

Introduction to R

Christopher J. Fonnesbeck

1 September 2011

Contents

Getting Up and Running	2
Running R	2
Getting help	3
Packages	4
Installing packages	5
Editing and running scripts	6
Frames and environments	7
Essentials of the R Language	8
Assignment	8
Built-in functions	8
Operators	9
Vectors	10
Logical expressions	11
Missing values	13
Functions	13
Matrices	14
Factors	17
Dates and times	19
Manipulating text	20
Looping and branching	21
Reading data from files	23
Complex Data Structures	25
Lists	25
Data Frames	27
Indexing	28
Modifying and creating variables	30
Subsetting	34
Attaching data frames	34
apply functions	35
Sorting	37
Saving data frames to files	38

Plotting	38
Saving plots	44
Trellis plots	45
Basic Statistical Analysis	52
Probability distributions	52
Two-sample t-Test	53
Checking Normality	54
Simple linear models	55
Model checking	59
Implementing logistic regression for binary outcomes	61
Survival analysis	64
Survival curves	65
Parametric survival models	68
The Cox model	70

Getting Up and Running

Running R

Upon starting the R console (or the RStudio application), you will see a console displaying several lines of information:

```
R version 2.13.0 (2011-04-13)
Copyright (C) 2011 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]
```

This includes information on your the version of the software, your platform, licensing, and a few handy commands to get you started. It also shows you how to quit the current R session `q()` and where your current workspace is located on your computer's file system. Then at the very bottom, you are presented with a command prompt:

```
>
```

It is at this command prompt that the user may interact with R, by typing expressions, then hitting return to have the expression evaluated. For example, you may simply use R as a calculator:

```
> 10 + 4
[1] 14
```

Expressions may be passed one-at-a-time to the prompt, or collectively in the form of a *script*. Scripts can be composed in any plain text editor (including the built-in editor in the upper left corner of RStudio). Consecutive expressions can be entered on the same line, if separated by semicolons:

```
> 2+3; 5*7; 3-7
[1] 5
[1] 35
[1] -4
```

Getting help

Since this course (or any R course, for that matter) is not comprehensive, it's a good idea to know where to go for help. Fortunately, R possesses a robust help system that is always only a click or command away. If you need additional information regarding any function, simply call its help file by typing a question mark (?) followed immediately by the function, then pressing enter. For example, take the standard deviation function `sd`:

```
> ?sd
```

```
sd                                package:stats                                R Documentation
```

```
Standard Deviation
```

```
Description:
```

```
This function computes the standard deviation of the values in
'x'. If 'na.rm' is 'TRUE' then missing values are removed before
computation proceeds. If 'x' is a matrix or a data frame, a
vector of the standard deviation of the columns is returned.
```

```
Usage:
```

```
sd(x, na.rm = FALSE)
```

```
Arguments:
```

```
x: a numeric vector, matrix or data frame. An object which is
not a vector, matrix or data frame is coerced (if possible)
by 'as.vector'.
```

```
na.rm: logical. Should missing values be removed?
```

Details:

```
Like 'var' this uses denominator n - 1.
```

```
The standard deviation of a zero-length vector (after removal of  
'NA's if 'na.rm = TRUE') is not defined and gives an error. The  
standard deviation of a length-one vector is 'NA'.
```

See Also:

```
'var' for its square, and 'mad', the most robust alternative.
```

Examples:

```
sd(1:2) ^ 2
```

(END)

As you can see, it describes this simple function in gory detail. Alternately, you can type `help(sd)` to obtain the same file. Additionally, you can pull up the entire help system in a browser by typing:

```
> help.start()
```

This will allow for browsing and searching through the help system, rather than pulling up the documentation for a particular function. In RStudio, the help system is always available in a tab within the lower righthand corner window.

Packages

Non-core R functionality is distributed in the form of discrete *packages*. This allows third parties to extend R, which allows users to access newer methods than are typically available in the base R application. Often, packages are written by the scientists who have themselves developed the statistical method.

If you type `sessionInfo()` into the console, you will discover which packages are currently loaded into your R session:

```
> sessionInfo()
R version 2.13.0 (2011-04-13)
Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)

locale:
[1] C/en_US.UTF-8/C/C/C/C

attached base packages:
[1] stats      graphics  grDevices  utils      datasets  methods   base

loaded via a namespace (and not attached):
[1] tools_2.13.0
```

There are three classes of package: *base*, *recommended*, and *other*. Base packages are installed with R and are loaded into the session automatically every time R is started; recommended packages ship with R, but need to be loaded manually as required; and other packages must be downloaded, installed and loaded before they can be used. To see the packages that are currently installed (but not necessarily loaded), use the function:

```
installed.packages()
```

This modularity makes R efficient relative to many commercial packages. Loading only packages that are relevant to your current work saves space, time and memory.

Recommended and other packages (once installed) are loaded via the `library` function. For example, to load the `ggplot2` library (used, as we will see, for generating plots):

```
> library(ggplot2)
Loading required package: reshape
Loading required package: plyr
```

```
Attaching package: 'reshape'
```

```
The following object(s) are masked from 'package:plyr':
```

```
rename, round_any
```

```
Loading required package: grid
Loading required package: proto
```

The output from the command shows that there were several *dependencies*, or other packages that `ggplot2` requires in order to function. If a particular package has no dependencies, there will be no message; if a package is either not installed or installed incorrectly, you will receive an error message.

Installing packages

The easiest and most common way to obtain and install third-party packages is through an online repository called the [Comprehensive R Archive Network](#), or CRAN. This network comprises several mirrored repositories that contain all available packages. Packages on CRAN have been vetted for appropriate format and documentation, and tend to contain the most up-to-date versions across all of its servers.

There are a number of ways to install packages from CRAN. For example, you can install from the command line when R is not running, which I will not cover here. A convenient method for installing a single package is to use the `install.packages` command from the R console:

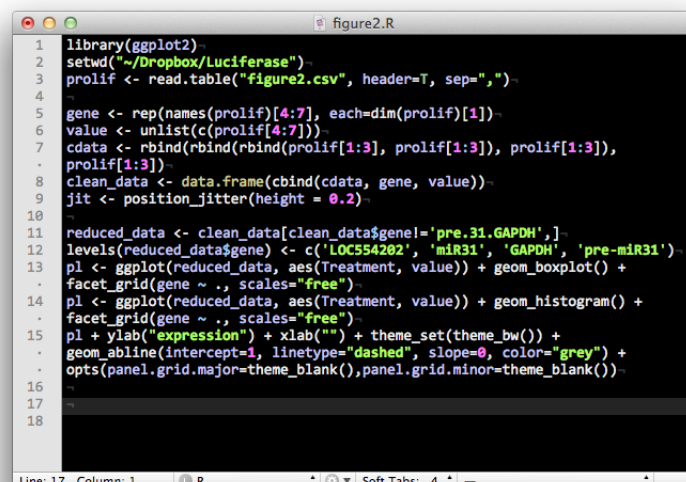
```
install.packages("ggplot2", dependencies=TRUE)
```

Including the `dependencies=TRUE` flag also installs any other packages that the target package requires to run on your machine. By default, R will install packages to your system's default location (which varies according to operating system), but this too can be changed by specifying `destdir="new/install/directory"` in the `install.packages` command.

An even easier way to download and install from CRAN is to run the graphical package tool that comes with Mac OS X and Windows versions of R. This is most useful for searching and browsing the available packages before you install, and for installing several packages in one session. You can often discover new packages related to the one you are interested in using! It also allows you to choose from among the dozens of CRAN servers to allow for the most efficient use of the network.

Editing and running scripts

As noted above, it is generally more efficient to perform statistical analyses in R by generating a *script* that runs a sequence of commands in succession, rather than interactively typing commands into the R console. Composing scripts is straightforward; all that is required is a text editor. There are general text editors available on all computing platforms, such as TextEdit on Mac OS X and Notepad on Windows, as well as more specialized editors, including TextMate on Mac OSX (see image below), Kate on Linux and Tinn-R for Windows that provide niceties such as colored syntax highlighting, an embedded R console and keyboard shortcuts for automating programming tasks. RStudio includes an editor (in the top-left corner of the application) that allows code to be sent directly to the R console.



```

1 library(ggplot2)
2 setwd("~/Dropbox/Luciferase")
3 prolific <- read.table("figure2.csv", header=T, sep=",")
4
5 gene <- rep(names(prolific)[4:7], each=dim(prolific)[1])
6 value <- unlist(c(prolific[4:7]))
7 cdata <- rbind(rbind(rbind(prolific[1:3], prolific[1:3]), prolific[1:3]),
8               prolific[1:3])
9 clean_data <- data.frame(cbind(cdata, gene, value))
10 jit <- position_jitter(height = 0.2)
11
12 reduced_data <- clean_data[clean_data$gene != 'pre.31.GAPDH',]
13 levels(reduced_data$gene) <- c('LOC554202', 'miR31', 'GAPDH', 'pre-miR31')
14 p1 <- ggplot(reduced_data, aes(Treatment, value)) + geom_boxplot() +
15   facet_grid(gene ~ ., scales='free')
16 p2 <- ggplot(reduced_data, aes(Treatment, value)) + geom_histogram() +
17   facet_grid(gene ~ ., scales='free')
18 p1 + ylab("expression") + xlab("") + theme_set(theme_bw()) +
19   geom_abline(intercept=1, linetype="dashed", slope=0, color="grey") +
20   opts(panel.grid.major=theme_blank(), panel.grid.minor=theme_blank())

```

Once you have created a script, it should be saved to your computer's hard disk before it is run in R. You should save the file to the current working directory that is being used by R. You can persuade R to give you this information by using the `getwd` (abbreviation for "GET Working Directory") command:

```

> getwd()
[1] "/Users/fonnescoj"

```

Alternately, you can save the file wherever you prefer, then tell R to switch its working directory using the `setwd` function. You can keep your scripts in a directory that is separate from R's working directory, but you will then have to provide R with this information each time you run the script.

Though it is not strictly necessary, it is wise to append the *.r* (or *.R*) suffix to the end of the filename. This will allow many text editors and other programs (such as RStudio) to recognize the file as an R script. To run any script, pass the filename to R's `source` function:

```
> source("quartiles.r")

1st Quartile: 2
Median:      4
3rd Quartile: 7
```

It is possible, of course, to simply cut-and-paste commands from a script into the R console. However, this is not recommended because if an error occurs for any of the pasted commands, the remaining code continues to run (if possible), since they are considered just a series of input commands. This can have unintended consequences for any output that you might be expecting.

There are additional reasons for saving your code to a file and executing them as a script: It makes modifying and re-running your code much easier and allows you to share, distribute and backup your work.

Frames and environments

R uses abstractions called *frames* and *environments* to help organize all the content that it manages (*i.e.* all of the objects created by the code that you run). The details of this management system is deeper than we are interested in going at this stage, but it is helpful to understand some of the inner workings.

Frames and environments can be thought of as hierarchical levels of organization within the R system. To use an analogy, if we think of variables and functions as named folders in a filing system, a frame is the catalog of folder names, that is, a collection of named objects. An environment contains this catalog, plus the name of an enclosing environment.

What does this mean? When you start an R session, a global environment is created, called `.GlobalEnv`. This is the default container for any objects, including other environments, that are created during the session. For example, loading a package creates its own environment that contains the variables, functions, data and other objects that it introduces. Try starting a new R session, and type the `search` function:

```
> search()
[1] ".GlobalEnv"          "package:stats"      "package:graphics"
[4] "package:grDevices"  "package:utils"      "package:datasets"
[7] "package:methods"    "Autoloads"          "package:base"
```

Recall that some packages were in the *base* set, meaning that they are loaded automatically when R is run. You can see here a listing of these base packages, including `stats`, `graphics`, and `datasets`. When you call a function, R creates a new environment, enclosed in the current environment. Objects (*e.g.* variables, data and other functions) created inside the function are not available in the enclosing environment. Objects available in the enclosing environment, on the other hand, are available to the function running within it.

Here is a quick example: Define a variable, say `x` and then a function that adds a value to `x`, and returns it:

```

> x <- 5
> my_function <- function() { x + 12 }
> my_function()
[1] 17

```

You can see that even though `x` is defined outside of the function, it is available to be used by expressions inside the function. On the other hand, consider another function:

```

> another_function <- function() {
+   y <- 5
+   y + 12
+ }
> another_function()
[1] 17
> y
Error: object 'y' not found

```

Now, we have defined a variable `y` inside of `another_function`, but this variable cannot be seen by the enclosing environment.

To see which objects occupy the frame of your current environment, you can use the `ls` command. This can get cluttered after continued use without restarting R, but you can manually remove all of the objects from the frame:

```

> rm(list=ls())
> ls()
character(0)

```

This indicates an empty frame. It's a good idea to clear your environment before switching projects, in case unwanted objects from your previous work accidentally get used by code in the next project that happen to use the same name for particular variables or functions.

Essentials of the R Language

Assignment

As you have seen from examples in the previous section, variables are given values using the assignment operator (`<-`), which looks like an arrow pointing from the value to the variable name. In recent versions of R, the equals sign can also be used, but this is discouraged by some.

Remember there is no space between the `<` and the `-`!

```

> x <- 5
> x < - 5
[1] FALSE

```

The latter expression tests whether `x` is less than `-5`.

The outcome of an assignment operation can be shown by enclosing the expression in parentheses:

```

> (x <- 5)
[1] 5

```


Built-in functions

Most commonly-used mathematical functions are available in the base R package:

```
> log(10)
[1] 2.302585
> exp(1)
[1] 2.718282
> log10(6)
[1] 0.7781513
> log(9,3)
[1] 2
```

The trigonometric functions in R measure angles in radians:

```
> pi
[1] 3.141593
> sin(pi/2)
[1] 1
> cos(pi/2)
[1] 6.123032e-017
```

Exponential notation is specified by e:

```
> 1.2e3
[1] 1200
> 1.2e-2
[1] 0.012
```

Modulo and integer division is performed to separate the integer and remainder portions of a division operation. Consider the quotient of 19 divided by 3:

```
> 19/3
[1] 6.333333
```

This is composed of an integer (6) and a fraction ($1/3=0.333333$). If we only wanted one or the other of these components, they can be obtained by the integer division (`%/%`) and modulo (`%%`) operators, respectively:

```
> 19 %/% 3
[1] 6
> 19 %% 3
[1] 1
```

Operators

R uses standard arithmetic operator symbols, including `+` `-` `*` `/` `%%` `^`. Similarly, relational operators are standard:

```
> 5 < 6
[1] TRUE
> 5 >= 6
[1] FALSE
> 5 != 6
```

```
[1] TRUE
> 5 == 6
[1] FALSE
```

R also allows for logical operators, including NOT (!), OR (|) and AND (&).

Vectors

Vectors are variables with (potentially) multiple values of the same type. Several scalar values can be used to compose a vector using the concatenation function (c):

```
y <- c(10, 11, 12, 13, 14, 15, 16)
```

A vector sequence can be generated automatically using the : operator between the upper and lower bounds of the sequence:

```
> y <- 10:16
> y
[1] 10 11 12 13 14 15 16
```

The sequence operator is a shorthand for the seq function, which allows for a step size to be specified as well:

```
> (y <- seq(1, 20, by=2))
[1] 1 3 5 7 9 11 13 15 17 19
```

A homogeneous vector of the same value can be generated using rep:

```
> rep(1, 5)
[1] 1 1 1 1 1
```

Its also possible to sequentially enter the values of a vector from the keyboard using scan:

```
> y <- scan()
1: 10
2: 11
3: 12
4: 13
5: 14
6: 15
7: 16
8:
Read 7 items
```

To access particular elements within a vector, bracket notation is used to retrieve elements corresponding to their order in the vector (index):

```
> y[5]
[1] 14
> y[-1]
[1] 11 12 13 14 15 16
> y[3:5]
[1] 12 13 14
```

In R, scalars are just a special case of vectors, with size equal to one:

```
> x <- 5
> x[1]
[1] 5
```

Its even possible to have a vector with no elements:

```
> x <- c()
> length(x)
[1] 0
```

Algebraic operations on vectors are performed element-wise:

```
> x <- c(4,5,6); y <- 1:3
> x+y
[1] 5 7 9
> x**2
[1] 16 25 36
```

If vectors being operated on are of unequal length, the shorter vector gets repeated until it becomes equal to the size of the larger vector. Since this is unusual behavior, and can result in silent (*i.e.* hard to detect) errors, R provides a warning message when this happens:

```
> x - c(7,8)
[1] -3 -3 -1
Warning message:
In x - c(7, 8) :
  longer object length is not a multiple of shorter object length
```

Note that this warning is not generated if the shorter vector is an exact multiple of the larger vector:

```
> y <- 1:10
> y/c(2,3)
[1] 0.5000000 0.6666667 1.5000000 1.3333333 2.5000000 2.0000000
     3.5000000 2.6666667 4.5000000
[10] 3.3333333
```

Many R functions are designed to operate over vectors:

```
> sqrt(y)
[1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
     2.828427 3.000000 3.162278
> mean(y)
[1] 5.5
> sum(y) / length(y)
[1] 5.5
> var(y)
[1] 9.166667
> sum((y - mean(y))^2) / (length(y) - 1)
[1] 9.166667
```

Logical expressions

We have seen relational operators above. The value yielded by expressions that use these operators is either `TRUE` or `FALSE` (or 1 and 0, respectively).

