An Introduction to the R Environment

Peter Dalgaard

Department of Biostatistics
University of Copenhagen

FMS Workshop, Uppsala, May 2005

Outline

About the Talk
Basics of R
Modeling
The Package System
Some Practical Issues
Graphics
Programming

Practicalities

▶ Short introduction (approx. 2hr)
▶ High coverage, not great depth
▶ More in-depth discussions will follow in the afternoon
▶ Short break (5–10 minutes) in the middle.
▶ Script of demos in
http://www.biostat.ku.dk/~pd/Uppsala-demos.R

Plan

▶ Elementary things about R, simple demo
▶ Modeling tools
▶ R packages
▶ Dealing with the R workspace
▶ Graphics in R
▶ Programming
The R environment

- Built around the programming language R, an Open Source dialect of the S language
- R is Free Software, and runs on a variety of platforms (I'll be using Linux here, mainly to avoid technical surprises).
- Command-line execution based on function calls
- Extensible with user functions
- Workspace containing data and functions
- Various graphics devices (interactive and non-interactive)

R is a vectorized language

- The basic data type in R is a vector
- Vectors often represent data (e.g. the age for each participant in a study), but also other things like regression coefficients.
- Numeric (integer/double), character (strings), logical (TRUE/FALSE)
- Factor (really integer + level attribute) for categorical variables
- Lists (generic vectors)

Basic operations

- Standard arithmetic is vectorized (x + y, etc.)
- Recycling: If operating on two vectors of different length, the shorter one is replicated (with warning if it is not an even multiple)
- sum, mean, range, ...
- Constructor functions: c (concatenation), seq (sequences), rep (replication), ...

Demo 1

```r
x <- round(rnorm(10,mean=20,sd=5)) # simulate data
x
mean(x)
m <- mean(x)
x - m # notice recycling
(x - m)^2
sum((x - m)^2)
sqrt(sum((x - m)^2)/9)
sd(x)
```
Extended data types

- The basic vector types can be combined and extended to form more complex data structures
- Attributes extend a basic type with further information. E.g., a vector can have a names attribute, for more readable printing
- Classes have two main functions:
  - Hide details
  - Allow function dispatch (functions that behave differently depending on the class)

Smart indexing

- a[5] single element
- a[c(5, 6, 7)] several elements
- a[-6] all except the 6th
- a[b>200] index by logical vector
- a["name"] by name

Matrices/tables/arrays

- Used in matrix calculus and as input to, e.g., chisq.test(). Results of tabulation.
- Vectors with dimension attribute
- Generate with matrix, array, table
- Indexing methods include [i,j], [i,], [,j] (the latter two extract entire rows/columns)

Data frames

- Like data set in other packages
- Technically: Lists of vectors/factors of same length
- Row names (must be unique)
- Indexed like matrices (Beware, though: Data frames are not matrices)
- Generate from read operation or with data.frame
- Many sample data frames are available using data()
### Demo 2

```r
data(airquality)
airquality$Month
airquality[airquality$Month==5,]$Ozone
oz <- airquality[airquality$Month==5,]$Ozone
mean(oz)
mean(oz, na.rm=TRUE)
attach(airquality)
mean(Ozone, na.rm=TRUE)
tapply(Ozone, Month, mean, na.rm=TRUE)
detach()
```

### Some standard procedures

- **Continuous data by group:** `t.test`, `wilcox.test`, `oneway.test`, `kruskal.test`
- **Categorical data:** `prop.test`, `chisq.test`, `fisher.test`
- **Correlations:** `cor.test`, with options for nonparametrics

### Demo 3

```r
library(ISwR)
data(intake)
attach(intake)
t.test(pre, post, paired=TRUE)
```

### Modeling Tools: Overview

- **Model formulas**
- **Model objects and summaries**
- **Comparing models**
- **Evaluating model fit**
- **Generalized linear models**
Model formulas

- Linear model, \( y = X\beta + \epsilon \)
- In practice something like
  \[ y = \beta_0 + \beta_1 \times \text{height} + \beta_2 \times 1_{(\text{type}=2)} + \beta_3 \times 1_{(\text{type}=3)} + \epsilon \]
- Wilkinson-Rogers formulas:
  \[ y = \text{height} + \text{type} \]

(Interpretation depends on whether variables are categorical or continuous)

Model formulas in R

- R representation \( y \sim \text{height} + \text{type} \) where \text{type} is a factor
- Interactions \( a:b, a*b = a + b + a:b \)
- Algebra \( a:(b + c) = a:b + a:c \) etc.
- Notice special interpretation of operators

Fitting linear models

```r
data(airquality)
aq <- transform(airquality, Month=factor(Month))
fit.aq <- lm(log(Ozone) ~ Solar.R + Wind + Temp + Month, data=aq)
```

- \( \text{lm} \) generates a fitted model object
- Extract information from model object
- Fit other models based on model object

Inspecting model objects

- Extract information about the fit
  ```r
  summary(fit.aq)
  fitted(fit.aq), resid(fit.aq)
  anova(model1, model2)
  plot(fit.aq) – diagnostics
  predict(fit.aq, newdata)
  ```
Model search

- `anova(model)` “Type I” sum of squares
- `drop1` (“Type III”)
- `step` (AIC/BIC) criteria
- `update`
R packages

- Collections of R functions, data, and compiled code
- Well-defined format that ensures easy installation, a basic standard of documentation, and enhances portability and reliability,
- You can write your own packages! It is not entirely trivial, but tools are there to help you.

Packages that come with R

- Standard R (2.3.0) loads with several packages available, covering the basic methodology (this is customizable (?Startup), but you only want less than these in special applications)
- Further packages are available in core R, but not automatically loaded (tcltk, grid, splines, stats4...)
- 10 further packages and package bundles are maintained separately, but included with source and binary distributions (survival, nlme, MASS, ...).

CRAN

- The Comprehensive R Archive Network
- Collection of servers mirroring a central server in Vienna. Modeled on CTAN and CPAN (for TEx and Perl code)
- http://cran.us.r-project.org
- Maintains a curated collection of R packages as well as the source and binary distributions R itself
- Over 700 packages available
- Unix/Linux variants generally install packages from sources. Windows and MacOSX have binary package formats which are even easier to install
- See also: Bioconductor,
  http://www.bioconductor.org

Demo 6

# Cheat for offline demo:
# Pretend CRAN is local directory
options(repos=c(CRAN="file:/home/pd/cran.r-project.org"))
# Manipulate install path
.libPaths("~/.Rlibrary")
.libPaths()
# Source install (gives harmless warning)
install.packages("mvtnorm")
library(mvtnorm)
library(help=mvtnorm)
Practical issues

- Dealing with the workspace
- Reading data
- Saving and restoring data and results

The workspace

- The global environment contains R objects created on the command line.
- There is an additional search path of loaded packages and attached data frames.
- The search path is maintained by library(), attach(), and detach().
- This determines the way R looks up objects by name.
- Notice that objects in the global environment may mask objects in packages.

