Tables

The built-in `Titanic` data set is a cross-tabulation of 4 characteristics of 2201 passengers on the Titanic

```
> dimnames(Titanic)
$Class
[1] "1st"  "2nd"  "3rd"  "Crew"

$Sex
[1] "Male"   "Female"

$Age
[1] "Child" "Adult"

$Survived
[1] "No"  "Yes"
```

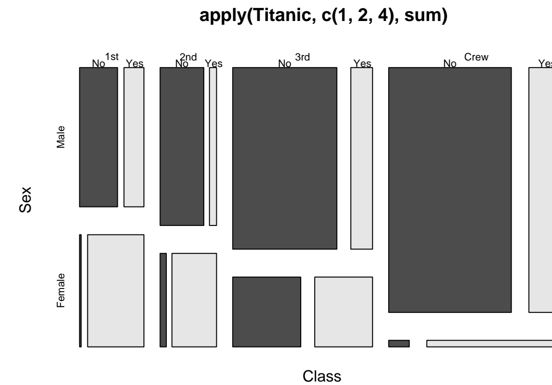
Titanic Survivors

```
> mosaicplot(Titanic, color = TRUE)
```



Titanic Survivors (simplified)

```
> mosaicplot(apply(Titanic, c(1, 2, 4), sum), color = TRUE)
```



Getting help

R has too many tools for anyone to remember them all, so it is very important to know how to find relevant information using the help system.

- `help.start()`
  Starts a browser window with an HTML help interface. One of the best ways to get started. Has links to a very detailed manual for beginners called 'An Introduction to R', as well as topic-wise listings.

- `help(topic)`
  Displays the help page for a particular topic or function. Every R function has a help page.

- `help.search("search string")`
  Subject/keyword search

Getting help (contd)

The help pages can be opened in a browser as well:

```
> help(plot, htmlhelp = TRUE)
```

The help pages are usually very detailed. Among other things, they often contain

- A 'See Also' section that lists related help pages
- A Description of what the function returns
- An 'Examples' section, with actual code illutrating how to use the documented functions. These examples can actually be run directly using the `example` function. e.g., try

```
> example(plot)
```

Getting help (contd)

The `help` function provides topic-wise help. When you know which function you are interested in, this is usually the best way to learn how to use it. There's also a short-cut for this; use a question mark (`?`) followed by the topic. The following are equivalent:

```
> help(plot)
> ?plot
```

When you want to know about a specific subject, but don't know which particular help page has the information, the `help.search` function is very useful. For example, try

```
> help.search("logarithm")
```

R packages

R makes use of a system of packages (*section 1.5.3*).

- Each package is a collection of routines with a common theme
- The core of R itself is a package called `base`
- A collection of packages is called a library
- Some packages are already loaded when R starts up. Other packages need be loaded using the `library()` function

R packages

Several packages come pre-installed with R.

```
> rownames(installed.packages())
 [1] "KernSmooth" "MASS"       "base"       "boot"
 [5] "class"      "cluster"    "datasets"   "foreign"
 [9] "grDevices"  "graphics"   "grid"       "lattice"
[13] "methods"    "mgcv"       "nlme"       "nnet"
[17] "rpart"      "spatial"    "splines"    "stats"
[21] "stats4"     "survival"   "tcltk"      "tools"
[25] "utils"
```

There are also many (more than 300) other packages contributed by various users of R available online, from the Comprehensive R Archive Network (CRAN):
http://cran.us.r-project.org/src/contrib/PACKAGES.html

The Bioconductor project provides an extensive collection of R packages specifically for bioinformatics
http://www.bioconductor.org/packages/bioc/stable/src/contrib/html/

R packages

It is fairly easy for anyone to write new R packages. This is one of the attractions of R over other statistical software.

Some packages are already loaded when R starts up. At any point, The list of currently loaded packages can be listed by the `search` function:

```
> search()
 [1] ".GlobalEnv"         "package:lattice"
 [3] "package:tools"      "package:methods"
 [5] "package:stats"      "package:graphics"
 [7] "package:grDevices"  "package:utils"
 [9] "package:datasets"   "Autoloads"
[11] "package:base"
```

R packages

Other packages can be loaded by the user. We will be interested in the `ISwR` package, which contains the datasets used in the text. This can be loaded by:

```
> library(ISwR)
```

New packages can be downloaded and installed using the `install.packages` function. For example, to install the `ISwR` package (if it's not already installed), one can use

```
> install.packages("ISwR")
> library(help = ISwR)
```

The last call gives a list of all help pages in the package.