The pipe (`|`) operator returns `TRUE` if either the right or the left element is true, or both. If we wish to determine if either one or the other is true, but *not* both, the function `xor` should be used:

```
> (4>3) | (7>5)
[1] TRUE
> xor((4>3), (7>5))
[1] FALSE
```

In addition to accessing elements according to their index value, a *logical* (i.e. `TRUE` or `FALSE`) vector can be used, which extracts elements where the corresponding element in the logical vector is true:

```
> z <- y<13
> z
[1] TRUE TRUE TRUE FALSE FALSE FALSE FALSE
> y[z]
[1] 10 11 12
```

Alternatively, the `subset` function can be used to obtain the same result:

```
> subset(y, subset=y<13)
[1] 10 11 12
```

If we only want to know the *position* of the values of interest, then `which` is the appropriate function:

```
> which(y%%2 == 0)
[1] 1 3 5 7
```

Though it may not be immediately clear why they are useful, R also provides sequentially-evaluated relational operators. Just as `&` and `|` represent *and* and *or*, respectively, `&&` and `||` work similarly, except that they sometimes do not need to evaluate the second (right-hand) element. If the left-hand side of the expression is `FALSE`, the `&&` operator returns `FALSE` without evaluating the right-hand side; if the left-hand side of the expression is `TRUE`, the `||` operator returns `TRUE` without evaluating the right-hand side.

For example, consider the following expressions:

```
> x <- 0
> sin(1/x) == 0
[1] NaN
Warning message:
In sin(1/x) : NaNs produced
```

When `x` is zero, `1/x` is infinite, resulting in an error. We may want to use this expression without worrying about `x` being zero, however. Notice that `|` generates a warning, while `||` does not:

```

> (x == 0) | (sin(1/x) == 0)
[1] TRUE
Warning message:
In sin(1/x) : NaNs produced

> (x == 0) || (sin(1/x) == 0)
[1] TRUE

```

Missing values

Real data often, for one reason or another, contain missing values. Because R is designed for working with data, there is a missing data value (NA) to act as placeholders, so that they may be dealt with statistically, either by ignoring them or imputing them.

```

> x <- c(14, NA, 32)
> is.na(x)
[1] FALSE TRUE FALSE

```

These missing values can propagate, unless they are explicitly dealt with:

```

> mean(x)
[1] NA
> mean(x, na.rm=TRUE)
[1] 23

```

The NA object should not be confused with a similar object, NULL. While NA is a placeholder for something that exists, but is missing, NULL represents something that does not exist.

Functions

Consider a generic function in mathematics:

$$x = f(y, z)$$

Here, the function f takes arguments y and z , and returns an output value x . Functions in R work the same way; to execute a function and produce an output value, the function name is followed immediately by a set of parentheses, which contain the appropriate arguments. For example, the function `rep`, which generates a repeated sequence of values, is specified as follows:

```
rep(x, times=1, length.out=NA, each=1)
```

The first is a required argument, which is the value to be replicated, while the remaining arguments are optional, with the default value listed to be used if no value is specified by the user.

In general, functions can be programmed using the following syntax:

```

func_name <- function(argument_1, argument_2, ...) {
  expression_1
  expression_2
  ...
  return(output_value)
}

```

Arguments are *passed* to the function, and they become variables in its environment; the function then evaluates the expressions within the curly braces. If there is no return statement, the function returns the value of the last expression in the function. When there is no return value, a NULL is returned. Arguments may be presented as a key-value pair, that is `key=value`. In this case, the argument is optional, with the specified value passed as a default.

Here is a simple function that returns the factorial of an integer value:

```
fact <- function(n) {
  # Calculate the factorial
  n_fact <- prod(1:n)
  return(n_fact)
}
```

This function is used by passing an integer inside a set of parentheses following the name of the function:

```
> fact(5)
[1] 120
```

Notice that the variable `n_fact` is only available inside the `fact` environment:

```
> n_fact
Error: object 'n_fact' not found
```

You can use key-value arguments even when they are not specified that way in the source code. For example, consider the trivial division function:

```
div <- function(x,y) { x/y }
```

This works as expected:

```
> div(6,5)
[1] 1.2
```

However, we can assign `x` and `y` in reverse order if we specify their values:

```
> div(y=6, x=5)
[1] 0.8333333
```

Matrices

A matrix is little more than a vector with two dimensions, conventionally referred to as rows and columns. The easiest way to specify a matrix is to use the `matrix` function, passing it a one-dimensional vector that is reshaped according to the desired dimension:

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

alternately:

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

You can specify both dimensions, but given the size of the input vector and one dimension, the second dimension is determined. Notice that the matrix is filled column-wise by default. You can override this with the `byrow` argument:

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

If you specify the dimensions to be *larger* than the length of the input vector, R repeats the vector to fill the residual elements of the matrix:

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

This is useful, for example, when you need a matrix of zeros:

```
> matrix(0, ncol=3, nrow=3)
      [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
[3,]    0    0    0
```

Rows, columns and elements can be indexed from a matrix using the following notation:

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

Algebraic operators act element-wise on matrices:

```
> (B <- matrix(6:1, nrow=2))
      [,1] [,2] [,3]
[1,]    6    4    2
[2,]    5    3    1
> A * B
      [,1] [,2] [,3]
[1,]    6   12   10
[2,]   10   12    6
```

Matrix multiplication is via the `%*%` operator. Ensure that the number of columns in the first matrix equals the number of rows in the second matrix!:

```

> (B <- matrix(6:1, ncol=2))
      [,1] [,2]
[1,]     6     3
[2,]     5     2
[3,]     4     1
> A %*% B
      [,1] [,2]
[1,]    41    14
[2,]    56    20

```

All standard matrix operations are built-in, such as transpose (t):

```

> t(A)
      [,1] [,2]
[1,]     1     2
[2,]     3     4
[3,]     5     6

```

Many of the common matrix operations, however, require a square matrix as an argument:

```

> (C <- A %*% B)
      [,1] [,2]
[1,]    41    14
[2,]    56    20
> det(C) # Determinant
[1] 36
> eigen(C) # Eigenvalues and eigenvectors
$values
[1] 60.4040131  0.5959869

$vectors
      [,1] [,2]
[1,] 0.5851057 -0.3274028
[2,] 0.8109570  0.9448849
> solve(C) # Inverse
      [,1] [,2]
[1,] 0.5555556 -0.3888889
[2,] -1.5555556  1.1388889

```

Notice what happens when we multiply a matrix by its inverse:

```

> C %*% solve(C)
      [,1] [,2]
[1,]     1 1.776357e-15
[2,]     0 1.000000e+00

```

We would expect this to be a square matrix with ones in the diagonal positions and zeros on the off diagonal. It is *almost* there, with one of the off-diagonals being very, very small, but not exactly zero. This is the result of rounding error, which will always happen when we store floating-point numbers on a computer, since it uses a binary format.

Matrices can also be composed of vectors, using either the `cbind` (binding columns) or `rbind` (binding rows) functions:


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

Occasionally, data structures of more than two dimensions are required. These can be generated using the `array(data, dim)` function, which like `matrix` takes “raw” data as the first argument, but allows for a more general specification of array dimensions than just rows and columns:

```
> (D <- array(1:20, dim=c(2,5,2)))
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
> array(1:20, dim=c(2,5,2))
, , 1

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

, , 2

      [,1] [,2] [,3] [,4] [,5]
[1,]   11   13   15   17   19
[2,]   12   14   16   18   20
```

Conversely, an array can be queried for its dimensions:

```
> dim(D)
[1] 2 5 2
```

Factors

Variables used in statistics can be classified into one of three types: numeric, ordinal or categorical. Categorical variables are unique because they comprise some unordered set of values: yes, no; red, blue, yellow; black, white, Asian, hispanic, other. In R, categorical variables are called *factors* and the particular values are referred to as *levels*. This unique data type is used because categorical variables have to be treated very differently from numeric and ordinal variables in statistical models.

R needs to be explicitly told to treat a vector as a factor:

```
> x <- c("yes", "yes", "no", "yes", "no", "yes", "yes")
> class(x)
[1] "character"
> x <- factor(x)
> x
[1] yes yes no  yes no  yes yes
Levels: no yes
> class(x)
[1] "factor"
```

In fact, `factor` will attempt to create factors from any type of data, not just characters:

```
> y
[1] 2 4 4 5 2 3 1 3 2 2
> class(y)
[1] "numeric"
> y <- factor(y)
> y
[1] 2 4 4 5 2 3 1 3 2 2
Levels: 1 2 3 4 5
```

If there are more factor levels in principle than are contained with a particular dataset, the complete set can be specified with the `levels` function:

```
> levels(x) <- c("yes", "no", "maybe")
> x
[1] no no yes no yes no no
Levels: yes no maybe
> levels(y) <- 1:7
> y
[1] 2 4 4 5 2 3 1 3 2 2
Levels: 1 2 3 4 5 6 7
```

To visualize the frequency of each factor level, you can use `table`:

```
> table(y)
y
1 2 3 4 5 6 7
1 4 2 2 1 0 0
```

Ordinal variables can be represented by factors using the `ordered=TRUE` argument when coercing a vector into a factor:

```
> y[1]>y[2]
[1] NA
Warning message:
In Ops.factor(y[1], y[2]) : > not meaningful for factors
> (y <- factor(y, ordered=TRUE))
[1] 2 4 4 5 2 3 1 3 2 2
Levels: 1 < 2 < 3 < 4 < 5
> y[1]>y[2]
[1] FALSE
```

Levels can also be given more meaningful labels than their values:

```
> x <- c("y", "y", "n", "y", "n", "y", "y")
> (x <- factor(x, levels=c("y", "n", "m"),
+ labels=c("yes", "no", "maybe")))
[1] yes yes no yes no yes yes
Levels: yes no maybe
```

Dates and times

Another relevant data type frequently encountered in data -- particularly longitudinal data -- is the date or time quantities. Measurements of time are idiosyncratic, changing from month to month (*e.g.* the number of days), season to season (*e.g.* daylight savings time) and place to place (*e.g.* the ordering of month and day in USA versus Europe). R has a robust system for expressing and manipulating times and dates.

As an example, obtain the current time:

```
> (now <- Sys.time())  
[1] "2011-08-18 11:09:36 CDT"
```

At first glance, it would appear that R simply uses strings to store dates. However, upon closer inspection, we see this is not the case:

```
> mode(now)  
[1] "numeric"  
> class(now)  
[1] "POSIXct" "POSIXt"
```

In fact, the time and date are represented by a class of object called `POSIXct`. As the `mode` function reveals, this is actually a numeric data type, and is stored as the number of seconds since 1 January, 1970. We can see this if we try to convert the `POSIXct` object into an integer:

```
> as.integer(now)  
[1] 1313683776
```

Dates can also be expressed in as a list data structure by converting the `POSIXct` object to a `POSIXlt` object:

```
now2 <- as.POSIXlt(now)
```

This allows us to use keywords to extract particular parts of the date for use elsewhere:

```
> now2$year  
[1] 111  
> now2$min  
[1] 9  
> now2$wday  
[1] 4
```

The advantage of storing the time in a numeric format is that it makes calculations easy. It is possible in R to add and subtract times, as well as compare them using logical operators:

```
> now <- Sys.time()  
> now > then  
[1] TRUE  
> now - then  
Time difference of 26.32586 mins
```

Notice that the `POSIXct` object expresses the date in descending order, from the longest time scale (year) to the shortest (second). Often, we receive dates in a variety of formats, depending on the source of the data. In situations like this, the `strptime` function can be used to strip out the components of any date/time character string. Let's take a simple, but realistic, example of a set of dates in Microsoft Excel format:

```
dates <- c("27/02/2004", "27/02/2005", "14/01/2003", "28/06/2005",
           "01/01/1999")
```

The `strptime` function simply requires the format used by the `dates` vector to represent the date:

```
> (posix_dates <- strptime(dates, "%d/%m/%Y"))
[1] "2004-02-27" "2005-02-27" "2003-01-14" "2005-06-28" "1999-01-01"
> class(posix_dates)
[1] "POSIXlt" "POSIXt"
```

A complete listing of the codes for each component of dates and times is given in the help file for `strptime`.

Manipulating text

Aside from `numeric` and `logical` modes, there is also a `character` mode that is used to represent strings of characters. As with most other languages, characters are specified by enclosing them in either single (') or double (") quotes.

Strings can be concatenated using the `cat` or `paste` functions:

```
> x <- "normal"
> y <- "Poisson"
> z <- "binomial"
> (distributions <- paste(x, y, z, sep=", "))
[1] "normal, Poisson, binomial"
```

There are a number of special characters for text formatting that are denoted by a backslash (). These include:

- quotes: \"
- newline: \n
- tab: \t
- backspace: \b
- backslash: \\

Often, we want to manipulate text to display it nicely on the screen or in a file. Here is an example of how `cat` and `paste` can facilitate this:

```
> n <- 5
> x <- 7
> cat(paste(format(1:n, width = 8), format(x^(1:n), width = 10),
+ "\n"), sep = "")
1          7
```

2	49
3	343
4	2401
5	16807

Note that `cat` and `print` are superficially similar, but with different customization and conversion options.

Looping and branching

Effective statistical programming must allow for code to repeat and run different parts of the code to run conditional on the state of the model. R has a number of facilities for looping and branching that makes programming more efficient.

An `if` expression differentially executes code depending on conditions:

```
if (x < 0.5) {
  y <- 1
} else {
  y <- -1
}
```

More generally, there can be any number of conditions for a particular variable, linked together with `else if` clauses:

```
if (age < 1) {
  age_class <- "infant"
} else if (age < 13) {
  age_class <- "child"
} else if (age < 18) {
  age_class <- "adolescent"
} else age_class <- "adult"
```

Conditions may also be embedded within other conditions:

```
if (continuous == TRUE) {
  if (positive == TRUE) {
    distribution <- "gamma"
  } else {
    distribution <- "normal"
  }
} else {
  distribution <- "Poisson"
}
```

Operations that must be repeated a number of times can be embedded in a programming idiom called a *loop*. If the number of iterations is known ahead of time, the `for` loop is appropriate:

```
> (x_list <- seq(1, 9, by=2))
[1] 1 3 5 7 9
> sum_x <- 0
> for (x in x_list) {
+   sum_x <- sum_x + x
+ }
```

```

+     cat("Cumulative total: ", sum_x, "\n")
+ }
Cumulative total: 1
Cumulative total: 4
Cumulative total: 9
Cumulative total: 16
Cumulative total: 25

```

The `for` loop iterates over the vector in `x_list`, changing the variable `x` to the next item in the sequence at each iteration. When all elements have been exhausted, the loop exits and the code continues to the next statement outside the brackets.

A number of factors affect the speed of loops. One is preallocation. Compare the speed of this code:

```

n <- 1000000
x <- rep(0, n)
for (i in 1:n) {
  x[i] <- i
}

```

to that of this version:

```

n <- 1000000
x <- 1
for (i in 2:n) {
  x[i] <- i
}

```

The first is faster because the vector is preallocated, instead of having to grow the vector at each step.

If we do not know ahead of time how many iterations are required, a `while` loop is more appropriate:

```

> x <- 5
> while (x>0) {
+   x <- x - abs(rnorm(1))
+   cat("x=", x, "\n")
+ }
x= 4.856406
x= 4.646546
x= 4.505288
x= 3.774414
x= 2.560863
x= 2.528701
x= 1.95953
x= 1.765434
x= 1.650093
x= -0.1577281

```

The logical expression is evaluated, and if it evaluates to `TRUE`, the expressions in the brackets are executed. It loops until the expression is evaluated to be `FALSE`.