Reading data

- Simple data vectors can be read using scan().
- Data frames can be read from most reasonably structured text file formats (space separated columns, tab- and comma-delimited files) using read.table() or read.delim(). Note colClasses.
- The foreign package can read files from Stata, SAS export libraries, SPSS, and Epi-Info, Minitab, and some S-PLUS versions.
- For spreadsheets and databases, the quick and easy way is to export to a delimited file, but you can work via ODBC connections and database access packages.

Getting organized

Several possibilities:

- Save/restore entire workspace (objects only)
- Save selected objects and load them
- source() script files
- Batch processing (R CMD BATCH file.R)
R graphics

- The standard interface
- Customizing plots
- Graphics parameters
- Math on plots
- Grid and lattice

Standard R graphics

- Ink on paper model; once something is drawn it cannot be erased.
- Sensible default plots
- Arguments can override defaults
- Options to turn off various elements of plots (e.g. the axes)
- Functions to add elements.

Basic x-y plots

- The plot function with one or two numeric arguments
- Scatterplot or line plot (or both) depending on type argument: "l" for lines, "p" for points (the default), "b" for both, plus quite a few more.
- Functions for adding to a plot: lines, points, segments, abline, text, mtext, axis
- Also: formula interface, plot(y~x)

Graphical parameters

- Arguments to plot et al. (67 possibilities!)
- The par function can be used to set most of them persistently. Most info is found via help(par)
- Look them up! Here are some of the more commonly used:
  - Point and line characteristics: pch, col, lty, lwd
  - Multiframe layout: mfrow, mfcol
  - Axes: xlim, ylim, xaxt, yaxt, log
Specific plots

- **Histograms** — `hist(x)`
- **Density plots** — `plot(density(x))`
- **Boxplots** — `boxplot(x)`
- **Barplots** — `barplot(x)` (x can be a matrix)
- **Pies** — `pie()`
- **Matrix plots (multiple y columns)** — `matplot()`

Math on plots

- Sort of like TeX
- Works on unevaluated expressions (quote(alpha), expression(alpha))
- Special conventions: `^`, `[]` sub/superscript, special names alpha, sum, int
- See `help(plotmath)`

Grid and Lattice graphics

- Standard R graphics allow graphs to be arranged in an $m \times n$ gridded layout.
- The `grid` package allows arbitrary viewports and create graph objects ("grobs") which can be modified before they are printed.
- The `lattice` package uses grid for a structural approach to multiframe graphs
- Model formulas, $y \sim x | g_1 \times g_2 \times \ldots$
- Shingles: Partially overlapping intervals used for conditioning plots
- Panel functions — potentially user codable

Demo 7

```r
data(intake)
par(mfrow=c(2,2))
matplot(intake)
matplot(t(intake))
matplot(t(intake), type="b")
matplot(t(intake), type="b", pch=1:11, col="black", lty="solid", xaxt="n")
axis(1, at=1:2, labels=names(intake))
```
library(lattice)

data(airquality)

lset(theme = col.whitebg())

myplot <-
  xyplot(log(Ozone) ~ Solar.R | equal.count(Temp),
         group=Month, data=airquality,
         ylab=list(label=expression("log"*O[3]), cex=2),
         xlab=list(cex=2))

myplot # OBS: no plot until object is printed!

---

### Demo 8

```r
library(lattice)
data(airquality)
lset(theme = col.whitebg())
myplot <-
  xyplot(log(Ozone) ~ Solar.R | equal.count(Temp),
         group=Month, data=airquality,
         ylab=list(label=expression("log"*O[3]), cex=2),
         xlab=list(cex=2))
myplot # OBS: no plot until object is printed!
```

### Programming

- With experience, you will soon be writing your own R functions
- Mostly, for ad-hoc tasks, collecting common functionality
- Or, because they are required input for certain tasks (e.g., optimizers!)
- R is a full programming language with mathematical functionality
  - Flow control
  - Scoping, local variables
  - Matrix algebra
- Expressions inside functions work just like on the command line, except that the result is not printed.
- User-written functions are not substantially different from system functions, making R very smoothly extensible.

---

### Simple functions

- logit <- function(p) log(p/(1-p))
- logit(0.5)
- Formal arguments
- Actual arguments
- Positional matching: plot(x,y)
- Keyword matching: t.test(x ~ g, mu=2, alternative="less")
- Partial matching: t.test(x ~ g, mu=2, alt="l")

---

### Summary

- So what have we seen?
  - R is a versatile working environment
  - There is a very flexible toolkit for building graphics displays
  - You can handle simple tasks quite easily
  - Complicated task can be handled via ad hoc programming, often elegantly
  - Extensions can be made to integrate seamlessly and a large body of such extensions is available from CRAN
Outline

An Introduction to the R Environment

R Basics

Peter Dalgaard

Department of Biostatistics
University of Copenhagen

FMS Workshop, Uppsala, May 2006

Basics of R

Key Points about R

- Environment built around the programming language R, (an Open Source dialect of the S language).
- R is Free Software, and runs on a variety of platforms (I'll be using Linux. Computer labs run on Windows.)
- Command-line execution based on function calls
- Extensible with user functions
- Workspace containing data and functions
- Graphics devices

- What is R?
- Interacting with R
- Extended user interfaces
- Later: Dealing with R's workspace
Interacting with R

- Command line interface (CLI)
  - The basic mode of interaction is “read – evaluate – print”
  - User types an expression at the command line,
  - R evaluates it
  - And prints the result
  - Batch variation: read commands from a file

Extended Interfaces

- Windows, Macintosh GUI: Fairly simple extensions of CLI, mostly offloads some tasks to menu interface, and adds command recall
- **Script editing**: The ability to work with multiple lines of R code, save them to a file for later use, etc. A simple script editor is built into the R GUI in recent versions.
- R embedded in a text editor (ESS – Emacs Speaks Statistics). Popular on Unix/Linux systems.

Demo 1

- 2+2
- log(10)
- help(log)
- summary(airquality)
- demo(graphics) # pretty pictures...