A more realistic example is the calculation of the time to loan repayment:

```

# Inputs
r <- 0.11                # Annual interest rate
period <- 1/12            # Time between repayments (in years)
debt_initial <- 1000     # Amount borrowed
repayments <- 12         # Amount repaid each period

# Calculations
time <- 0
debt <- debt_initial
while (debt > 0) {
  time <- time + period
  debt <- debt*(1 + r*period) - repayments
}

# Output
cat('Loan will be repaid in', time, 'years\n')

```

Note that you can create infinite loops:

```

while (TRUE) {
  # some expressions
}

```

Its also possible, though not recommended, to replicate a `for` loop using a `while` loop:

```

> y <- rnorm(10)
> y_sum <- 0
> x <- 1
> while (x<10) {
+   y_sum <- y_sum + y[x]
+   x <- x + 1
+ }
> y_sum
[1] 3.145644

```

While loops are useful, in general, it is better (faster, more efficient) to apply vectorized functions where they are available, rather than looping.

Reading data from files

In order to input data, we typically import from some type of file. The most straightforward format to read into R is a plain text file. Even plain text data tend to have some sort of structure, and the type of structure will dictate how the file is imported. The functions used to import data in R generally take a filename as an argument (along with other options), and return some data structure containing the data from the file.

The `scan` function is used to read vectors from a file, returning a vector object. Recall that in the [Vectors](#) section, we used `scan` to read values input from the keyboard.

There are a number of simple options for the user to specify:

```

scan(file = "", what = 0, n = -1, sep = "", skip = 0)

```

Arguments: - file: The file name to read from - what: mode of data to read - n: number of elements to read - sep: value-separating character - skip: number of lines at the beginning of the file to skip

Here is an example of using `scan` to import a very simple data file:

```
# Read from file
data <- scan(file = file_name)

# Calculations
n <- length(data)
data.sort <- sort(data)
data.1qrt <- data.sort[ceiling(n/4)]
data.med <- data.sort[ceiling(n/2)]
data.3qrt <- data.sort[ceiling(3*n/4)]

# Output
cat("1st Quartile:", data.1qrt, "\n")
cat("Median: ", data.med, "\n")
cat("3rd Quartile:", data.3qrt, "\n")
```

For an even more general way of importing data from a file, we can use the `readLines` function. This simply reads a file, one line at a time, returning a vector containing a character string for each line in the file. If the structure of the data file is too complicated for one of R's import facilities, it is often best to read the data in as strings, and manipulate them once imported.

```
> readLines("sample_data.csv")
```

Often, we receive data in the form of a table; that is, a rectangular grid of data. Tables often contain many different data types and hence it would be convenient to have a way of reading such input that will automatically distinguish among the types of columns within the table. As a result, `read.table` is probably the most convenient way to read tables of data into R.

`read.table` has a number of optional arguments, but the important ones are as follows:

```
read.table(file, header = FALSE, sep = ",", quote = "\"'\"",
           dec = ".", row.names, col.names,
           as.is = !stringsAsFactors,
           na.strings = "NA")
```

Argument definitions:

- file: name of file to import
- header: flag to indicate if the first line is a set of column headers
- sep: character used to separate values in each row
- quote: if character fields are delimited with quotes, specify the type of quotes
- row.names: optional vector of row names, such as observation IDs

- `column.names`: optional vector of column names; if header exists, it will use this for column names
- `as.is`: vector of column numbers to override the default variable of converting characters to factors
- `na.strings`: vector of strings that should be interpreted as NA values (i.e. missing data)

One of the commonest text input formats for data is the comma-separated values (CSV) format. These will typically have a header, as well as using a comma for the value delimiter. Here is a sample CSV data file:

```
"plot", "tree", "species", "dbh.cm", "height.m"
2, 1, "DF", 39, 20.5
2, 2, "WL", 48, 33
3, 2, "GF", 52, 30
3, 5, "WC", 36, 20.7
3, 8, "WC", 38, 22.5
4, 1, "WC", 46, 18
4, 2, "DF", 25, 17
```

In this case, it is convenient to use the `read.csv` function that assumes comma-separated values and a header.

`read.table` (and `read.csv`) produce the data in another data structure, called a data frame.

Complex Data Structures

Lists

A limitation of the `vector` data structure is that all of its elements must be the same type (or *mode*, in R parlance). You cannot mix numeric, character and logical types, for example. Try it yourself:

```
> c(4.5, 6)
[1] 4.5 6.0
> c(4.5, 6, "foo")
[1] "4.5" "6"   "foo"
```

In the first example, building a vector consisting of a floating-point number and an integer causes the integer to be cast into a float; in the second, including a character-valued element caused the other elements to become characters as well. As a rule, R selects the most general mode, and coerces all the elements to that mode. You can represent the integer's value as a float, but not the other way around; you can represent a numeric quantity as a string, but there is no numeric representation for a word.

The facility for storing multiple elements of different types in the same data structure in R is called a `list`. A list is an indexed set of objects, just like a vector, but its elements can be of various types -- including vectors and lists themselves. Here is a trivial example:

```

> my_list <- list("blue", FALSE, 1:7, 4.5534)
> my_list
[[1]]
[1] "blue"

[[2]]
[1] FALSE

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

[[4]]
[1] 4.5534

```

There are two ways of indexing list elements: if you want the result as another list, single square brackets are used:

```

> (sublist <- my_list[1:2])
[[1]]
[1] "blue"

[[2]]
[1] FALSE

> mode(sublist)
[1] "list"

```

Whereas, if the list element itself is wanted, double square brackets are used. Keep in mind that these can only be extracted one-at-a-time:

```

> my_list[[4]]
[1] 4.5534
> mode(my_list[[4]])
[1] "numeric"
> my_list[[3]][2]
[1] 2

```

List elements can also be indexed by keyword, if they are provided:

```

> (my_list <- list(A="blue", B=FALSE, C=1:7, D=4.5534))
$A
[1] "blue"

$B
[1] FALSE

$C
[1] 1 2 3 4 5 6 7

$D
[1] 4.5534

```

```
> my_list$D
[1] 4.5534
```

Notice that we use keyword indices by appending the keyword to a `$` following the list name. This syntax is common throughout R data structures whenever items are indexed by a keyword.

Lists can be “flattened” into a vector using the `unlist` function. Remember, of course, that since vectors must only contain elements of the same mode, elements will be converted to the most general mode:

```
> unlist(my_list)
      A      B      C1      C2      C3      C4      C5      C6
"blue" "FALSE" "1"    "2"    "3"    "4"    "5"    "6"
```

Notice that there is now only a one-dimensional structure, as the elements of `my_list$C` have been appended to the vector; notice also that the names are retained. In order to avoid having redundant keywords for the elements of `my_list$C`, their names have been appended by numbers corresponding to their order in the original vector.

Because of its flexibility, lists are often used to hold the outputs of functions with heterogeneous types of return values. For example, a regression model generates coefficient estimates, residuals, the model formula, degrees of freedom, and more.

Data Frames

The data frame is the closest analog in R to the data table, a structure with data organized into rows and columns, with rows representing individual observational units and columns representing the variables for each. As with lists, data frames allow for heterogeneous collections of variables. Unlike lists, however, the data frame has a rigid structure whereby rows and columns are associated: each row has the same number of columns, and vice versa. Lists, recall, could have any number of elements within each member of the list.

Let’s try importing the `haart.csv` sample dataset into a dataframe using `read.csv`. Here is what the first 4 lines (header + 3 data rows) look like:

```
"male", "age", "aids", "cd4baseline", "logvl", "weight", "hemoglobin", "init.reg",
1, 25, 0, NA, NA, NA, NA, "3TC, AZT, EFV", "2003-07-01", "2007-02-26", 0, NA, 0, 365, 0, 1
1, 49, 0, 143, NA, 58.0608, 11, "3TC, AZT, EFV", "2004-11-23", "2008-02-22", 0, NA, 0, 365
1, 42, 1, 102, NA, 48.0816, 1, "3TC, AZT, EFV", "2003-04-30", "2005-11-21", 1, "2006-01-
```

You can see that this appears to be a “clean” file, with respect to the appropriate formatting for `read.table`: the strings are enclosed in quotes, NA is used for missing values, and there do not appear to be any troublesome variables (at least, based on the first 3 rows!).

Ensure that you switch to the directory containing the file (either using the menu item or `setwd`), and import the data:

```
> haart <- read.csv("haart.csv")
```

If there are no error messages, then the data was successfully imported and turned into a data frame called `haart`. It is often useful to be able to peek at the first few lines, without scrolling through the entire data file. For this, we call the `head` function:

```
> head(haart)
  male age aids cd4baseline logvl weight hemoglobin init.reg init.date
1    1  25   0         NA     NA      NA         NA 3TC,AZT,EFV 2003-07-01
2    1  49   0        143     NA  58.0608        11 3TC,AZT,EFV 2004-11-23
3    1  42   1        102     NA  48.0816         1 3TC,AZT,EFV 2003-04-30
4    0  33   0        107     NA  46.0000        NA 3TC,AZT,NVP 2006-03-25
5    1  27   0         52     4      NA         NA 3TC,D4T,EFV 2004-09-01
6    0  34   0        157     NA  54.8856        NA 3TC,AZT,NVP 2003-12-02

  last.visit death date.death event followup lfup pid
1 2007-02-26     0      <NA>     0      365     0   1
2 2008-02-22     0      <NA>     0      365     0   2
3 2005-11-21     1 2006-01-11     0      365     0   3
4 2006-05-05     1 2006-05-07     1       43     0   4
5 2007-11-13     0      <NA>     0      365     0   5
6 2008-02-28     0      <NA>     0      365     0   6
```

Similarly, `tail` gives the last several lines of the data frame:

```
> tail(haart)
  male age aids cd4baseline logvl weight hemoglobin init.reg
4625  1 29.00000 NA         NA      NA 42.0000 10.53333 3TC,DDI
4626  1 25.00000   0        136 4.875061 57.0000        NA 3TC,AZT,LPV
4627  0 27.00000   0        232      NA      NA         NA 3TC,AZT
4628  1 38.72142   0        170      NA 84.0000        NA 3TC,AZT
4629  1 23.00000 NA         154 3.995635 65.5000 14.00000 3TC,DDI
4630  0 31.00000   0        236      NA 45.8136        NA 3TC,D4T

  init.date last.visit death date.death event followup lfup pid
4625 2007-03-17 2007-03-28     0      <NA>     0      11     0 4625
4626 2006-10-25 2007-08-02     0      <NA>     0     281     0 4626
4627 2003-12-01 2004-01-05     0      <NA>     0      35     1 4627
4628 2002-09-26 2004-03-29     0      <NA>     0     365     0 4628
4629 2007-01-31 2007-04-16     0      <NA>     0      75     0 4629
4630 2003-12-03 2007-10-11     0      <NA>     0     365     0 4630
```

There appear to be several missing values, but they appear to have been handled correctly. Notice that, as advertised, the character column `init.reg` (initial regimen) was converted to a factor. Its not clear that we want this, so we can re-import the data with the `as.is` flag set to `TRUE`, since there are no other character fields (other than dates) in the dataset.

Indexing

Thanks to the header, each column of the data frame has a unique, meaningful name. Just as with a list, individual variables in the data frame can be extracted using the dollar sign notation:

```
> haart$age[1:50]
 [1] 25.00000 49.00000 42.00000 33.00000 27.00000 34.00000 39.00000 31.00000
 [9] 52.00000 23.00000 49.40726 43.00000 42.00000 30.82272 37.00000 43.00000
[17] 35.00000 33.85079 38.00000 41.00000 35.00000 39.60575 32.00000 57.00000
[25] 29.00000 41.00000 27.00000 54.00000 42.00000 49.79877 38.00000 22.00000
```

```
[33] 32.00000 36.00000 52.00000 31.01164 32.00000 37.00000 23.88501 32.00000
[41] 28.00000 19.00000 29.00000 32.00000 47.52361 61.44559 50.00000 42.00000
[49] 48.00000 24.00000
```

Equivalently, columns can be indexed by column number, using the double-bracket notation; in this case, the `age` variable can be accessed via `haart[[2]]`, since it is the second column (recall the double-brackets index the vector, while single brackets will generate another data frame). Notice that I indexed just the first 50 values, since the dataset is large. How large? We can check:

```
> dim(haart)
[1] 4630 16
```

The dimensions are 4630 rows (observations) by 16 columns (variables).

We can also index out multiple specific columns by using a vector of variable names:

```
> x <- haart[c("male", "age", "event")]
> head(x)
  male age event
1     1  25     0
2     1  49     0
3     1  42     0
4     0  33     1
5     1  27     0
6     0  34     0
```

Not only can we index columns, but we can also extract particular rows according to the value of one or more variables. For example, perhaps we are interested only in the above columns for male individuals:

```
> y <- x[x$male==1,]
> head(y)
  male age event
1     1  25     0
2     1  49     0
3     1  42     0
5     1  27     0
8     1  31     0
9     1  52     0
```

Notice the comma after `x$male==1`, which indicates that we are interested in the *rows* with that criterion. Indeed, we could have combined the previous two operations into a single call that subsets the appropriate rows and columns:

```
> y <- haart[haart$male==1, c("male", "age", "event")]
> head(y)
  male age event
1     1  25     0
2     1  49     0
3     1  42     0
5     1  27     0
```

```

8      1  31      0
9      1  52      0

```

So, the order of indexing for data frames is always rows first, then columns.

Modifying and creating variables

Another operation we might consider, if we plan on comparing dates in the dataset, is to convert the date fields (`init.date`, `last.visit`, `date.death`) from the character type that they were imported as into proper `POSIXct` or `POIXlt` objects.

```

> haart$last.visit <- as.POSIXct(haart$last.visit)
> haart$init.date <- as.POSIXct(haart$init.date)
> haart$date.death <- as.POSIXct(haart$date.death)

```

Suppose now we wish to create a derived variable, one based on the values of one or more other variables in the data frame. For example, we might want to refer to the number of days between `init.date` and the last visit. As we were told, now that we have `POSIXct` objects, we can simply subtract the later date from the earlier to get the time elapsed between visits:

```

> (haart$last.visit - haart$init.date)[1:50]
Time differences in secs
 [1] 115434000 102470400  80874000   3538800 100918800 133833600 129164400
 [8] 171680400  70851600   7257600  21942000  37846800  77065200  42253200
[15]   7344000   432000   56761200  45795600 154137600  68688000 107139600
[22]  34992000  94176000 112492800  20822400 109728000  35679600  1814400
[29]  24710400  61344000  95904000  22892400  94867200  90806400 118544400
[36]  48812400  81565200   9075600  70938000 100998000  34992000  2073600
[43]  96768000 109897200  59875200 137552400 104630400 103852800  74480400
[50] 139798800

```

But, as we can see, the time is given in seconds. One solution is to use the `difftime` function, which takes an optional `units` argument that we can set to “days”:

```

> difftime(haart$last.visit, haart$init.date, units="days")[1:50]
Time differences in days
 [1] 1336.04167 1186.00000  936.04167   40.95833 1168.04167 1549.00000
 [7] 1494.95833 1987.04167  820.04167   84.00000  253.95833  438.04167
[13]  891.95833  489.04167   85.00000    5.00000  656.95833  530.04167
[19] 1784.00000  795.00000 1240.04167  405.00000 1090.00000 1302.00000
[25]  241.00000 1270.00000  412.95833   21.00000  286.00000  710.00000
[31] 1110.00000  264.95833 1098.00000 1051.00000 1372.04167  564.95833
[37]  944.04167  105.04167  821.04167 1168.95833  405.00000   24.00000
[43] 1120.00000 1271.95833  693.00000 1592.04167 1211.00000 1202.00000
[49]  862.04167 1618.04167

```

An even easier approach, since we are only interested in days, is to convert the dates to `Date` objects, rather than `POSIXct`; this class ignores units smaller than day. Subtracting these dates results in a difference in days expressed as integers:

```
> (as.Date(haart$last.visit) - as.Date(haart$init.date))[1:50]
Time differences in days
 [1] 1336 1186  936   41 1168 1549 1495 1987  820   84  254  438  892  489
[16]    5  657  530 1784  795 1240  405 1090 1302  241 1270  413   21  286
[31] 1110  265 1098 1051 1372  565  944  105  821 1169  405   24 1120 1272
[46] 1592 1211 1202  862 1618
```

Since this is what we want, we can add this derived variable to our data frame:

```
> haart$time.diff <- as.Date(haart$last.visit) -
+ as.Date(haart$init.date)
```

Another common operation is the creation of variable categories from raw values. For example, perhaps we want to classify subjects into age groups, with those 30 or younger in the youngest group, those over 30 but no older than 50 in the middle group, and those over 50 in the oldest group. R includes a useful function called `cut` that will take care of this. It only requires the boundaries of the three groups, which implies five values:

```
> haart$age_group <- cut(haart$age, c(min(haart$age), 30, 50,
+ max(haart$age)))
```

This creates a group for each group of ages in (18,30], (30, 50], and (50, 89]:

```
> table(haart$age_group)

(0, 30]  (30, 50]  (50, Inf]
  1167      3021      442
```

If we wanted to use less than (rather than less than or equal to), we could have specified `right=FALSE` to move the boundary values into the upper group:

```
> table(cut(haart$age, c(min(haart$age), 30, 50, max(haart$age)),
+ right=FALSE))

[18, 30)  [30, 50)  [50, 89)
  1011      3115      502
```

Now let's look at another variable that requires special treatment. The field `init.reg` describes the initial drug regimens of each individual, and is imported by default as a factor. However, each entry is in fact a list of drugs, and is difficult to interpret as a factor per se. There are two approaches to making this variable more usable.

First, the type of the variable can be changed to something more sensible, such as a list. This requires a handful of steps; first, we will convert the variable to a character type, and assign it (temporarily) to an external variable:

```
> init.reg <- as.character(haart$init.reg)
```

The reason that we do this is to take advantage of the `strsplit` function, which takes two primary arguments, a character vector that we wish to split up and a character string that we want to use as a delimiter for splitting. In our case, we do the following:

```

> haart$init.reg.list <- strsplit(init.reg, ",")
> head(haart$init.reg.list)
[[1]]
[1] "3TC" "AZT" "EFV"

[[2]]
[1] "3TC" "AZT" "EFV"

[[3]]
[1] "3TC" "AZT" "EFV"

[[4]]
[1] "3TC" "AZT" "NVP"

[[5]]
[1] "3TC" "D4T" "EFV"

[[6]]
[1] "3TC" "AZT" "NVP"

```

Now you can see that the variable `init.reg.list` is a list, each element of which can in turn contain an arbitrary number of elements. So, it can accommodate regimens of different combinations of drugs. How can we use this? Perhaps we want to know all the patients that have D4T as part of their regimens. We can use an `apply` function to search the `init.reg.list` variable to see if it contains the value D4T. For this, we make list of the `%in%` operator, which returns `TRUE` if the value on the left hand side of the operator is contained in the vector on the right hand side, or `FALSE` otherwise. Using this in a function passed to `sapply`, we get a list of `TRUE` and `FALSE` values, which can be used to index the rows that contain D4T in their regimens:

```

> haart.D4T <- haart[sapply(haart$init.reg.list, function(x)
+ 'D4T' %in% x), ]
> head(haart.D4T)
  male age aids cd4baseline logvl weight hemoglobin init.reg i
5    1 27.00000    0         52 4.000000      NA      NA 3TC,D4T,EFV 20
16   0 43.00000    1         49      NA      NA    3.00000 3TC,D4T,NVP 20
18   1 33.85079    1          4      NA     64   12.00000 3TC,D4T,EFV 20
25   0 29.00000   NA         25 4.463878    44      NA 3TC,D4T,EFV 20
30   0 49.79877    1        207      NA     36   10.66667 3TC,D4T,EFV 20
38   0 37.00000    1         NA      NA     NA   12.80000 3TC,D4T,NVP 20

  date.death event followup lfup pid male_factor init.reg.list
5      <NA>     0       365    0   5      Male 3TC, D4T, EFV
16 2004-06-12    1         5    0  16     Female 3TC, D4T, NVP
18      <NA>     0       365    0  18      Male 3TC, D4T, EFV
25      <NA>     0       241    0  25     Female 3TC, D4T, EFV
30      <NA>     0       365    0  30     Female 3TC, D4T, EFV
38      <NA>     0       105    0  38     Female 3TC, D4T, NVP

```

Another (slightly more complicated) way to transform `init.reg` is to break it into multiple columns of indicators, which specify whether each drug is in that individual's regimen. The first step here is to obtain a unique list of all the drugs in all the

regimens. Recall the function `unlist`, which takes all the list elements and concatenates them together. We can use this to get a non-unique vector of drugs:

```
> unlist(haart$init.reg.list)[1:25]
[1] "3TC" "AZT" "EFV" "3TC" "AZT" "EFV" "3TC" "AZT" "EFV" "3TC" "AZT" "NVP"
[17] "AZT" "NVP" "3TC" "AZT" "NVP" "3TC" "AZT" "EFV" "3TC"
```

Now, we use the function `unique` to extract the unique items within this vector, which comprises a list of all the drugs:

```
> (all_drugs <- unique(unlist(haart$init.reg.list)))
[1] "3TC" "AZT" "EFV" "NVP" "D4T"
[8] "IDV" "LPV" "RTV" "SQV" "FTC"
[15] "NFV" "T20" "ATV" "FPV" "TPV"
[22] "APV"
```

Now that we have all the drugs, we want a logical vector for each drug that identifies its inclusion for each individual. We have already seen how to do this, for D4T:

```
> sapply(haart$init.reg.list, function(x) 'D4T' %in% x)[1:25]
[1] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[17] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE TRUE
```

What we need to do now is generalize this by writing a function that performs this operation for each drug in turn. This can be done in one line:

```
> for (drug in all_drugs) sapply(haart$init.reg.list,
+ function(x) drug %in% x)
```

Notice that when you run this function, nothing is returned. This is because we have not assigned the resulting vectors to variables, nor have we specified that they be printed to the screen. Let's turn them into a data frame of their own. We can use the function `cbind`, which stands for “column bind”, concatenating vectors together column-wise. Let's create an empty vector to hold these, then include `cbind` in the loop, adding each logical vector as it is created:

```
> reg_drugs <- c()
> for (drug in all_drugs) reg_drugs <- cbind(reg_drugs,
+ sapply(haart$init.reg.list, function(x) drug %in% x))
> head(reg_drugs)
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
[1,] TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[2,] TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[3,] TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[4,] TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[5,] TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[6,] TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
      [,17] [,18] [,19] [,20] [,21] [,22]
[1,] FALSE FALSE FALSE FALSE FALSE FALSE
[2,] FALSE FALSE FALSE FALSE FALSE FALSE
```

```
[3,] FALSE FALSE FALSE FALSE FALSE FALSE
[4,] FALSE FALSE FALSE FALSE FALSE FALSE
[5,] FALSE FALSE FALSE FALSE FALSE FALSE
[6,] FALSE FALSE FALSE FALSE FALSE FALSE
```

Turning this into a data frame is as simple as a call to `data.frame`, using `all_drugs` as a set of column labels:

```
> reg_drugs.df <- data.frame(reg_drugs)
> names(reg_drugs.df) <- all_drugs
```

Of course, we really want these variables to be part of our full data set, so we can again use `cbind` to merge them into a single data frame:

```
> haart_merged <- cbind(haart, reg_drugs.df)
```

Subsetting

Though you can manually extract subsets of a particular data frame by manually indexing rows, the `subset` function is a more convenient method for extensive subsetting. For example, we may want to select the endpoint event, weight and hemoglobin for just the male subjects over 30 years old. This is straightforward:

```
> haart_m30 <- subset(haart, male==1 & age>30, select=c(event, weight,
+ hemoglobin))
> head(haart_m30)
  event weight hemoglobin
2     0 58.0608   11.00000
3     0 48.0816    1.00000
8     0      NA        NA
9     0      NA        NA
11    1 57.0000   12.33333
12    0 48.0000        NA
```

So, the first argument is the data frame of interest, the second argument are the subset conditions and the third is a vector of variables to be included in the resulting dataset.

Attaching data frames

If you do not wish to prefix each variable by the name of the data frame, it is possible to attach the data frame to the current environment. This moves the variables in the data frame's environment into the environment of your current workspace. Thus,:

```
> attach(haart)
> weight[1:20]
[1]      NA 58.0608 48.0816 46.0000      NA 54.8856 55.3392      NA      NA
[10]      NA 57.0000 48.0000 55.3392 53.0000      NA      NA      NA 64.0000
[19] 61.2360 73.0000
> death[1:20]
[1] 0 0 1 1 0 0 0 0 0 1 1 0 0 1 0 1 0 0 0 0
```

This function should be used carefully; some R users recommend that it not be used at all. This is because it is easy to cause namespace conflicts; this is when variables that already exist in one environment (namespace) are silently overwritten by other variables that are attached. So, if a `weight` variable already existed (could have been another vector, a function, or anything), it would have been replaced by `haart$weight` without warning.

If you do choose to use `attach`, it is good practice to detach the workspace at the end of any script that attaches it:

```
> detach(haart)
> weight
Error: object 'weight' not found
```

Note that the changing the variables that are attached to a particular environment does not change them in the original data frame:

```
> attach(haart)
> weight[1:5] = 100
> weight[1:20]
 [1] 100.0000 100.0000 100.0000 100.0000 100.0000 100.0000  54.8856  55.3392
 [9]      NA      NA  57.0000  48.0000  55.3392  53.0000      NA
[17]      NA  64.0000  61.2360  73.0000
> haart$weight[1:20]
 [1]      NA 58.0608 48.0816 46.0000      NA 54.8856 55.3392      NA
[10]      NA 57.0000 48.0000 55.3392 53.0000      NA      NA      NA 64.0000
[19] 61.2360 73.0000
```

apply functions

In some situations, users may want to apply functions to elements of a list or data frame. To facilitate this, there is a family of functions called `apply` functions that permit functions to be called on subsets of data without having to manually loop over elements in complex data structures.

`tapply` applies a function to different subsets of the data, grouped according to factor variables. For example, suppose we wanted to know the mean weight of subjects by gender:

```
> tapply(haart$weight, haart$male, mean, na.rm=TRUE)
 0      1
51.65059 60.33728
```

The first argument is the target vector to which the function will be applied, the second argument is the index variable that dictates by what factor the application of the function will be grouped, and the third argument is the function that will be used. Finally, notice the use of the `na.rm=FALSE` flag. This tells `tapply` to ignore the missing values, as there are several in the `weight` column.

Multiple factors can be passed to `tapply` simultaneously, resulting in cross-tabulated output:

```
> tapply(haart$weight, haart[c("male", "aids")], mean, na.rm=TRUE)
```

```

      aids
male    0      1
0 53.95558 48.41616
1 63.11145 57.38151

```

A simpler, related function, `lapply` simply vectorizes (*i.e.* allows it to operate on vectors) a function to each variable of a list or data frame, without using a grouping factor, and returns the results as a list. The related function `sapply` does the same thing, but tries to return a simpler data structure, generally a vector. Vectorization can be very convenient. For example, we may simply want to know which of our variables are numeric:

```

> sapply(haart, is.numeric)
      male      age      aids cd4baseline      logvl      weight
      TRUE      TRUE      TRUE      TRUE      TRUE      TRUE
hemoglobin  init.reg  init.date  last.visit      death  date.death
      TRUE      FALSE      FALSE      FALSE      TRUE      FALSE
event  followup      lfup      pid  time.diff  age_group
      TRUE      TRUE      TRUE      TRUE      FALSE      TRUE

```

Or, perhaps we are interested in standardizing some of the variables in our data frame:

```

> sapply(haart[c("cd4baseline", "weight", "hemoglobin")],
+ scale)[1:5,]
      cd4baseline      weight  hemoglobin
[1,]      NA      NA      NA
[2,] 0.08539255 0.09425106 -0.06792736
[3,] -0.24968729 -0.72919671 -4.23067363
[4,] -0.20882389 -0.90096287      NA
[5,] -0.65832124      NA      NA

```

In this case, we probably either want to replace the unscaled variables with the scaled values, or append a new set of variables to the data frame:

```

> haart[c("cd4baseline_scaled", "weight_scaled", "hemoglobin_scaled")]
+ <- sapply(haart[c("cd4baseline", "weight", "hemoglobin")], scale)
> head(haart)  male age aids cd4baseline logvl  weight hemoglobin  init.r
1      1  25    0      NA      NA      NA      NA 3TC,AZT,EFV 2003-07-01
2      1  49    0     143     NA 58.0608     11 3TC,AZT,EFV 2004-11-23
3      1  42    1     102     NA 48.0816      1 3TC,AZT,EFV 2003-04-30
4      0  33    0     107     NA 46.0000     NA 3TC,AZT,NVP 2006-03-25
5      1  27    0      52      4      NA     NA 3TC,D4T,EFV 2004-09-01
6      0  34    0     157     NA 54.8856     NA 3TC,AZT,NVP 2003-12-02
      last.visit death date.death event followup lfup pid time.diff age_group
1 2007-02-26      0      <NA>      0      365    0    1 1336 days      1
2 2008-02-22      0      <NA>      0      365    0    2 1186 days      2
3 2005-11-21      1 2006-01-11      0      365    0    3  936 days      2
4 2006-05-05      1 2006-05-07      1       43    0    4   41 days      2
5 2007-11-13      0      <NA>      0      365    0    5 1168 days      1

```

	6	2008-02-28	0	<NA>	0	365	0	6	1549 days	2
		cd4baseline_scaled		weight_scaled		hemoglobin_scaled				
1		NA		NA		NA				
2		0.08539255		0.09425106		-0.06792736				
3		-0.24968729		-0.72919671		-4.23067363				
4		-0.20882389		-0.90096287		NA				
5		-0.65832124		NA		NA				
6		0.19981006		-0.16775505		NA				

Sorting

Though the `sort` function in R is the easiest way to sort the elements of a vector, we are usually interested in sorting entire records/observations/rows according to the value of one or more parameters. In this case, it is a two-step process.

First, we create a numeric vector of the indices of each row in our data frame, according to the order that we wish to have them. We can generate such a vector using the `order` function. This simply requires as arguments the variables you wish to sort by, in preferred sort order. For example, we might want to order our HAART dataset first by `init.date` and then by `last.visit`. Hence, the call is (showing the first 5 entries):

```
> order(haart$init.date, haart$last.visit)[1:5]
[1] 4375 4466 2874 1891 4458
```

So, the earliest date is the 4375th row, the second 4466, *et cetera*. The second step is to use these index values to generate a sorted version of our data frame:

```
> haart_sorted <- haart[order(haart$init.date, haart$last.visit),]
> head(haart_sorted)
  male age aids cd4baseline logvl weight hemoglobin init.reg
4375  0  24  0         450 4.538071    61      NA D4T,DDI,RTV
4466  1  32  0         NA      NA      NA      NA 3TC,AZT,IDV
2874  1  51  0         NA      NA      NA      NA 3TC,AZT,EFV
1891  1  32  0         NA      NA      74      NA 3TC,AZT,IDV
4458  1  53  1        290      NA      NA      NA 3TC,AZT,IDV
1455  1  43  0         NA      NA      NA      NA 3TC,D4T,IDV

  init.date last.visit death date.death event followup lfup pid time.o
4375 1997-01-01 2007-02-13    0      <NA>    0    365    0 4375 3695 d
4466 1997-06-15 2007-03-29    0      <NA>    0    365    0 4466 3574 d
2874 1997-06-15 2008-01-30    0      <NA>    0    365    0 2874 3881 d
1891 1997-10-29 2006-10-01    0      <NA>    0    365    0 1891 3259 d
4458 1997-12-27 2007-03-07    0      <NA>    0    365    0 4458 3357 d
1455 1998-04-01 2007-04-16    0      <NA>    0    365    0 1455 3302 d

  age_group cd4baseline_scaled weight_scaled hemoglobin_scaled
4375      1         2.594405    0.3367833      NA
4466      2              NA      NA      NA
2874      3              NA      NA      NA
1891      2              NA    1.4094966      NA
4458      3         1.286776      NA      NA
1455      2              NA      NA      NA
```

Saving data frames to files

The function `write.table` allows data frames to be written to a file, so that it may be moved, backed up or reloaded later. One only needs to supply the name of the data frame and the name of a file (which likely does not yet exist):

```
> write.table(haart, "haart.dat")
```

By default, this writes the data to a space-delimited text file, which looks like this:

```
"male" "age" "aids" "cd4baseline" "logvl" "weight" "hemoglobin" "init.reg"
"1" 1 25 0 NA NA NA NA "3TC,AZT,EFV" 2003-07-01 2007-02-26 0 NA 0 365 0 1 1
"2" 1 49 0 143 NA 58.0608 11 "3TC,AZT,EFV" 2004-11-23 2008-02-22 0 NA 0 365
"3" 1 42 1 102 NA 48.0816 1 "3TC,AZT,EFV" 2003-04-30 2005-11-21 1 2006-01-1
"4" 0 33 0 107 NA 46 NA "3TC,AZT,NVP" 2006-03-25 2006-05-05 1 2006-05-07 1
"5" 1 27 0 52 4 NA NA "3TC,D4T,EFV" 2004-09-01 2007-11-13 0 NA 0 365 0 5 11
```

Additional options are available; for example, you may want to use a comma as the delimiter, rather than a space, in which case you would add `sep=","` to the list of arguments to `write.table`. Similarly, you can change the default missing value, string quoting, the precision of decimals, and more (see the help file).