Basic Vector Types

- R is a *vector based* language, data types include
  - Numeric (integer,double) vectors
  - Character (strings) vectors
  - Logical vectors
- These types are combined and extended to form more complex objects
Basic operations

- Standard arithmetic is \textit{vectorized}: \(x + y\) adds each element of \(x\) to the corresponding element of \(y\).
- Recycling: If operating on two vectors of different length, the shorter one is replicated (with warning if it is not an even multiple).
- \texttt{c} — concatenate: \(c(7, 9, 13)\)
- \texttt{seq} — sequences: \(\text{seq}(1, 9, 2)\), short form: \(1:5\) is the same as \(\text{seq}(1,5,1)\)
- \texttt{rep} — replication \(\text{rep}(1:3, 3:1)\) (1 1 1 2 2 3)
- \texttt{sum}, \texttt{mean}, \texttt{range}, ...

Attributes

- Attributes extend the basic vector types in various ways
- \texttt{attributes(x)} shows them
- Names, set with \texttt{names(x) <- c("Huey", "Dewey", "Louie")}
- Dimensions (\texttt{dim()})
- Dimnames
- Classes (S3)

Classed Objects

- In R objects can have \textit{classes}
- These are used as the basis for \textit{function dispatch}
- I.e. the same (\textit{generic}) function can have different methods for different classes
- Print methods are a prototypical example
- There are two object systems, based (roughly) on S version 3 and version 4. We will not go into details.

Factors

- Factors are used to describe groupings (the term originates from \textit{factorial designs})
- Basically, these are just integer codes plus a set of names for the \textit{levels}
- They have class "\textit{factor}" making them (a) print nicely and (b) maintain consistency
- A factor can also be \textit{ordered} (class "\textit{ordered}"), signifying that there is a natural sort order on the levels
- In model specifications, factors play a fundamental role by indicating that a variable should be treated as a classification rather than as a quantitative variable (similar to a CLASS statement in SAS)
Smart Indexing

- R has several unusual but highly useful indexing mechanisms:
  - `a[5]` single element
  - `a[5:7]` several elements
  - `a[-6]` all except the 6th
  - `a[b>200]` logical index
  - `a["name"]` by name

Lists

- A vector where the elements can have different types
- Functions often return lists
  - `lst <- list(A=rnorm(5), B="hello")`
- Special indexing:
  - `lst$A` first element
  - `lst[[1]]` list containing the first element

Matrices/Tables/Arrays

- Used in matrix calculus and as input to, e.g., `chisq.test()`. Results of tabulation.
- Vectors with dimensions
- Dimnames can be added for nicer printing
- Matrices: Generate with `matrix`
- Indexing methods include `[i,j], [i,], [,j]

Data frames

- Like data set in other packages
- Technically: Lists of vectors/factors of same length
- Row names (must be unique)
- Indexed like matrices (Beware, though: Data frames are not matrices)
- Generate from read operation or with `data.frame`
- Many sample data frames are available using `data()`
The workspace

- **The global environment** contains R objects created on the command line.
- There is an additional *search path* of loaded packages and attached data frames.
- When you request an object by name, R looks first in the global environment, and if it doesn’t find it there, it continues along the search path.
- The search path is maintained by `library()`, `attach()`, and `detach()`.
- Notice that objects in the global environment may mask objects in packages and attached data frames.

Demo 2

```r
attach(airquality)
mean(Ozone, na.rm=TRUE)
tapply(Ozone, Month, mean, na.rm=TRUE)
detach()
search()
library(ISwR)
data(intake)  # From ISwR
ls()
attach(intake)
search()
ls("intake")  # show variables in data frame
post - pre
rm(intake)  # remove data frame
detach()  # remove from search path
```

A Common Mistake

```r
attach(mydata)
sex <- factor(sex)
tapply(height, sex, mean)
detach()
attach(subset(mydata, age > 25))
sex <- factor(sex)
tapply(height, sex, mean)
```

You get an error saying that `height` and `sex` are of different length. What went wrong?
Second time around, `sex` was found in the global environment *before* the attached data frame.

Getting Organized

Several possibilities:
- Save/restore entire workspace (objects only)
- Save selected objects and load them
- `source()` script files
- Batch processing (`R CMD BATCH file.R`)
Reading Data, Overview

- Simple data vectors can be read using `scan()`.
- Data frames can be read from most reasonably structured text file formats (space separated columns, tab- and comma-delimited files) using `read.table()` or `read.delim()`.
- The `foreign` package can read files from Stata, SAS export libraries, SPSS, and Epi-Info, Minitab, and some S-PLUS versions.
- For spreadsheets and databases, the quick and easy way is to export to a delimited file, but you can work via ODBC connections and database access packages.

The Simplest Way to Read Data

- This is what you’d normally want to do:
- Have data in a plain text file
- Columns separated by whitespace
- Missing values coded as the string "NA"
- Preferably have a row of variable names at the top
- Use `d <- read.table("myfile", header=TRUE)`

Demo 3

```r
dir <- system.file("data", package="ISwR")
fname <- file.path(dir, "thuesen.txt")
fname

file.show(fname)
read.table(fname, header=TRUE)
```

(Notice the use of portable constructs to find the data directory inside a package and the construction of the full pathname.)

Options and Details

- `read.table` has quite a few options and details
- Different codings of missing values (`na.strings`)
- Different decimal separators (`dec` argument)
- Text strings can be quoted if embedded blanks
- You may skip lines, read a limited number of lines, and more. Please consult the manual page for details.
Delimited Files

- These are conveniently obtained via export or cut-and-paste from standard applications—generally given .DAT or .CSV extensions on Windows
- Data columns separated by separator character (exactly one, so that fields can be empty)
- Typically TAB, comma, or semicolon separated
- Notice that the files are locale-dependent due to the number format; CSV format also switches the separator from comma to semicolon (or else . . .)
- There are precooked variations of \texttt{read.table} that sets the proper options: \texttt{read.csv}, \texttt{read.csv2}, \texttt{read.delim}, \texttt{read.delim2}

Column Input

- Also known as fixed-width format (somewhat dated, but still being used)
- The \texttt{read.fwf} function handles this
- The \texttt{widths} argument is a vector of field widths (negative numbers flags columns that should be skipped)
- For multiline records, you give a list of vectors. Notice that the lines are concatenated prior to the splitting into fields, so the widths had better add up correctly.

Special Considerations

- The \texttt{read.table} variations do a fair amount of guesswork trying to do the right thing with the data
- However, this can be computationally demanding
- and sometimes, the guess is wrong!
- In particular, watch out for automatic conversion to factor type
- Some items require additional information. E.g. dates and times are more likely than not in a nonstandard format.