For simpler data structures, there is a similarly simpler function, `write`. It also takes the name of the object to be saved, and the name of a file, and writes the data to that file. This is more appropriate for vectors and matrices.:

```
> x <- matrix(1:10,ncol=5)
> write(x, "some_data.dat")
```

Another alternative to write the binary representation to a file, using the `save` function. This is usually a portable, machine-independent format and allows for compression of very large data:

```
> save(x, file="x.rda", compress=TRUE)
```

Now, however, the contents of the file are only machine-readable:

```
^_<8B>^H^@^@^@^@^@^@^C^Kr<89>0<E2><8A><E0>b``b`<E2>^E<92><CC>@&^K^S<90>``^
```

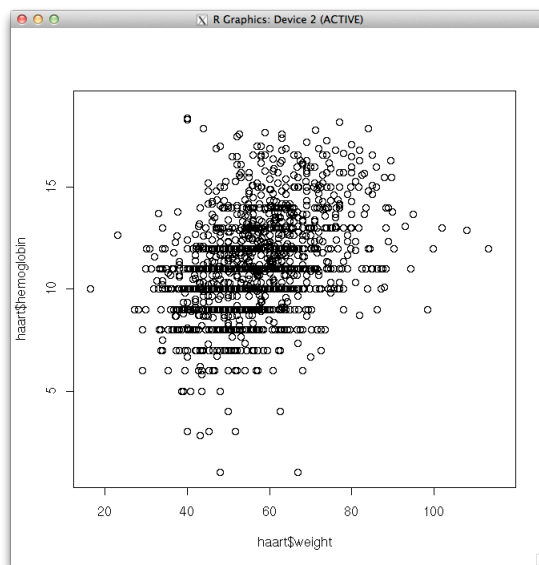
Binary files can be imported back into R using the `load` function, which only takes the data file itself as an argument.

Plotting

R offers a few different plotting options, but here we will focus on the built-in graphics functionality.

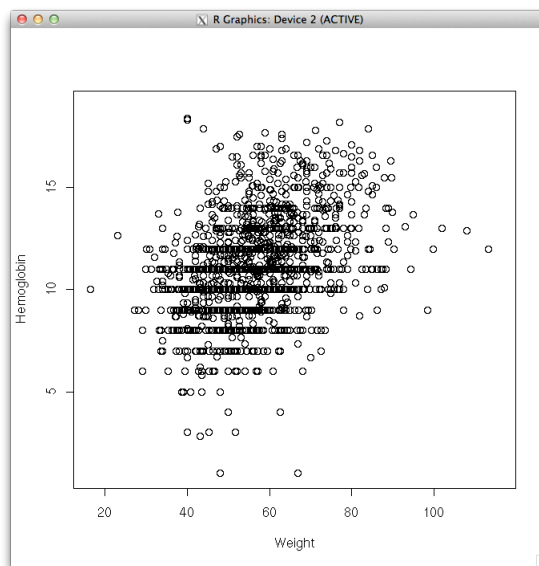
At the very simplest level, one may generate a scatterplot by calling the `plot` function with the predictor and response variables as an argument:

```
> plot(haart$weight, haart$hemoglobin)
```



Notice here that some values of hemoglobin appear to have been rounded to the nearest integer. We can make aesthetic improvements to the plot by adding axis labels:

```
> plot(haart$weight, haart$hemoglobin, xlab="Weight", ylab="Hemoglobin")
```



We can entirely change the type of plot by specifying the `type` argument, which allows for a variety of arguments:

- “p”: points (the default)
- “l”: lines
- “b”: both, with gaps in the lines for the points
- “c”: the lines part alone of “b”, which is useful if you want to combine lines with other kinds of symbols

- “o”: both lines and points ‘overplotted’, that is, without gaps in the lines
- “h”: vertical lines, giving a ‘histogram’ like plot; “s”: a step function, going across then up
- “S”: a step function, going up then across
- “n”: no plotting

Though `plot` usually does a reasonable job of setting the boundaries of the plot, you can manually change these using `xlim` and `ylim`, as well as adding color to plot symbols (`col`), manipulating the shape of the points (`pch`), and the width of the lines (`lwd`).

Each plotting device (or backend) is specific to whichever platform R is running on: `quartz` for Mac OS X, `X11` for Linux and `windows` for Windows. Instead of passing arguments to the plot itself, parameters for the plotting device can also be set using the `par` function. The list of available parameters is extensive, but here are a few:

- `mfrow = c(a,b)`: creates a matrix of plots (a rows and b columns) on the same page.
- `mar = c(bottom, left, top, right)`: creates margins around individual plots, in character widths units
- `oma = c(bottom, left, top, right)`: creates margins around matrix of plots, in character width units
- `las = 1`: rotates y-axis labels to be horizontal
- `pty = "s"`: forces plots to be square (contrast with `pty = "m"`)
- `new = TRUE`: plots subsequent figures over the previous one, rather than on a new set of axes
- `cex = x`: magnifies symbols by a factor of x
- `bty = "n"`: removes box drawn around plots; a number of other options are available

`par` accepts multiple arguments, so several options can be set at once. Additionally, `par` returns a list of the current parameters, which can be saved to be restored after we are finished with our custom parameters:

```
> old_pars <- par(mfrow=c(2,3), bty="7", mar=c(4,4,1,1))
> plot(<some_plot>)
> par(old_pars)
```

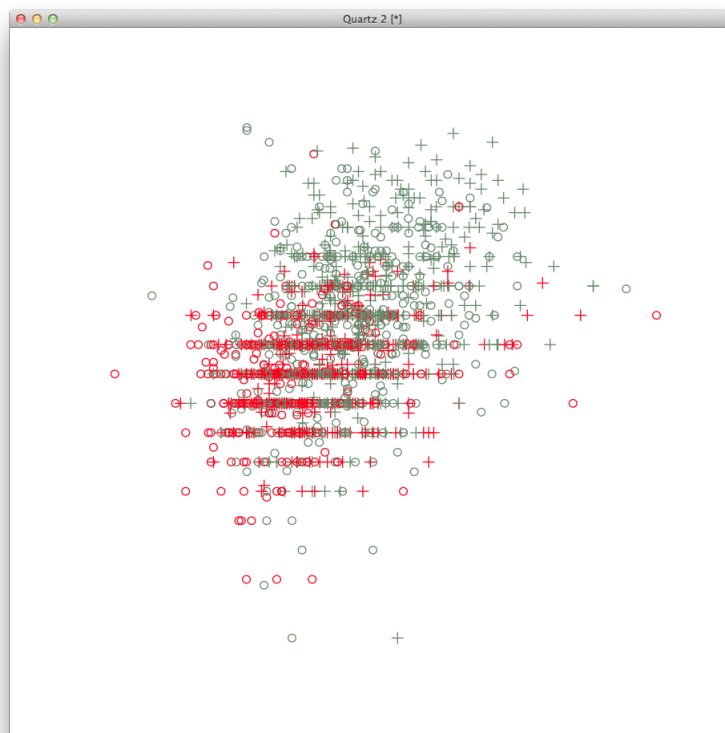
In addition to setting plotting device parameters, plots may be augmented after they are drawn. This allows desired custom plots to be built up in several steps, interactively. Here is an example of a number of plotting functions that are used to customize the layout of a scatterplot:

- 1) Create an “empty” scatterplot that sets up the dimensions of the space, but does not plot anything. This should result in a blank graphics device popping up:

```
> some_pars <- par(las=1, mar=c(4,4,3,2))
> plot(haart$weight, haart$hemoglobin, xlab="", ylab="",
+ axes=FALSE, type="n")
```

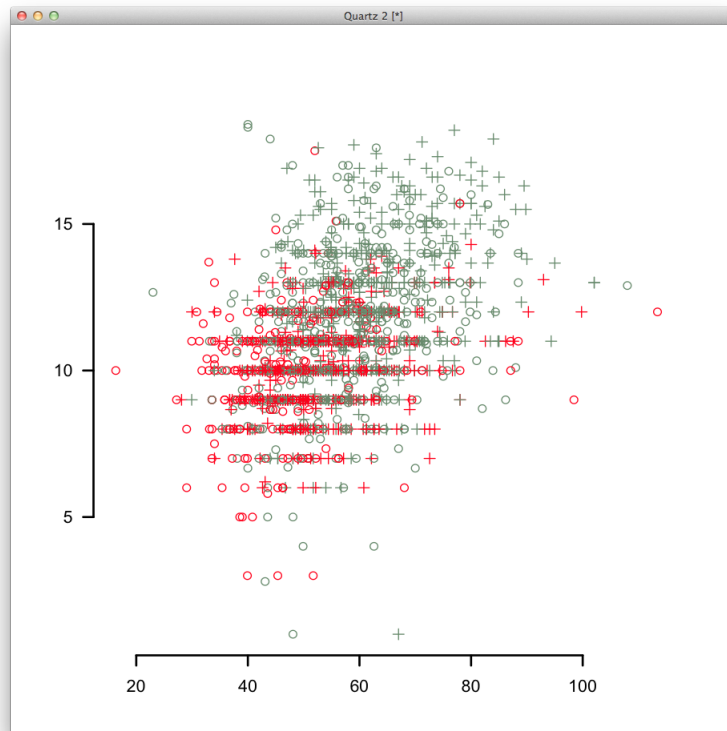
- 2) We now add the points of the scatterplot, using different colors and symbols for different groups:

```
> points(haart$weight, haart$hemoglobin, col=ifelse(haart$male==1,
+ "darkseagreen4", "red"), pch=ifelse(haart$aids==1, 1, 3))
```



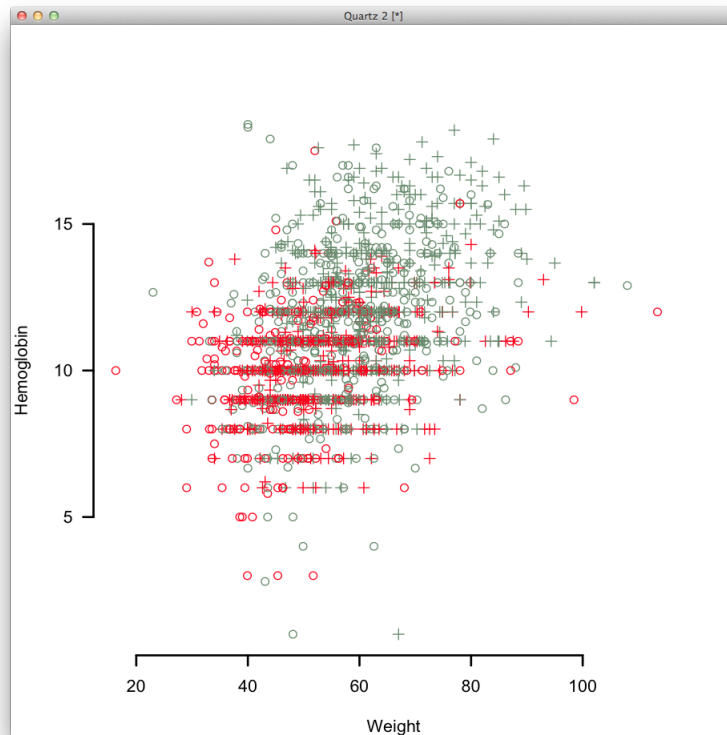
- 3) Now axes can be added, with any desired options:

```
> axis(1, lwd=2)
> axis(2, lwd=2)
```



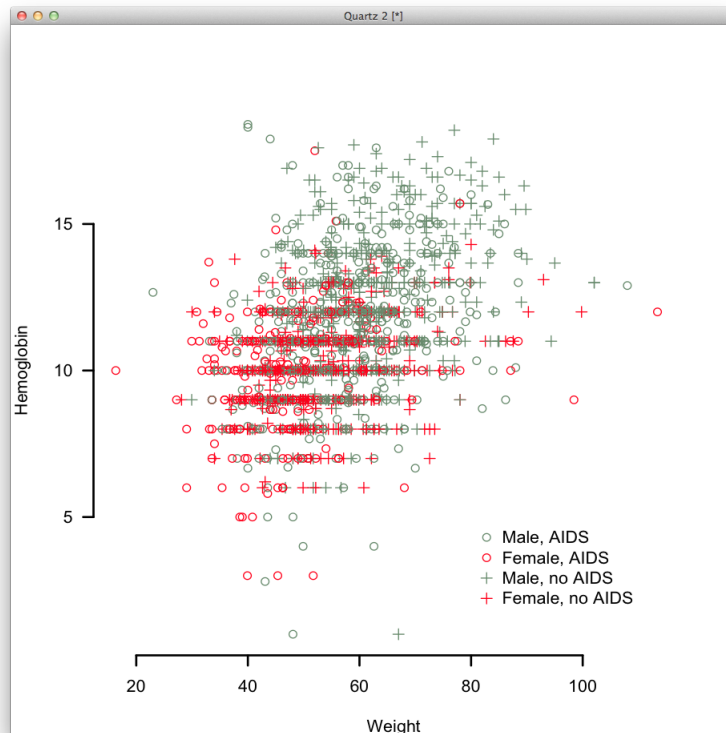
4) Now axis labels can be added, in the desired orientation:

```
> mtext("Weight", side=1, line=3)
> mtext("Hemoglobin", side=2, line=3)
```



5) Finally, a legend to decipher the color and shape of the points:

```
> legend(x=80, y=5,
+ c("Male, AIDS", "Female, AIDS", "Male, no AIDS", "Female, no AIDS"),
+ col=c("darkseagreen4", "red", "darkseagreen4", "red"),
+ pch=c(1,1,3,3), bty="n")
```



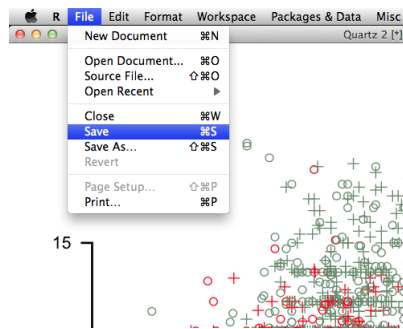
Saving plots

Rather than writing plots to the screen, as we have done, it is straightforward to instead write them directly to a file, where they can be imported into other documents, or posted online. One common format is *pdf*. We simply call the function of the same name in order to use it as a plotting device:

```
> pdf(file="sample_plot.pdf", width=4, height=3)
> plot(haart$weight, haart$hemoglobin, xlab="Weight", ylab="Hemoglobin")
> dev.off()
```

The `dev.off` call closes the pdf file, so that it may be used elsewhere. If we create another plot before closing the file, it will add another page to the pdf file. This can be overridden with `onfile=FALSE`.

If you wish to save a plot that was drawn to the screen, you can use the appropriate menu item for your particular operating system to save the plot to a graphic file format.



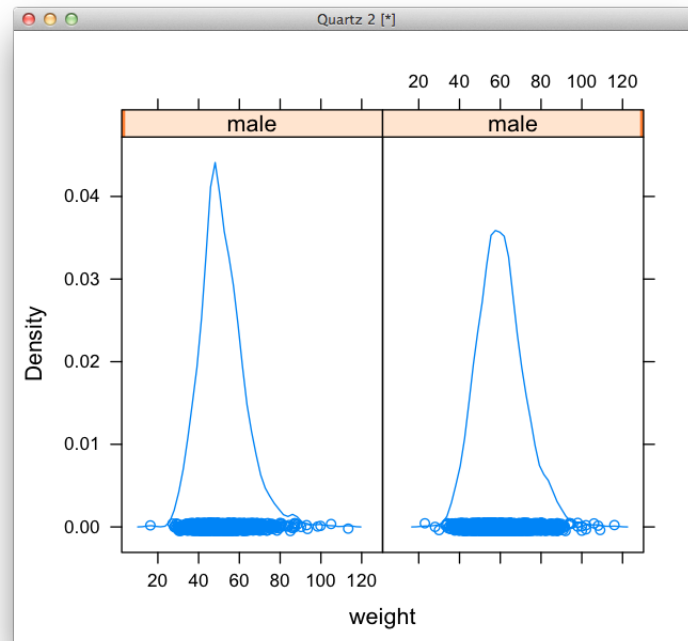
Trellis plots

Often we wish to express more than two dimensions of data simultaneously. While 3-dimensional plots are often available, they tend not to be very effective, as humans are generally not good at interpreting them. A more reasonable alternative is the use of trellis plots, or conditioning plots, which instead display a series of plots, each of which is two dimensional and conditions on particular values of the third variable.