Converting Data after Reading

- Notice the \texttt{as.is} and \texttt{colClasses} arguments which are useful to specify columns that should be left alone or converted to a specific data class.
- However, you cannot specify everything. Sometimes you need to postprocess.
- Most frequently, data have numeric columns that are really group codes. Use \texttt{factor} to convert them and put labels on them.
- Dates are handled conveniently using \texttt{as.Date} which converts from character and (per R-2.1.1) factor variables.
Improving Efficiency

- Sometimes you run into resource limitations, or reading becomes slow
- Here are some things to consider:
  - `read.table` reads everything as character vectors, then decides how to store it. Character vectors take up a lot of space.
  - If you know the column types, the `scan` function can be much more efficient.
  - Also, both `scan` and `read.table` allow you to skip columns entirely.

When the Going Gets Tough...

- Sometimes you have to deal with irregular data formats, perhaps from a measurement device.
- You can read an entire file or parts of it as a character vector, using `readLines`
- Once the file is read, you can process it with string handling functions like `gsub` or `substr`
- Finally, read from a `textConnection` to the modified vector.

Data Manipulation Functions

- Constructors of simple objects
- Single-column modifications
- Modifying and subsetting data frames

Constructors

- R deals with many kinds of objects besides data sets
- Need to have ways of constructing them from the command line
- We have (briefly) seen the `c` and `list` functions
- Notice the naming forms `c(boys=1.2, girls=1.1)`
- Extracting and setting names with `names(x)`
- For matrices and arrays, use the (surprise) `matrix` and `array` functions. `data.frame` for data frames.
- It is also fairly common to construct a matrix from its columns using `cbind`
The **factor** Function

- This is typically used when `read.table` gets it wrong
- E.g. group codes read as numeric
- Or read as factors, but with levels in the wrong order (e.g. `c("rare", "medium", "well-done")` sorted alphabetically)
- Notice the slightly confusing use of `levels` and `labels` arguments.
- `levels` are the value codes *on input*
- `labels` are the value codes *on output* (and become the levels of the resulting factor)

The **cut** Function

- The cut function converts a numerical variable into groups according to a set of break points
- Notice that the number of breaks is one more than the number of intervals
- Notice also that the intervals are left-open, right-closed by default (`right=FALSE` changes that)
- ... and that the lowest endpoint is *not* included by default (set `include.lowest=TRUE` if it bothers you)

Working with Dates

- Dates are usually read as character or factor variables
- Use the `as.Date` function to convert them to objects of class "Date"
- If data are not in a standard format (YYYY-MM-DD) you need to supply a format specification
  > `as.Date("11/3-1959",format="%d/%m-%Y")`
  > `1959-03-11`
- You can calculate differences between codeDate objects. The result is an object of class "difftime", with a unit of days. You need `as.numeric` to get the actual number.

Sorting Things

- Sorting is not used quite as much in R as in other packages, because few procedures rely on presorted data.
- However, it is easy enough: `sort(x)`
- To put `y` in the order of `x`: `y[order(x)]`
- or to sort an entire data frame `mydata[order(sex,age),]`
- Notice that the semantics of the `order` function. It is *not* the same as ranking, rather its complement.
Modifying and Subsetting Data Frames

- The syntax for indexing data frames easily gets heavy:
  ```r
  airquality[airquality$Month == 5 &
  airquality$Ozone > 50,]
  ```

- The `subset` function uses nonstandard evaluation to allow you to say `subset(airquality, Month == 5 &
  Ozone > 50)`. I.e., it evaluates the second argument within the data frame.

- The `transform` function is similar. It allows you to define new variables or modify old ones using code like:
  ```r
  juulnew <- transform(juul,
    sex=factor(sex, labels=c("M","F")),
    tanner=factor(tanner))
  ```
An Introduction to the R Environment

R Graphics

Peter Dalgaard

Department of Biostatistics
University of Copenhagen

FMS Workshop, Uppsala, May 2006

Outline

Overview

Standard R Graphics

▶ The standard interface
▶ Customizing plots
▶ Graphics parameters
▶ Math on plots
▶ Grid and lattice graphics

Standard Graphics

▶ Ink on paper model; once something is drawn it cannot be erased.
▶ Sensible default plots
▶ Arguments can override defaults
▶ Options to turn off various elements of plots (e.g. the axes)
▶ Functions to add elements.
Types of Plotting Functions

- **High level**
  - Create a new page of plots with reasonable default appearance.

- **Low level**
  - Draw elements of a plot on an existing page:
    - Draw title, subtitle, axes, legend...
    - Add points, lines, text, math expressions...

- **Interactive**
  - Querying mouse position (locator), highlighting points (identify)

Managing graphics devices

- You can have several graphics devices open at the same time: `dev.list()` shows them
- Graphics output only goes to the **current** device: `dev.cur()`
- Turn off a graphics device with `dev.off()`
- Print the current plot: `dev.print()`
- Make a copy of the current plot using another device: `dev.copy()`, `dev.copy2eps`
- However, best results are obtained by plotting directly to the target device

The `plot()` Function

A **generic function**: does the right thing based on the class of its arguments

- Plot sequential values of a numeric variable:
  \[ x \leftarrow \text{rnorm}(x); \text{plot}(x) \]
- Bar chart for factors: `plot(cut(x, 5))`
- Time series plot: `plot(ts(x))`
- Density plot: `plot(density(x))`
- Two-way plots: `plot(y \sim x)`
  - Scatterplot if \( y, x \) are both numeric
  - Series of boxplots if \( x \) is a factor

Further High-level Plotting Functions

- **Histogram**: `hist(x)`
- **Boxplot**: `boxplot(x)`
- **Barplot**: `barplot(x)` (\( x \) can be a matrix)
- **Pie chart**: `pie(x)`
  - Not encouraged: use barplot or dotchart instead
  - You will never see a 3-D exploded pie chart in R!
- **Plot multiple variables**: `matplot(x)` (\( x \) is a matrix)
Basic x-y Plots

- The `plot` function with one or two numeric arguments
- Scatterplot or line plot (or both) depending on `type` argument: "l" for lines, "p" for points (the default), "b" for both, plus quite a few more
- Also: formula interface, `plot(y~x)`, with arguments similar to the modeling functions like `lm`

Customizing Plots

- Most plotting functions take optional parameters to change the appearance of the plot
- Most of these parameters can be supplied to the `par()` function, which changes the default behaviour of subsequent plotting functions
- Look them up via `help(par)`! Here are some of the more commonly used:
  - Point and line characteristics: `pch`, `col`, `lty`, `lwd`
  - Multiframe layout: `mfrow`, `mfcol`
  - Axes: `xlim`, `ylim`, `xaxt`, `yaxt`, `log`

Adding to Plots

- `points()`, `lines()` adds points and (poly-)lines
- `text()` text strings at given coordinates
- `mtext()` margin text. Given in plotting coordinates along one edge and `lines` in the perpendicular direction.
- `abline()` line given by coefficients \(a\) and \(b\) or by fitted linear model
- `axis()` adds an axis to one edge of the plot region. Allows some options not otherwise available.