Trellis plots are available in the recommended R package `lattice`. As an example, perhaps we would like to plot the distribution of weights separately for men and women. For this, we can use the `densityplot` function:

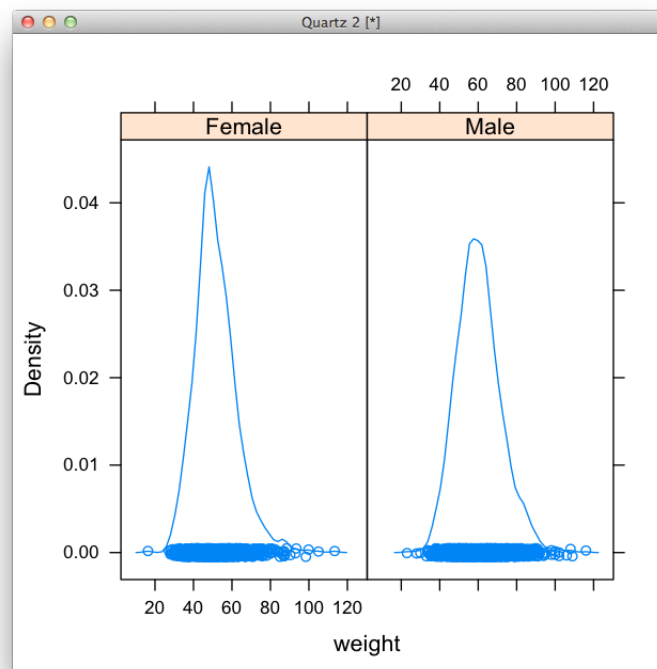
```
> densityplot(~ weight | male, data=haart)
```

This notation is slightly different from previous plotting functions, in that a *model* is given as the primary argument, rather than variables. In this case, we indicate that we would like to plot the distribution of weight (hence, no variable to the left of the tilde), conditioning on the variable `male` (the *pipe* symbol indicates conditioning). Note that you can specify the data frame as an additional `data` argument, so that you do not have to index individual variables. This results in the following:



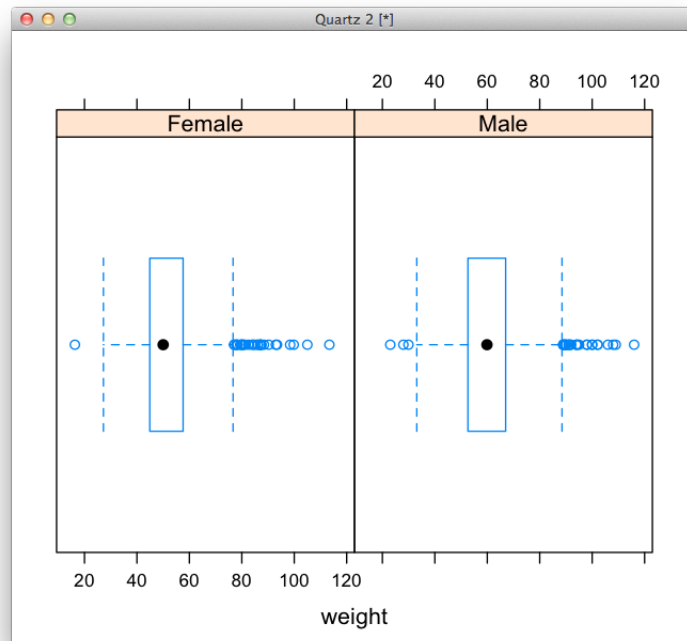
We see separate density plots of weight by sex, with a *rug* of actual data points along the x-axis. However, the labeling of the panels is odd, as they both say “male”; this is because by default it uses the name of the variable as the label, with a dark orange bar indicating the value of the variable for the particular plot (here, 0 and 1). It would be preferable to simply label the panels “Female” and “Male”. For this, we need to change `male` to a factor, and rename its levels. This is done as follows:

```
> haart$male_factor <- factor(haart$male)
> levels(haart$male_factor) <- c("Female", "Male")
> densityplot(~ weight | male_factor, data=haart)
```



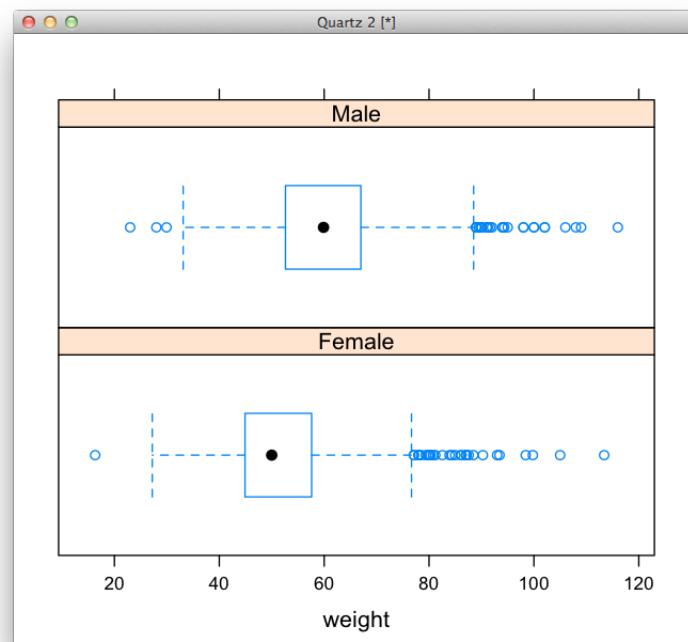
Alternatively, we can display the same data as a box-and-whisker plot, using the `bwplot` function, with similar syntax:

```
> bwplot(~ weight | male_factor, data=haart)
```



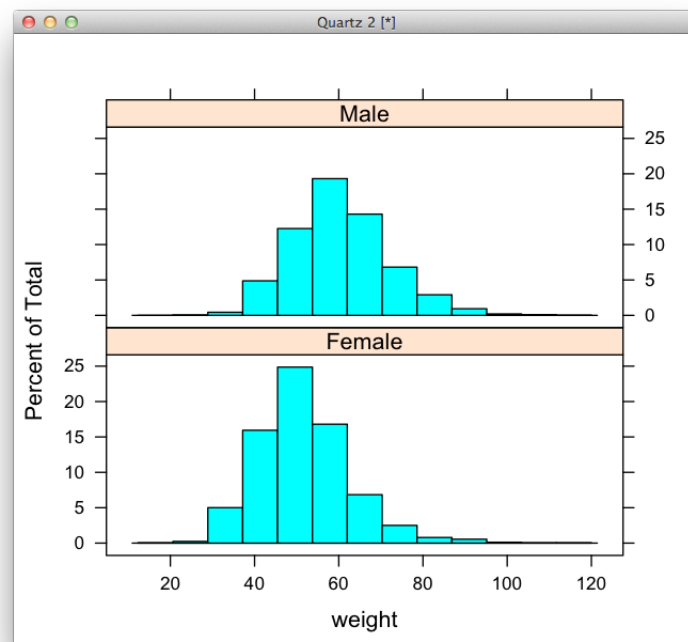
Notice, however, that this is not an optimal layout of the panels; they are compressed horizontally and it is not easy to compare them side-by-side. It would be better displayed row-wise rather than column-wise. We can change this layout easily:

```
> bwplot(~ weight | male_factor, data=haart, layout=c(1,2))
```

Now it is much easier to visualize the difference between the two distributions. Yet a third way to display the same information is using a histogram:

```
> histogram(~ weight | male_factor, data=haart, layout=c(1,2))
```



We can also create paneled scatterplots of the relationship between two variables of interest, conditioned on a third, using `xyplot`:

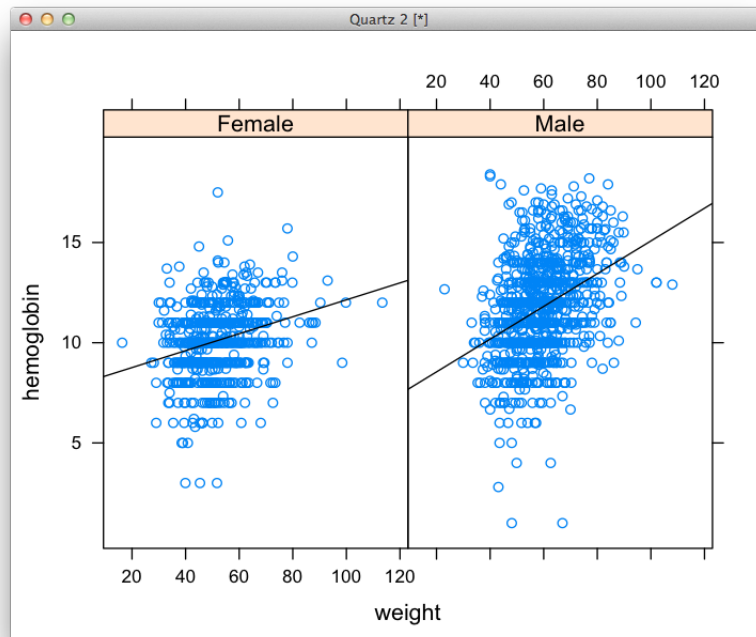
```
> xyplot(hemoglobin ~ weight | male_factor, data=haart)
```



For an advanced touch, suppose we want to add a linear regression line through each panel's scatterplot. This requires using the `panel` argument, which controls the appearance of the plot in each panel. We are going to write a function to do just that: generate a scatterplot of points, and an associated regression line, using another function, `abline`:

```
> regline <- function(x,y,...) {
+   panel.xyplot(x,y,...)
+   panel.abline(lm(y~x), ...)
+ }
> xyplot(hemoglobin ~ weight | male_factor, data=haart, panel=regline)
```

Now, `xyplot` runs the `regline` function for each panel of the plot, passing the appropriate subset of data in each case.



Basic Statistical Analysis

Probability distributions

The `stats` package, included in the base installation of R, provides functions for calculating the density, distribution function, quantile function and random generation for the several common statistical distributions. For example, the functions related to the binomial distribution include: `rbinom` (random number generation), `pbinom` (distribution function), `dbinom` (density function) and `qbinom` (quantile function).

Using `rnorm`, it is easy to generate a sample dataset of normally-distributed values:

```
> ndata = rnorm(100, mean=3, sd=2)
> summary(ndata)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-1.812  1.737   2.802   2.936  4.380   7.286
```

This generates 100 pseudo-random deviates drawn from a normal distribution with mean 3 and standard deviation 2. It is straightforward to generate a 95% confidence interval for the sample mean of this simulated dataset, using the quantile function for Student's t-distribution. Recall that the 95% confidence interval for the mean of normally-distributed data satisfies:

$$Pr(\bar{X} - t_{0.025, n-1}S/\sqrt{n} < \mu < \bar{X} + t_{0.025, n-1}S/\sqrt{n}) = 0.95$$

where $t_{p, n-1}$ is the (100p) percentile of the t-distribution with n-1 degrees of freedom, and S is the sample mean. In R, these confidence bounds can be calculated as:

```
> mean(ndata)+qt(0.025,99)*sqrt(var(ndata)/100)
[1] 2.563714
> mean(ndata)+qt(0.975,99)*sqrt(var(ndata)/100)
[1] 3.307351
```

Hence, the interval is (2.563714, 3.307351).

Two-sample t-Test

Load the built-in dataset CO2, using the data function:

```
> data(CO2)
```

Have a look at the data structure by indexing out the first few rows:

```
> CO2[1:5,]
  Plant   Type Treatment conc uptake
1  Qn1 Quebec nonchilled   95   16.0
2  Qn1 Quebec nonchilled  175   30.4
3  Qn1 Quebec nonchilled  250   34.8
4  Qn1 Quebec nonchilled  350   37.2
5  Qn1 Quebec nonchilled  500   35.3
```

The CO2 data frame has 84 rows and 5 columns of data from an experiment on the cold tolerance of the grass species *Echinochloa crus-galli*. This data frame contains the following columns:

- **Plant**: an ordered factor with levels Qn1 < Qn2 < Qn3 < ... < Mc1 giving a unique identifier for each plant.
- **Type**: a factor with levels {Quebec, Mississippi} giving the origin of the plant
- **Treatment**: a factor with levels non-chilled chilled
- **conc**: a numeric vector of ambient carbon dioxide concentrations (mL/L).
- **uptake**: a numeric vector of carbon dioxide uptake rates (umol/m² sec).

The CO2 uptake of six plants from Quebec and six plants from Mississippi was measured at several levels of ambient CO2 concentration. Half the plants of each type were chilled overnight before the experiment was conducted.

The following command performs a two-sample t-test of the null hypothesis that the mean CO2 uptake of the two plant types are equal:

```
> t.test(uptake~Type, data=CO2)
```

```
Welch Two Sample t-test
```

```
data: uptake by Type
t = 6.5969, df = 78.533, p-value = 4.451e-09
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 8.839475 16.479572
sample estimates:
mean in group Quebec mean in group Mississippi
      33.54286          20.88333
```

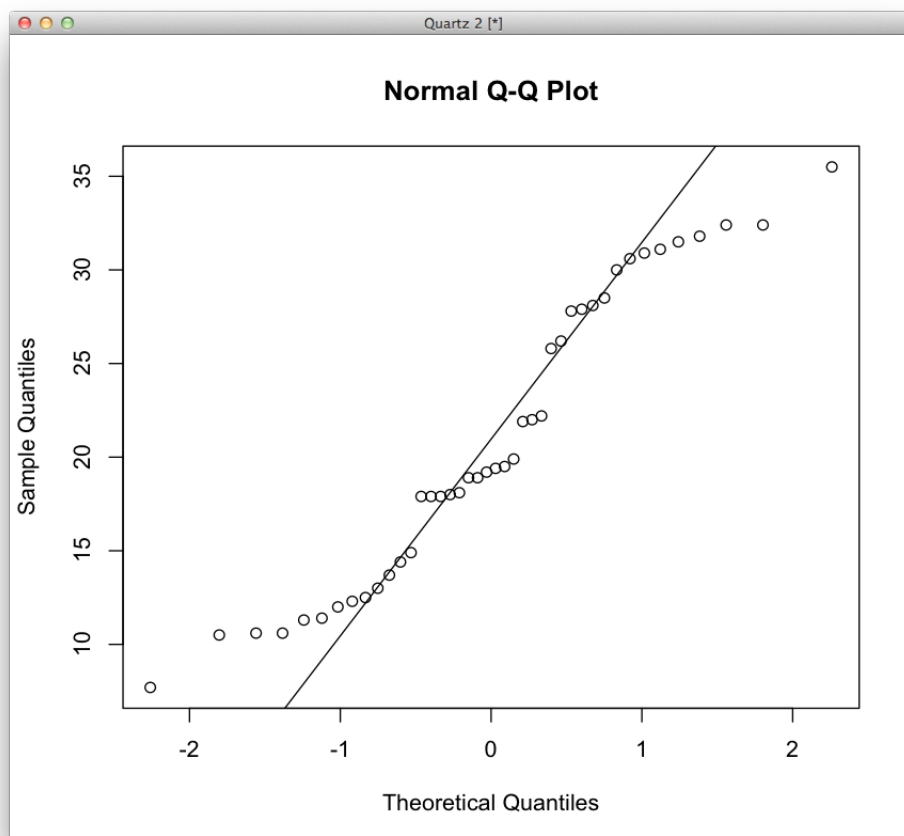
Note that by default, `t.test` assumes that the variances of the two groups are unequal. The test rejects the null hypothesis for any of the usual choices of α . Of course, this isn't a great null hypothesis, is it -- no two biological populations are exactly equal. However, notice that the function also yields a confidence interval for the difference between the two populations. Based on this estimate, the difference could be as small as 8.8 and as large as 16.5.

Checking Normality

Quite a few statistical tests are based on the normality of the underlying population. Here we illustrate the Q-Q plot and Shapiro-Wilk test to check the normality assumption for some of the CO₂ data. The Q-Q plot compares the quantiles of the normal distribution to your sample. If the sample is normally-distributed, the points on the plot should fall approximately on a line drawn at 45 degrees through the origin. Lets generate a Q-Q plot of the plants from Mississippi:

```
> miss = CO2$uptake[CO2$Type=='Mississippi']  
> qqnorm(miss)  
> qqline(miss)
```

This should yield the following plot:



The curve has obvious deviations from the straight line, so we suspect that the data may not in fact be normally-distributed. Lets run a more formal test:

```
> shapiro.test(miss)

      Shapiro-Wilk normality test

data:  miss
W = 0.9363, p-value = 0.0213
```

If the chosen alpha level is 0.05, then the reported p-value rejects the null hypothesis that the data are normally distributed.