Approach to Customization

- Start with something that looks nearly right
- Modify parameters (using `par()` settings or plotting arguments)
- Add more graphics elements. Notice that there are graphics parameters that turn things off, e.g. `plot(x, y, xaxt="n")` so that you can add completely customized axes.
- Multiframe layouts
- In emergency: overplot using `par(new=TRUE)}`
Demo 1

```r
data(intake)
par(mfrow=c(2,2))
matplot(intake)
matplot(t(intake))
matplot(t(intake),type="b")
matplot(t(intake),type="b",pch=1:11,col="black",
       lty="solid", xaxt="n")
axis(1,at=1:2,labels=names(intake))
```

Margins

- R sometimes seems to leave too much empty space around plots
- There is a good reason for it: You might want to put something there (titles, axes).
- This is controlled by the `mar` parameter. By default, it is `c(5,4,4,2)+0.1`
- The units are *lines of text*, so depend on the setting of `pointsize` and `cex`
- The `mtext` function is designed to write in the margins of the plot
- There is also an *outer margin* settable via the `oma` parameter. Useful for adding overall titles etc. to multiframe plots

Demo 2

```r
x <- runif(50,0,2)
y <- runif(50,0,2)
plot(x, y, main="Main title", sub="subtitle",
     xlab="x-label", ylab="y-label")
text(0.6,0.6,"text at (0.6,0.6)")
abline(h=.6,v=.6)
for (side in 1:4) mtext(-1:4,side=side,at=.7,line=-1:4)
mtext(paste("side",1:4), side=1:4, line=-1,font=2)
```
Math on Plots

- Sort of like TeX
- Works on unevaluated expressions (quote(alpha), expression(alpha))
- Special conventions: `^`, `[ ]` sub/superscript, special names `alpha, sum, int`
- See `help(plotmath)` and `demo(plotmath)`
- Manipulating the unevaluated expressions (“Computing on the Language”) is useful, albeit somewhat tricky.

Grid and Lattice Graphics

- Standard R graphics allow graphs to be arranged in an $m \times n$ gridded layout.
- The `grid` package allows arbitrary *viewports* and create *graph objects* (“grobs”) which can be modified before they are printed.
- The `lattice` package uses `grid` for a structural approach to multiframe graphs
- This is mostly compatible with S-PLUS’s Trellis graphics
- Model formulas, `y~x|g1*g2*...
- Shingles: Partially overlapping intervals used for conditioning plots

Demo 3

```r
y <- rnorm(25)
curve(dnorm(x, mean(y), sd(y)), from=-3, to=3)
rug(y)
abline(h=0)
title(main=substitute(paste(mu==m, " ", sigma==s),
list(m=mean(y), s=sd(y) )))
```

Demo 4

```r
library(lattice)
trellis.par.set(theme = col.whitebg())
myplot <-
  xyplot(log(Ozone)~Solar.R | equal.count(Temp),
         group=Month, data=airquality,
         ylab=list(label=expression("log"*O[3]), cex=2),
         xlab=list(cex=2))
myplot # OBS: no plot until object is printed!
```
Panel Functions

- What goes inside each panel of a Lattice plot is controlled by a *panel function*
- There is a number of standard functions: `panel.xyplot`, `panel.lmline`, etc. (38 of them, currently)
- You can write your own panel functions, most often by combining some of the standard ones

```r
xyplot(log(Ozone) ~ Solar.R | equal.count(Temp), ......
  panel=function(x,y,...){
    panel.xyplot(x,y,...)
    panel.lmline(x,y,type="l")
  }
)```

Simple Descriptives

- mean, median, sd, etc.
- quantile(x, p) where p is a vector of proportions
- (actually, there is nine different types of quantiles)
- summary gives some key quantities or a variable, depending on its type. This also works on entire data frames

Tabulation

- For simple tables of discrete variables, use the table function, as in `table(sex, tanner)`
- For tables of descriptives the first choice is `tapply`, for example `tapply(age, tanner, mean, na.rm=TRUE`)
- Explanation: age is split according to groups and mean is called on each piece with an extra argument, evaluating mean(age, na.rm=TRUE) within each group.
Neater Tables

- Some variations over `tapply` is given by the `by` and `aggregate` functions.
- Multiway tables are often hard to read and use for presentation purposes. Look into the `ftable` (“flattened tables) and Martyn Plummer’s `stat.table` function in the `Epi` package.

Some Standard Tests

- Continuous data by group: `t.test`, `wilcox.test`, `oneway.test`, `kruskal.test`
- Categorical data: `prop.test`, `chisq.test`, `fisher.test`
- Correlations: `cor.test`, with options for nonparametrics

Demo 1

```r
library(ISwR)
data(intake)
attach(intake)
t.test(pre, post, paired=TRUE)
detach()
```

Demo 2

```r
data(caesarean) # loads a table
cesar.shoe
chisq.test(caesar.shoe)
fisher.test(caesar.shoe)
x <- caesar.shoe[1,]
n <- margin.table(caesar.shoe,2)
n
prop.trend.test(x,n)
```
Modeling Tools: Overview

- Model formulas
- Model objects and summaries
- Comparing models
- Evaluating model fit
- Generalized linear models

Model Formulas

- Linear model, \( y = X\beta + \epsilon \)
- In practice something like
  \[
  y = \beta_0 + \beta_1 \times \text{height} + \beta_2 \times 1(\text{type}=2) + \beta_3 \times 1(\text{type}=3) + \epsilon
  \]
- Wilkinson-Rogers formulas:
  \[
  y = \text{height} + \text{type}
  \]
  (Interpretation depends on whether variables are categorical or continuous)

Model Formulas in R

- \( R \) representation \( y \sim \text{height} + \text{type} \) where \( \text{type} \) is a factor
- Interactions \( a:b, a\times b = a + b + a:b \)
- Algebra \( (a:(b + c)) = a:b + a:c \) etc.
- Notice special interpretation of operators
- Special items: \texttt{offset}, \texttt{-1} (no intercept)

Fitting Linear Models

```r
data(airquality)
aq <- transform(airquality, Month=factor(Month))
fit.aq <- lm(log(Ozone) ~ Solar.R + Wind + Temp + Month, data=aq)
```

- \texttt{lm} generates a fitted model object
- Extract information from model object
- Fit other models based on model object
Inspecting Model Objects

- Extract information about the fit
  - `summary(fit.aq)`
  - `fitted(fit.aq), resid(fit.aq)`
  - `anova(fit.aq, fit.aq2)`
  - `plot(fit.aq) – diagnostics`
  - `predict(fit.aq, newdata)`

Model Search

- `anova(model) “Type I” sum of squares`
- `drop1 (“Type III”), add1`
- `step (AIC/BIC) criteria`
- `update modifies a previous model`

Demo 3

data(airquality)
aq <- transform(airquality, Month=factor(Month))
fit.aq <- lm(log(Ozone) ~ Solar.R + Wind + Temp + Month, data=aq)
fit.aq2 <- update(fit.aq, ~ . - Month)
summary(fit.aq)
plot(fit.aq)
drop1(fit.aq, test="F")
anova(fit.aq, fit.aq2)

Contrasts

- Categorical variables generally need a representation using one less coefficient than the number of categories
- Notice that the coefficients for Month are differences to the month of May.
- The R default is to set the first level of a factor to zero (treatment contrasts) for an unordered contrast
- However, other parametrizations are available: Sum contrasts, Helmert contrasts, successive differences etc.
- These can be set as attributes to individual factors, e.g. `contrasts(fff) <- "contr.sum"`. The right hand side can be a name of a contrast generating function, or a matrix $(n \times (n – 1))$
- You can also set a specific contrast in a model formula, using terms like `C(Month,"contr.sum")`
Contrast Matrices

▶ Contrast matrices are typically created by calls like

> contr.treatment(3)

2 3
1 0 0
2 1 0
3 0 1

▶ (some contrast types allow further parameters so that you can limit the degree of polynomial contrasts)

▶ Each row of the contrast matrix is inserted in the design matrix for observations in that group.

▶ Notice that the matrices map from contrasts to fitted values.

Generalized Linear Models

▶ Statistical distribution (exponential) family

▶ Link function transforming mean to linear scale

▶ Deviance

▶ Examples; Binomial, Poisson, Gaussian (σ known — in principle)

▶ Canonical link functions: logit, log, identity

▶ Fit using glm in R

Demo 4

```r
no.yes <- c("No","Yes")
smoking <- gl(2, 1, 8, no.yes)
obesity <- gl(2, 2, 8, no.yes)
snoring <- gl(2, 4, 8, no.yes)
n.tot <- c(60,17,8,2,187,85,51,23)
n.hyp <- c(5,2,1,0,35,13,15,8)
data.frame(smoking,obesity,snoring,n.tot,n.hyp)
hyp.tbl <- cbind(n.hyp,n.tot-n.hyp)
glm.hyp <- glm(hyp.tbl~smoking+obesity+snoring,
family=binomial("logit"))
summary(glm.hyp)
library(MASS)
confint(glm.hyp)
confint.default(glm.hyp)
```

Likelihood-Based Inference for GLMs

▶ Wald approximations (\(\hat{\beta}/\text{s.e.}(\hat{\beta})\), etc.) can be badly inaccurate in small samples

▶ Likelihood-based inference is preferable

▶ Use `drop1(model, test="Chisq")` (for binomial/Poisson)

▶ `profile` (in MASS for glm) investigates behaviour of likelihood around maximum

▶ `plot(profile(model))` shows signed LR statistic \(\text{sign}(\hat{\beta} - \beta)\sqrt{Q}\) when varying each parameter.

▶ `confint` gives likelihood-based confidence intervals
I shall only scratch the surface here!

A central concept is `Surv` objects, encapsulating right-censored survival time, and possibly also left-truncation (delayed entry), and more.

- `survfit` for Kaplan-Meier curves
- `survdiff` for logrank test
- `coxph` for fitting the proportional hazards model, using an interface very similar to that of `glm`

```r
data(melanom)
attach(melanom)
surv.all <- survfit(Surv(days,status==1))
plot(surv.all)
survdiff(Surv(days,status==1)~sex)
summary(coxph(Surv(days,status==1)~sex))
```

Nonlinear Regression – `nls`

- Fits nonlinear models using least squares
- Self-starting models allows automatic determination of start values for iteration
- Writing self-starting models