Simple linear models

Consider a simple linear regression model, with normal, independent and homoscedastic errors:

$$y \sim \beta_0 + \beta_1 x_1 + \dots + \beta_k x_k + \epsilon_i$$

This can be expressed in matrix notation as:

$$y = X\beta + \epsilon$$

where y is the response vector, X the design matrix and β a vector of coefficients. In R, the tilde (\sim) operator (as seen in the plotting tutorial from last time) is used to define a model formula (*i.e.* a relationship between variables). The general syntax is:

```
<response> ~ <predictive1> <operator> <predictive2> ...
```

For example, several alternative formulations could implement the linear model described above. The simplest is:

```
y ~ x
```

or

```
y ~ 1 + x
```

the former having an implicit intercept term, the latter explicit. Or, the model could be forced through the origin by explicitly removing the intercept term:

```
y ~ 0 + x
y ~ x - 1
```

Alternatively, a second-order polynomial regression could be defined using either:

```
y ~ poly(x, 2)
y ~ 1 + x + I(x^2)
```

Interaction terms may also be defined:

```
y ~ x1 + x2 + x1:x2
y ~ x1*x2
```

These formulations apply broadly to R statistical functions. The basic function for fitting linear models is `lm`. Though more specialized functions exist, it can be used to carry out regression, single stratum analysis of variance and analysis of covariance. A typical `lm` formulation is:

```
lm(y ~ x, data=my.data.frame, weights=my.weights, subset=subset.vector,
+ na.action=na.omit)
```

Note that after the formula, all the additional arguments are optional. `data` specifies the data frame, if used, `weights` is a vector of positive weights, if non-uniform weights are applicable, `subset` is a vector indicating the subset of the data to be used (the entire dataset is used if not specified), and `na.action` instructs the procedure how to handle missing values. By default, `na.omit` is used, which simply excludes them.

Lets examine a sample dataset of plant dry weight data, whereby some plants were subjected to a treatment of some kind:

```
ctl = c(4.17, 5.58, 5.18, 6.11, 4.50, 4.61, 5.17, 4.53, 5.33, 5.14)
```

and others were untreated, as controls:

```
trt = c(4.81, 4.17, 4.41, 3.59, 5.87, 3.83, 6.03, 4.89, 4.32, 4.69)
```

An obvious analysis to conduct for this data is an analysis of variance (ANOVA), which is just a special kind of linear model. Clearly, our response variable are the control and treatment weights; we can concatenate these into a single vector:

```
> weight = c(ctl, trt)
```

The predictor variable is simply the corresponding group for each observation. An easy way to generate factor levels is to use the `gl` function, that uses the number of levels, the number of replications and the group names to generate a vector of levels:

```
> group = gl(2, 10, 20, labels=c("Ctl", "Trt"))
> group
[1] Ctl Ctl Ctl Ctl Ctl Ctl Ctl Ctl Ctl Ctl Trt Trt Trt Trt Trt Trt Trt Trt Trt Trt
[20] Trt
Levels: Ctl Trt
```

We now have everything required to build the linear model. It is common in R to combine variables of interest into a single data structure, called a data frame:

```
> plants = data.frame(weight, group)
> plants
  weight group
1   4.17   Ctl
2   5.58   Ctl
3   5.18   Ctl
4   6.11   Ctl
5   4.50   Ctl
6   4.61   Ctl
7   5.17   Ctl
8   4.53   Ctl
9   5.33   Ctl
10  5.14   Ctl
11  4.81   Trt
12  4.17   Trt
```


13	4.41	Trt
14	3.59	Trt
15	5.87	Trt
16	3.83	Trt
17	6.03	Trt
18	4.89	Trt
19	4.32	Trt
20	4.69	Trt

We can now run a simple linear model by calling `lm` with the appropriate arguments:

```
> wt.lm = lm(weight ~ group, data=plants)
```

The output is sent to another variable `wt.lm`, which can be summarized:

```
> summary(wt.lm)
```

Call:

```
lm(formula = weight ~ group)
```

Residuals:

	Min	1Q	Median	3Q	Max
	-1.0710	-0.4938	0.0685	0.2462	1.3690

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	5.0320	0.2202	22.850	9.55e-15 ***
groupTrt	-0.3710	0.3114	-1.191	0.249

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.6964 on 18 degrees of freedom

Multiple R-Squared: 0.07308, Adjusted R-squared: 0.02158

F-statistic: 1.419 on 1 and 18 DF, p-value: 0.249

Notice that the format of the summary is the very general form that presents the estimates for the intercept and coefficient. Here, the control group is treated as the “baseline” making the coefficient represent the magnitude of effect of the treatment. However, this is not the standard output associated with ANOVA, which is a comparison of the variation within groups to variance among groups. This output can be obtained by calling the `anova` function on the linear model object:

```
> anova(wt.lm)
```

Analysis of Variance Table

Response: weight

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
group	1	0.6882	0.6882	1.4191	0.249
Residuals	18	8.7293	0.4850		

This analysis suggests that the MSE for the groups is not much different from that of the residual. We therefore conclude that there is not a strong effect of the treatment on plant weight.

An alternative approach for assessing the treatment effect is to build a second competing model that does not include a treatment effect. This model includes only an intercept term; that is, no group fixed effect:

```
> wt.lm2 = lm(weight ~ 1, data=plants)
```

Now that we have two models, we can compare them using model selection. A common means for model selection is Akaike's Information Criterion (AIC). We can apply the AIC function to both models:

```
> AIC(wt.lm, wt.lm2)
      df      AIC
wt.lm   3 46.17648
wt.lm2  2 45.69419
```

For ease of interpretation, AIC values can be converted to model weights. Weights are derived by subtracting the smallest AIC value from each model in the set:

$$\Delta AIC_i = AIC_i - \min[AIC]$$

then applying the following formula:

$$w_i = \frac{e^{-\frac{1}{2}\Delta AIC_i}}{\sum_{j=1}^n e^{-\frac{1}{2}\Delta AIC_j}}$$

This is easily done in R:

```
> aic = AIC(wt.lm, wt.lm2)$AIC
> daic = aic - min(aic)
> exp(-0.5*daic)/sum(exp(-0.5*daic))
[1] 0.440004 0.559996
```

This result indicates that there is a 56% probability that the “no effect” model is the true model (assuming the true model is in the set).

If we are interested in fitting other classes of linear models that, for example, have non-Gaussian error distributions, `glm` may be used instead. Consider a model that predicts outcomes based on group and treatment effects, but where the residual error is thought to be Poisson-distributed. A generalized linear model might be constructed as follows:

```
> counts = c(18,17,15,20,10,20,25,13,12)
> group = gl(3,1,9)
> treatment = gl(3,3)
> print(mydf = data.frame(treatment, group, counts))
  treatment outcome counts
1          1         1     18
2          1         2     17
3          1         3     15
4          2         1     20
5          2         2     10
6          2         3     20
7          3         1     25
8          3         2     13
```

```

9          3          3      12
> my.glm = glm(counts ~ group + treatment, family=poisson())
> anova(my.glm)
Analysis of Deviance Table

Model: poisson, link: log

Response: counts

Terms added sequentially (first to last)

              Df Deviance Resid. Df Resid. Dev
NULL              8    10.5814
outcome      2     5.4523      6     5.1291
treatment    2     0.0000      4     5.1291
> summary(my.glm)

Call:
glm(formula = counts ~ outcome + treatment, family = poisson())

Deviance Residuals:
    1         2         3         4         5         6         7         8         9
-0.67125  0.96272 -0.16965 -0.21999 -0.95552  1.04939  0.84715 -0.09656
-0.96656

Coefficients:
              Estimate Std. Error   z value Pr(>|z|)
(Intercept)  3.045e+00  1.709e-01   17.815   <2e-16 ***
outcome2     -4.543e-01  2.022e-01   -2.247   0.0246 *
outcome3     -2.930e-01  1.927e-01   -1.520   0.1285
treatment2    3.999e-16  2.000e-01   2.00e-15  1.0000
treatment3   -3.874e-17  2.000e-01  -1.94e-16  1.0000
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

Null deviance: 10.5814 on 8 degrees of freedom
Residual deviance: 5.1291 on 4 degrees of freedom
AIC: 56.761

Number of Fisher Scoring iterations: 4

```

Model checking

Our next example examines the results of several hill races in Scotland. The hills dataset consists of the record times for 35 different races that involve an elevational gradient. The data frame includes the total map distance (miles), the change in eleva-

tion (feet) and the record time:

```
> data(hills)
> hills[1:5,]
      dist climb  time
Greenmantle  2.5   650 16.083
Carnethy     6.0  2500 48.350
Craig Dunain  6.0   900 33.650
Ben Rha      7.5   800 45.600
Ben Lomond    8.0  3070 62.267
```

A sensible model might consist of predicting time as a function of both dist and climb.

```
> hills.lm = lm(time ~ dist + climb, data=hills)
> summary(hills.lm)
```

Call:

```
lm(formula = time ~ dist + climb, data = hills)
```

Residuals:

Min	1Q	Median	3Q	Max
-16.215	-7.129	-1.186	2.371	65.121

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	-8.992039	4.302734	-2.090	0.0447 *
dist	6.217956	0.601148	10.343	9.86e-12 ***
climb	0.011048	0.002051	5.387	6.45e-06 ***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 14.68 on 32 degrees of freedom
Multiple R-squared: 0.9191, Adjusted R-squared: 0.914
F-statistic: 181.7 on 2 and 32 DF, p-value: < 2.2e-16

There appears to be a particularly strong relationship here, but let us have a look at some diagnostics.

An essential tool for assessing model fit, as we saw in the previous example, is the set of residuals. Residuals can be informative; they are not independent--they usually sum to zero--and they do not generally have the same variance. One useful metric is the *leverage* of each observation. Leverage is a characteristic of the individual observation, and determines to what degree an observation influences the fitted model. High-leverage observations will cause the corresponding residual to have a lower variance. To compensate for this, it is useful to standardise the residuals by rescaling them to have unit variance:

$$e'_i = \frac{e_i}{s\sqrt{1-h_{ii}}}$$

Here, s is the square root of the residual mean square and h is the leverage of observation i .

Calling `plot` on the linear model object generates a number of diagnostic plots; you need to hit return a few times to see each of the figures:

```
> plot(hills.lm)
Hit <Return> to see next plot:
```

This produces four diagnostic plots:

1. A plot of raw residuals on the fitted values from the model. A good fit produces a cluster of points along the horizontal at zero, with not apparent pattern.
2. A Q-Q plot of theoretical quantiles against standardised residuals. A good fit will have points aggregated along the diagonal, with no pattern.
3. The ‘Scale-Location’ plot, also called ‘Spread-Location’ or ‘S-L’ plot, takes the square root of the standardised residuals against the fitted values. A good fit produces an amorphous cluster of points.
4. The Residual-Leverage plot shows contours of equal Cook’s distance (a standardised measure of leverage) to give a sense of high-influence observations.

It is clear from these graphics that there are two points with large residuals, and two points with high leverage; one point is has both problems. We can see that the Knock Hill race is an outlier, with the model severely under-predicting the result. Using update again, we can re-fit the model without the outlier, by using the subset argument to remove the Knock Hill observation:

```
> update(hills.lm, subset=-18)

Call:
lm(formula = time ~ dist + climb, data = hills, subset = -18)

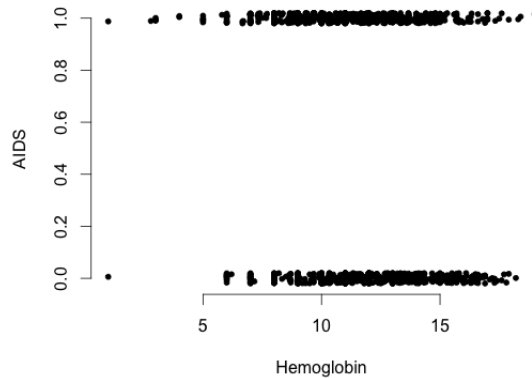
Coefficients:
(Intercept)          dist          climb
   -13.53035         6.36456         0.01185
```

Implementing logistic regression for binary outcomes

The standard way to model binary outcomes is via logistic regression. Binary outcomes are those which take on the values 0 or 1, where 1 defines the occurrence of some event and 0 is the non-occurrence of that event. We may wish to model such outcomes as a function of some other predictive variable. Here, for example, is the distribution of the onset of AIDS as a function of hemoglobin measurements:

```
> plot(haart$hemoglobin, jitter(haart$aids, 0.1), xlab="Hemoglobin",
+ ylab="AIDS", bty="n", pch=20)
```

(The jitter allows individual points to be seen more clearly)



Clearly, it would be unreasonable to fit a simple linear regression model to this data. Obviously, the errors are not going to be normally distributed. Rather than model 0 and 1 directly, we want to model the *probability* that $y=1$:

$$Pr(y_i = 1) = \text{logit}^{-1}(X\beta)$$

Though we no longer assume normal errors, we still assume the conditional independence of the outcomes. The logit function is defined as:

$$\text{logit}(p) = \log \left[\frac{p}{1-p} \right]$$

and hence:

$$\text{logit}^{-1} = \frac{\exp(x)}{1 + \exp(x)}$$

This link function serves to transform linear predictors defined on the real line into values that lie on the $(0,1)$ interval.

Since the logistic function (and its inverse) is non-linear, the change in p in response to a particular fixed change in x is not constant for all points in x . For example, the change from 0 to 0.4 in x corresponds to a change from 0.5 to 0.5 in p , while a change from 2.2 to 2.6 in x corresponds to just a change from 0.9 to 0.93 in p .

This is the logistic regression model. It specifies a sigmoid curve that is constrained to lie between zero and one, with an inflection point at $p=0.5$.

To fit a logistic regression in R, we use the `glm` function, which stands for *generalized* linear model. It is generalized because rather than assume normal errors, as in regular linear models, we generalize the model to allow for alternative choices. In this case, we have binomial error structure:

```
> model_fit <- glm(aids ~ hemoglobin, family=binomial, data=haart)
> summary(model_fit)
```

Call:

```
glm(formula = aids ~ hemoglobin, family = binomial, data = haart)
```

Deviance Residuals:

Min	1Q	Median	3Q	Max
-1.6860	-1.1617	-0.9027	1.1882	1.6129

Coefficients:

	Estimate	Std. Error	z value	Pr(> z)
(Intercept)	1.26330	0.22163	5.700	1.2e-08 ***
hemoglobin	-0.11822	0.01949	-6.067	1.3e-09 ***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

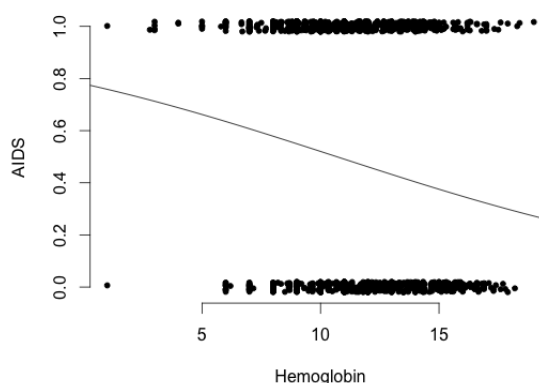
(Dispersion parameter for binomial family taken to be 1)

Null deviance: 2711.5 on 1956 degrees of freedom
Residual deviance: 2673.5 on 1955 degrees of freedom
(2673 observations deleted due to missingness)
AIC: 2677.5

Number of Fisher Scoring iterations: 4

To help us interpret the model output, we can plot the logistic regression line, by first extracting the coefficients from the model:

```
> betas <- model_fit$coefficients
> lines(x, invlogit(betas[1] + betas[2]*x))
```



Just as we had some intuition of the relationship from viewing the raw data, we see now that the probability of AIDS decreases with higher hemoglobin measurements.

The interpretation of the intercept for this model is meaningless, as it stands for a baseline value when the predictor's value is zero. However, since there are no individuals with zero hemoglobin, this is not a useful baseline. It often pays to scale predictors by subtracting the mean and dividing the standard deviation. In this case, fitting the model to scaled hemoglobin results in an estimate of the intercept of -0.056, which corresponds to a probability of AIDS of 48.6%. This can be interpreted as the predicted value for an individual with hemoglobin equal to the population mean.