```r
nlsout <- nls(y ~ A*exp(-alpha*t),
               start=list(A=2,alpha=0.05))
```

Right side of model formula is arithmetic expression (no special interpretation for factors, etc.)

Notice that this is a `vectorized` expression (very useful for ODE solvers)

The `start` argument defines which are parameters to be estimated and starting values for the iterative algorithm
Summary Output

> summary(nlsout)

Formula: y ~ A * exp(-alpha * t)

Parameters:

|        | Estimate | Std. Error | t value | Pr(>|t|) |
|--------|----------|------------|---------|----------|
| A      | 4.75402  | 0.29408    | 16.166  | 5.88e-08 *** |
| alpha  | 0.18364  | 0.02023    | 9.079   | 7.95e-06 *** |

Residual standard error: 0.3892 on 9 degrees of freedom

Correlation of Parameter Estimates:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.6724</td>
<td></td>
</tr>
<tr>
<td>alpha</td>
<td></td>
<td>1.0000</td>
</tr>
</tbody>
</table>

Profiling

par(mfrow=c(2,1))
plot(profile(nlsout))

- Very much the same idea as for glm
- Calculate profile t statistics, i.e. signed values of
  \( \sqrt{\Delta \text{SSD}/\text{SE}(\hat{\theta})} \) for varying values of \( \theta \), maximized over other parameters and signed according to which side of \( \hat{\theta} \) you're on.
- Plots of \(|t|\) with indication of approximate confidence levels (.99, .95, .90, .80, .50)

Profile Plots

Confidence Intervals

> confint(nlsout)

Waiting for profiling to be done...

<table>
<thead>
<tr>
<th></th>
<th>2.5%</th>
<th>97.5%</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4.0931837</td>
<td>5.4381261</td>
</tr>
<tr>
<td>alpha</td>
<td>0.1395008</td>
<td>0.2342045</td>
</tr>
</tbody>
</table>

- The same procedure as in profile plots, but showing results numerically
Supplying Gradient Information

- Often not worth it... If not supplied, numeric differentiation will be done.
- Right hand side must evaluate to something that has a gradient attribute, e.g.
  \[ \text{nls}(y=\text{structure}(A*\exp(-\alpha*t),
    \text{gradient}=\text{cbind}(A=\exp(-\alpha*t),
    \alpha=-A*t*\exp(-\alpha*t))),
  \text{start}=\text{list}(A=10,\alpha=3.5)) \]
- The \text{deriv()} function can do much of the hard work for you if the model is given as a simple arithmetic expression.

Selfstarting Models

- How to get starting values?
- Mosty an art, but can be worked out for typical situations
- Typical tricks:
  - transform to linearity
  - calculate “landmarks” as function of parameters (AUC, initial slope, position and value of maximum) estimate them empirically and solve for parameters
- Idea: Store algorithm for starting value within model object
- Standard models supplied: \text{SSfol}, etc.

Writing Your Own Selfstart Models

- Main thing to supply is the initializer. Must be
  \begin{verbatim}
  \text{function}(\text{mCall, LHS, data})
  \end{verbatim}
- \text{mCall} is the matched call. Used to access the names passed for the parameters.
- \text{LHS} is the left hand side (response). NB: Unevaluated.
- \text{data} is the modelling frame.
- Return value should be a named list.
- \text{selfStart()} constructor function does the rest. If used on a model formula, \text{deriv} is used for gradient info.

Demo 6

- Selfstart example:
  \begin{verbatim}
  \text{initexp <- function}(\text{mCall, data, LHS}) {
    y <- \text{eval(LHS, data)}
    cc <- \text{coef(lm(\text{log}(y)~t, data=\text{data}))}
    l <- \text{list(\text{exp(cc[1]),-cc[2])}}
    \text{names(l) <- mCall[c("A","a")]}  
    \text{l}
  }
  \end{verbatim}

  \begin{verbatim}
  \text{SSexp<-\text{selfStart}(~A*\exp(-a*t),}
  \text{initial = initexp},
  \text{parameters = c("A","a")})

  \text{summary(nls(y~SSexp(t,A,a)))}
  \text{summary(nls(y~SSexp(t,B,b)))}  \text{# mCall stuff needed here}
  \end{verbatim}
Overview

- What does an R function look like
- Flow control
- Matrix algebra
- Optimizers
- Largish example: Time splitting
- (The whole thing will be rather superficial)

Simple Functions

- \texttt{logit} <- \texttt{function(p) log(p/(1-p))}
- \texttt{logit(0.5)}
- Formal arguments
- Actual arguments
- Positional matching: \texttt{plot(x,y)}
- Keyword matching: \texttt{t.test(x ~ g, mu=2, alternative="less")}
- Partial matching: \texttt{t.test(x ~ g, mu=2, alt="1")}

Flow control

- if/else
- ifelse()
- switch()
- for loops
- repeat, while
- break
Apply-functions/loop avoidance

- `lapply` – list-apply
- `sapply` – simplifying apply
- `tapply` – tabulating apply
- `apply, sweep` – along slices of tables
- `replicate` – repeat expression

Matrix algebra

- R contains a pretty full set of primitives for matrix calculus
- `A %*% B` for matrix multiplication
- `solve(A, b)` for solving linear equations. (`solve(A)` for matrix inverse)
- Various special products and decompositions

Example of Matrix Code

Code to calculate the Greenhouse-Geisser epsilon:

```r
Psi <- T %*% Sigma %*% t(T)
B <- T %*% object$SSD %*% t(T)
pp <- nrow(T)
U <- solve(Psi, B)
lambda <- Re(eigen(U)$values)
GG.eps <- sum(lambda)^2/sum(lambda^2)/pp
```

Optimization Tools

- 1-dimensional: `optimize`
- `nlm`, Newton-style optimizer
- `optim`, wrapper for several advanced optimizers
Demo 1

```r
mll <- function(theta) -dbinom(4, 10, theta, log=TRUE)
plot(mll)
optimize(mll, c(0,1))
nlm(mll, .5)
optim(.5, mll, method="BFGS")
```

---

Example: Time Splitting

- Split survival data into bands according to some time scale
- Used in survival analysis and epidemiology
- Vector of (delayed-entry) survival times
- Vector of break points
- (Possibly) individual origin of time scale for splitting

---

“The SAS Way”

(pseudocode)

```r
loop over individuals (implicit)
  loop over intervals
    if overlap with survival time
      trim survival time to interval
      output modified case
```

The R Way

Might mimic the SAS strategy, but it is inefficient in R. Here’s another idea:

```r
loop over intervals
  select subjects that overlap with interval
  trim times to interval
  keep resulting vector
stick all vectors together
```

That way, we can utilize vectorization of the selection and trimming tasks.
Trimming to a Single Interval

- **Survival:** time1, time2, event
- **Interval:** left, right
- Turns out to be easier to "shoot first and ask questions later" about the overlap:

```
en <- pmax(time1, left)
ex <- pmin(time2, right)
ev <- event & (time2 <= right)
valid <- (en < ex)
Surv(en[valid], ex[valid], ev[valid])
```

Notice that left and right can be vectors and incorporate a subject-specific origin.

---

Trimming as Function

```
trimToI <- function(I)
{
  left <- I[1] + origin
  right <- I[2] + origin
  en <- pmax(time1, left)
  ex <- pmin(time2, right)
  ev <- event & (time2 <= right)
  valid <- (en < ex)
  data.frame(S=Surv(en[valid], ex[valid], ev[valid]), subj=subj[valid])
}
```

This is designed to be defined inside the actual timesplit function. Notice that some variables are left to be bound via *lexical scope*.

---

Processing All Time Bands

We want to use some sort of apply-function and collect results as list of data frames. Here's a nice way:

```
nbrk <- length(brks)
Imat <- cbind(brks[-nbrk], brks[-1])
l <- apply(Imat, 1, trimToI)
```

and then just stick things together with

```
result <- do.call("rbind", l)
```

as it turns out, you need to do a little bit more because the `rbind` turns the `Surv` objects into matrices.

---

Describing the Timebands

The output from our timesplitting function should contain a variable describing to which time band each piece belongs (this is not obvious if the origin differs between individuals).

Getting the result as a factor with the proper level set is a little tricky. Here's one way:

```
lbl <- apply(Imat, 1, function(I) paste("(" , I[1] , ",", I[2] , ")", sep=""))
f <- factor(lbl, levels=lbl) # avoid level sorting
rep(f, lapply(l, nrow))
```

Next slide gives the final function.
timesplit <- function(S, brks, origin=0, subj=1:nrow(S))
{
  time1 <- S[,1] ; time2 <- S[,2] ; event <- S[,3]

  trimToI <- function(I)
  {
    en <- pmax(time1, left)
    ex <- pmin(time2, right)
    ev <- event & (time2 <= right)
    valid <- (en < ex)
    data.frame(S=Surv(en[valid], ex[valid], ev[valid]),
               subj=subj[valid])
  }

  nbrk <- length(brks)
  Imat <- cbind(brks[-nbrk], brks[-1])
  l <- apply(Imat, 1, trimToI)
  result <- do.call("rbind", l)
  attr(result$S, "type") <- "counting"
  class(result$S) <- "Surv"

  lbl <- apply(Imat, 1,
               function(I) paste("(",I[1], ",", I[2], "]", sep=""))
  f <- factor(lbl, levels=lbl) # avoid level sorting issues
  cbind(result, band=rep(f, lapply(l, nrow)))
}
Exercises

1. The file nickel.dat will be made available to you somehow. It contains
the following variables

id Subject ID
icd Cause of death (0: not dead, 160: nasal cancer, 162,163: lung cancer)
expos Index of exposure to arsenic
date.bth Date of birth
date.1st Date of 1st exposure
date.in Date of entry into study
date.out Date of exit from study (death of censoring)

(a) Read the data as a data frame in R, using
nickel <- read.table(....) (you get to fill in the dots!).
Look at the first few lines of the data frame and explain what you see. Also use summary(nickel).

(b) Try hist(nickel$expos). Notice the skewness, and the peak at zero. Use cut(....) to create a factor containing a grouping of expos into five groups: 0, 0.5–4.0, 4.5–8.0, 8.5–12.0, 12.5+. Assign prettier level names to the result if you want:
level(myfactor) <- ....

(c) The four date variables were read as factors. Convert them using as.Date.

(d) Make a summary and a histogram of age at first exposure. You need to take the difference (a difftime object), then convert using as.numeric.

(e) Create a binary indicator for death from lung cancer vs. censoring or death from other causes.

(f) Use save(....) to save your modified data frame to disk

2. Continuing with the nickel data,

(a) Tabulate the number of lung cancer deaths by exposure level (in five groups as defined earlier)

(b) Tabulate the total time at risk in the same five groups, and the crude death rate: The number of deaths per person-year

(c) (Explain why this is at best a semi-sensible thing to do!)
3. Look at the data for the Ashina cross-over trial (use `library(ISwR);data(ashina)` to make them available.

(a) Perform a simple paired t-test comparing placebo and active groups. However, notice that this ignores a potential period effect.

(b) One way to incorporate a period effect is to do an *unpaired* t-test comparing the period differences between the two groups. Try this (notice that you get period differences from treatment differences by switching the sign in one group).

(c) Another way is to generate data in *long format*:

```r
attach(ashina)
vas <- c(vas.active, vas.plac)
treat <- rep(1:2, each=16)
id <- rep(1:16,2)
period <- rep(c(1,2,2,1), c(10,6,10,6))
detach()
```

and then fit a linear model to `vas` with additive effect of `treat`, `period`, and `id`. (Notice that they need to enter as factors). Do this, and compare with the t-test approach.

(d) (advanced, you may want to skip this) The `reshape` function can be used to generate the long format. Try this.

4. Consider the `graft.vs.host` data from ISwR.

(a) Fit a logistic regression model set, predicting the `gvhd` response. Use different transformations of the `index` variable. Reduce the model using backward elimination.

(b) Use the `confint` function from the MASS package to find improved confidence intervals for the regression coefficients.
5. The plot below is from a randomized study of the effect of Tamoxifen treatment on bone mineral metabolism, in a group of patients that were treated for breast cancer.

It was originally created by S-PLUS in 1993. The data are available in the file alkfos.csv (using comma as separator, so read.csv will read it).

(a) Calculate the number of patients available by group and time. This can be done using `aggregate(!is.na(alkfos), list(alkfos$grp), sum)`

(b) Calculate the statistics for the inner part of the plot. This is most easily done by using `aggregate` as above on `sweep(alkfos[-1], 1, alkfos$c0, "/")` (figure out what this does first!). To get the percentage change subtract 1 and multiply by 100.

(c) Try generating an R plot as similar to the original as possible. (You cannot reproduce it perfectly because the fonts and plot symbols differ.) Hints: `segments` for the error bars, notice the slight offset of the lower curve, make sure to increase the margins to make room for the numbers at the bottom. To get numeric vectors from the output of `aggregate`, either convert the data frame to a matrix with `as.matrix` or use `unlist` on the rows.

6. Generate a Trellis plot or `sqrt(igf1)` by age from the juul data in ISwR, grouped by the sex and tanner variables. Add a regression line to each plot.
7. Sometimes, you generate a regression analysis and want to compare regression coefficients for a factor in an all-versus-all fashion. Consider the Month variable in a regression analysis of the airquality data. For simplicity, set

\[
\text{fit.aq} \leftarrow \text{lm(Ozone ~ Wind+factor\{Month\}, data=airquality)}
\]

(a) The covariance matrix of the fit can be obtained by
\[
V \leftarrow \text{vcov(fit.aq)}
\]
View it.

(b) Notice that the \(4 \times 4\) block at the lower right of the matrix is what refers to the Month factor. Hence define
\[
\text{where} \leftarrow 3:6
\] and
\[
M \leftarrow V[\text{where}, \text{where}]
\]

(c) Now, the month of May is missing since it is set to zero. It is convenient to expand the matrix with a corresponding row and column of zeros:
\[
M \leftarrow \text{cbind(0, \text{rbind(0,M) })}
\]

(d) Now, we can get the corresponding levels
\[
m \leftarrow \text{c(0,coef(fit.aq)[where])}
\]
A nice trick to get all the differences is
\[
D \leftarrow \text{outer(m, m, "-")}
\]

(e) Convince yourself that the corresponding variances can be generated as
\[
\text{DI} \leftarrow \text{diag(M)} \text{ and } \text{outer(DI, DI, "+") - 2*M}
\]

(f) Use the above considerations to write a small function that calculates all the pairwise t-statistics, given a model and the where specification.

(g) (advanced) The above relies heavily on the fact that “treatment contrasts” were used. How can you modify the code to accommodate other kinds of contrasts (contr.helmert, contr.sum, etc.)?

(h) (horribly advanced) The information about where and the type of contrasts used is actually available inside the fitted model object. Try figuring out where it is and how to make use of it.