As for the slope, the estimate of -0.118 indicates that an increase of 1 g/dL in hemoglobin results in a change of -0.118 in *logit* probability of developing AIDS. For

an individual with a near-average measurement of 10 g/dl, this would entail a change in probability from 0.520 to 0.491 (a difference of 0.029); for an individual with a low hemoglobin reading, such as 5 g/dl, this corresponds to a change from 0.662 to 0.635 (a difference of 0.027). This difference is not great, which you can see from the plot of the regression line -- it is approximately linear over the range of the observed data.

An alternative interpretation of the logistic regression slope coefficient is as an odds ratio. The odds is simply the ratio of the probability of an event occurring to the probability of it not occurring:

$$\text{OR} = \frac{p}{1-p}$$

Hence, the odds ratio for this model is $\exp(-0.118) = 0.889$. Unlike probability difference, the odds ratio does not change for different values of the predictor variables. For every unit increase in hemoglobin, the odds of the event occurring decrease by a factor of 0.889.

Survival analysis

Survival analysis is concerned with the distribution of lifetimes. Hence, rather than modeling the occurrence of events *per se*, it looks at the time until the occurrence of events to infer survival probabilities. Hence, we consider a random variable T to represent a failure event time that can take values on the continuous interval $(0, \infty)$, and the probability that $T=t$ can be modeled using the probability density $f(t)$ or the cumulative distribution $F(t)$.

Alternately, we may consider the complementary survivor function:

$$S(t) = 1 - F(t)$$

or the hazard function:

$$h(t) = \frac{\lim_{\Delta t \rightarrow 0} \Pr(t \leq T < t + \Delta t | T \geq t)}{\Delta t}$$

or even the cumulative hazard function:

$$H(t) = \int_0^t h(s) ds$$

These are three functions are inter-related by:

$$h(t) = \frac{f(t)}{S(t)}$$

and:

$$H(t) = -\log S(t)$$

One approach is to use a parametric distribution to describe survival, for example:

1) Exponential:

$$S(t) = \exp(-\lambda t), h(t) = \lambda$$

2) Weibull:

$$S(t) = \exp(-\lambda t)^\alpha, h(t) = \lambda \alpha (\lambda t)^{\alpha-1}$$

3) Log-logistic:

$$S(t) = \frac{1}{1 + (\lambda t)^\tau}, h(t) = \frac{\lambda \tau (\lambda t)^{\tau-1}}{1 + (\lambda t)^\tau}$$

What prevents us from simply modeling survival times directly, or regressing survival time on some predictive covariates, is censoring. Some individuals may not have been observed over their entire lifetime, but rather, only until the end of the study. Thus, the precise lifetime of such individuals is not known.

We are concerned with the mechanism with which censoring occurred; we would be concerned if several cases were removed from the study just prior to death, as it would bias our estimates of survival. For example, consider *right censoring*, which describes censoring times C_i such that we know T_i if $T_i \leq C_i$ or simply that $T_i > C_i$. If censoring is *random*, then T_i and C_i are independent random variables; in other words, the censoring time carries no information about the failure time.

If censoring is independent, we can write the likelihood as follows:

$$L = \prod_{\delta_i=1} f(t_i) \prod_{\delta_i=0} S(t_i) = \prod_{\delta_i=1} h(t_i) S(t_i) \prod_{\delta_i=0} S(t_i) = \prod_{i=1}^n h(t_i)^{\delta_i} S(t_i)$$

We are usually interested on how survival times vary by group, or according to one or more covariates.

R has a purpose-built data structure for handling survival data that contains the relevant information in the simplest format. The `Surv` class can be generated by the `Surv` function, and subsequently used as the response variable by a number of survival analysis procedures. In the case of our sample dataset, we need to include the followup time (`followup`), as well as `event`, which is death within the 365 day study period:

```
> haart_surv <- with(haart, Surv(followup, event))
> haart_surv[1:25]
[1] 365+ 365+ 365+ 43 365+ 365+ 365+ 365+ 365+ 87 254 365+ 365+ 365+
[19] 365+ 365+ 365+ 365+ 365+ 365+ 241+
```

The object that is returned contains the followup times, with a + after those observations that are censored. Hence, events are indicated by times that are not appended with +.

Survival curves

For non-censored data, the survival curve is estimated simply by the complement of the empirical distribution function $F(t)$. However, for censored data, we must account for the fact that the number of at-risk individuals decreases over time.

If we let $r(t)$ be the number of cases at risk just prior to time t , then an estimate of survival for the interval $[t_i, t_{i+1})$ is simply:

$$p_i = \frac{r(t_i) - d_i}{r(t_i)}$$

where d_i is the number of deaths in interval i . Hence, the probability of surviving until t_i is:

$$Pr(T > t_i) = S(t_i) \approx \prod_{j=0}^i \frac{r(t_j) - d_j}{r(t_j)}$$

This is the Kaplan-Meier (or product-limit) estimator.

Kaplan-Meier survival curves can be calculated using the `survfit` function in the `survival` package:

```
> haart_surv <- survfit(Surv(followup, event) ~ aids, data=haart)
> summary(haart_surv)
Call: survfit(formula = Surv(followup, event) ~ aids, data = haart)
```

67 observations deleted due to missingness

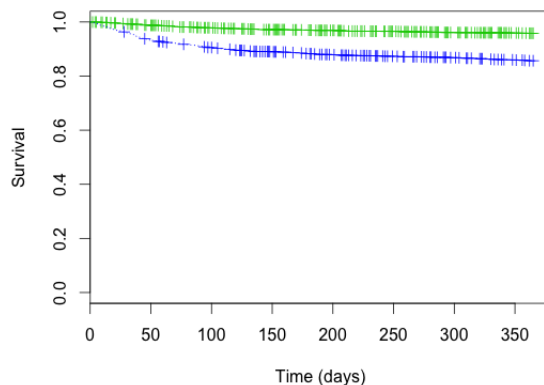
```
aids=0
time n.risk n.event survival std.err lower 95% CI upper 95% CI
1 2452 1 1.000 0.000408 0.999 1.000
6 2446 1 0.999 0.000577 0.998 1.000
7 2441 1 0.999 0.000707 0.997 1.000
11 2436 1 0.998 0.000817 0.997 1.000
12 2435 1 0.998 0.000914 0.996 1.000
```

...

```
aids=1
time n.risk n.event survival std.err lower 95% CI upper 95% CI
0 2039 1 1.000 0.000490 0.999 1.000
1 1996 3 0.998 0.000995 0.996 1.000
2 1992 2 0.997 0.001221 0.995 0.999
3 1989 1 0.997 0.001319 0.994 0.999
4 1988 1 0.996 0.001410 0.993 0.999
```

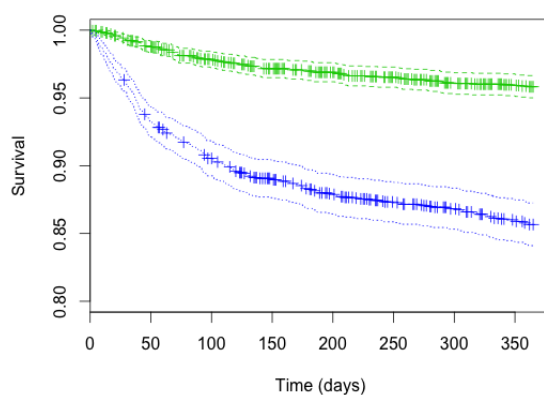
The resulting `survfit` object can be passed to `plot` to obtain Kaplan-Meier curves:

```
> plot(haart_surv, lty=2:3, col=3:4, xlab="Time (days)",
+ ylab="Survival")
```



By default, R plots the entire $[0,1]$ interval on the y-axis, which is not optimal for the range of the survival estimates here. We can re-scale the y-axis, and add confidence intervals as well:

```
> plot(haart_surv, lty=2:3, col=3:4, xlab="Time (days)",
+ ylab="Survival", ylim=c(0.8, 1), conf.int=TRUE)
```



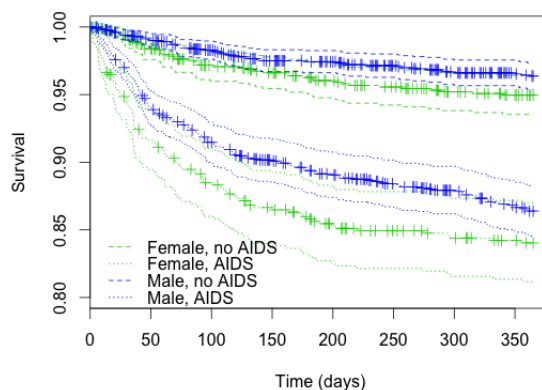
This shows the clear survival difference between the aids (blue) and non-aids (green) groups. It is possible to split the data on multiple factors, for example aids by male:

```
> haart_surv2 <- survfit(Surv(followup, event) ~ aids + male,
+ data=haart)
> plot(haart_surv2, lty=c(2,2,3,3), col=3:4, xlab="Time (days)",
+ ylab="Survival", ylim=c(0.8, 1), conf.int=TRUE)
```

This separates males from females, and aids from no aids based on color and line type, respectively. Of course, with four groups, a legend is helpful:

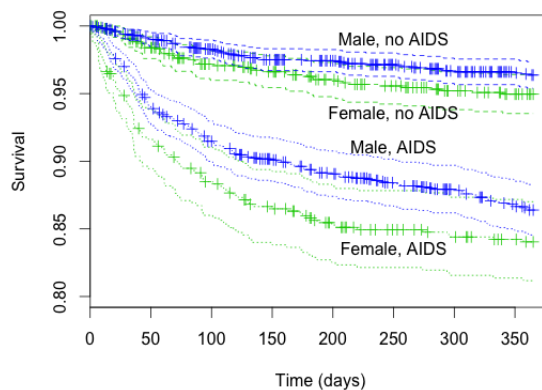
```
> legend(5, 0.85, c("Female, no AIDS", "Female, AIDS", "Male, no AIDS",
```

```
+ "Male, AIDS"), lty=c(2,3), col=c(3,3,4,4), bty="n")
```



Alternatively, since there is a small number of groups, this plot may be easier to read by labeling the lines directly, using the interactive `text` function:

```
> text(250, 0.99, "Male, no AIDS")
> text(250, 0.935, "Female, no AIDS")
> text(250, 0.91, "Male, AIDS")
> text(250, 0.835, "Female, AIDS")
```



Parametric survival models

Different survival models can be expressed by changing the form of the density function (and hence, the survival function), as we showed earlier in this section. The choice of an appropriate model will depend on how we believe survival changes according to any number of covariates, notably including time, age and risk factors. It is usually difficult to identify the best model, and so it is useful to try several models to see which error distribution seems most appropriate.

The simplest parametric model is the exponential, which simply specifies a hazard of $h(t) = \lambda > 0$. If we wish to relate the hazard to one or more covariates, we can

specify:

$$\lambda_i = \exp(\beta' x_i)$$

Where x_i is a vector of covariates. This is expressed as an exponential function to enforce the positivity constraint. For a Weibull model, this is:

$$h(t) = \lambda_i^\alpha \alpha t^{\alpha-1} = \alpha t^{\alpha-1} \exp(\alpha \beta' x_i)$$

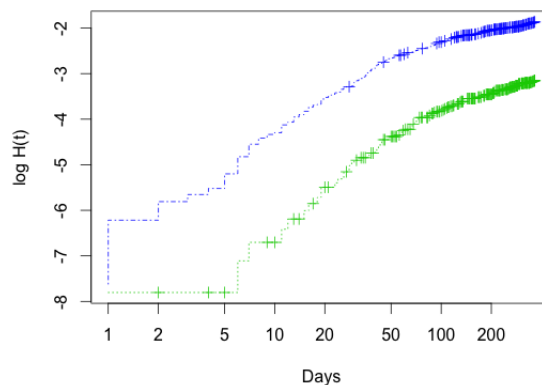
The general form of this model is:

$$h(t) = h_0(t) \exp(\beta' x)$$

Which is known as a proportional hazards model, since the hazards for an individual with covariate x is just a multiplicative factor of the baseline hazard $h_0(t)$.

A diagnostic test of the cumulative hazard assumption is the cumulative log-log plot:

```
> plot(haart_surv, lty = 3:4, col = 3:4, fun = "cloglog",
+ ylab="log H(t)", xlab="Days")
```



This shows that the hazards are roughly proportional, though not linear (which we would expect from a Weibull model).

The function `survreg` (survival regression) fits parametric survivals of the following form:

$$g(T) = \beta' x + \sigma \epsilon$$

where the link function g is usually the natural logarithm.

The error distribution of ϵ can be `weibull` (default), `exponential`, `rayleigh`, `lognormal` or `loglogistic` with a log link function, and `extreme`, `logistic`, `gaussian` or `t` with an identity link.

The scale is a fixed value greater than zero.

Running a parametric model of our sample dataset reveals a problem:

```
> survreg(Surv(followup, event) ~ aids, data=haart)
Error in survreg(Surv(followup, event) ~ aids, data = haart) :
  Invalid survival times for this distribution
```

The problem is, since the default distribution is set to `weibull`, an error is generated because of the associated log transformation of the followup times. There are a large number of zero values in the dataset, and the logarithm of zero is undefined. We have two options: either change the error distribution to one of the identity link function group, or remove the zeros from the dataset. Here is the first option:

```
> (fit_gaussian <- survreg(Surv(followup, event) ~ aids,
+ data=haart, dist="gaussian"))
Call:
survreg(formula = Surv(followup, event) ~ aids, data = haart,
        dist = "gaussian")

Coefficients:
(Intercept)          aids
    1145.6819    -305.6306

Scale= 459.8602

Loglik(model)= -3585    Loglik(intercept only)= -3658.1
      Chisq= 146.14 on 1 degrees of freedom, p= 0
n=4563 (67 observations deleted due to missingness)
```

and here is the second:

```
> (fit_weibull <- survreg(Surv(followup, event) ~ aids, data=haart,
+ subset=followup>0))
Call:
survreg(formula = Surv(followup, event) ~ aids, data =
        haart[haart$followup > 0, ])

Coefficients:
(Intercept)          aids
    11.282294    -2.235762

Scale= 1.708009

Loglik(model)= -3276.5    Loglik(intercept only)= -3347.5
      Chisq= 141.82 on 1 degrees of freedom, p= 0
n=4448 (64 observations deleted due to missingness)
```

Notice that the parameter estimates are vastly different; in order for parameters to be interpretable, they must be divided by the scale factor. Also, the Weibull model describes time on the log scale. Moreover, we have dropped 118 zero followup values from the Weibull model, so they are using slightly different datasets.

The Cox model

A less parametric approach to modeling proportional hazards saves us from having to specify a distribution for the baseline hazard function (we are more interested in

the proportional factors anyway). The Cox model estimates the parameter vector by maximizing a partial likelihood. If we assume that *one death* occurred at time t , then the probability that it is individual i that dies is:

$$\frac{h_0(t) \exp(\beta' x_i)}{\sum_j I(T_j \geq t) h_0(t) \exp(\beta' x_j)} = \frac{\exp(\beta' x_i)}{\sum_j I(T_j \geq t) \exp(\beta' x_j)}$$

Since the baseline hazard cancels out, we do not have to specify it. This approach is semi-parametric because it only relies on the *ranks* of the survival times.

While the parametric model asks “how will the expected survival time increase or decrease with the covariate?”, the Cox model asks “how much does the risk of dying increase or decrease with the covariate?”. The Cox model is more flexible, and requires fewer assumptions, but if a parametric assumptions are valid, it results in more precise inference.

The Cox model, rather than modeling survival time, models the hazard rate:

$$\log h_i(t) = \log h_0 + \beta' x_i$$

Applying the Cox model to our sample data yields:

```
> summary(coxph(Surv(followup, event) ~ aids, data=haart))
Call:
coxph(formula = Surv(followup, event) ~ aids, data = haart)

n= 4563, number of events= 371
(67 observations deleted due to missingness)

      coef exp(coef) se(coef)      z Pr(>|z|)
aids 1.3073    3.6961  0.1186 11.03  <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

      exp(coef) exp(-coef) lower .95 upper .95
aids      3.696      0.2706      2.93      4.663

Concordance= 0.654 (se = 0.013 )
Rsquare= 0.031 (max possible= 0.741 )
Likelihood ratio test= 141.7 on 1 df,  p=0
Wald test               = 121.6 on 1 df,  p=0
Score (logrank) test = 139.9 on 1 df,  p=0
```

Thus, the estimated hazard of the aids group is $\exp(1.3073) = 3.7$ times that of the non-aids group, with a confidence interval of (2.93, 4.66). The tests at the bottom of the output are asymptotically-equivalent tests of the null hypothesis that the regression parameters are equal to zero.