Statistikværktøjet R

Peter Dalgaard

p.dalgaard@biostat.ku.dk

Biostatistisk Afdeling, Kbh. Univ.

Indledende spørgsmål

- Hvem er jeg?
- Hvad er **R**
- Hvem er I?
- Hvad gør jeg her?

Hvem er jeg

- Lektor, sundhedsvid.
- Univ. uddannet statistiker
  - Underviser
  - Arbejder med forskningsdata
  - Mange i medicinalindustrien
  - Men også blandt ingeniører, i bankerne, i fødevaresektoren, osv.

Frit software i forskning

- Videnskabeligt software er en måde at kommunikere ideer på
- Algoritmer stilles til rådighed for andre forskere med henblik på studium, kritik og forbedring
- Umiddelbare gevinster ved frit software:
  - Værktøjer af høj kvalitet
  - . . . som kan spille sammen
  - Portabilitet
  - Hurtig tilgængelighed
  - Overholdelse af standarder

Hvad er R?

- Statistikværktøj (“environment”)
- Analyser
- Grafik
- Centreret omkring et funktionsprogrammeringssprog der på overfladen ligner C, men faktisk er tættere på Lisp og APL
- GPL 1996, Core Team 1997, v.1.0.0 29. feb 2000
- I dag 235000 linjer C, 85000 linjer Fortran, 65000 linjer R + udvidelsespakker

Hvem er I?

Gæt:

- Ikke meget forstand på teoretisk statistik eller matematik
- En temmelig grundig Unix/Linux baggrund
- Ikke så mange fra forskningsmiljøer, snarere erhvervsvirksomheder
- Nysgerrighed
- Måske behov for datapræsentation og enkle analyser

Hvad gør jeg her?

- Korte tutorials i **R** sproget og grafiske faciliteter (hugget fra kursusmateriale af Doug Bates & Thomas Lumley)
- **R** projektet
- **R** pakker
- **R**-Tcl/Tk interface

An Overview of R

- What is **S**?
- What is **R**?
- Installation of **R**
- Demonstration

The S language

- It is a language and system developed by John Chambers and his co-workers at Bell Laboratories (formerly part of AT&T, now part of Lucent Technologies).
- The Association for Computing Machinery presented its 1998 Software System Award to John stating, “S has forever altered the way people analyze, visualize, and manipulate data . . . ”
- It is the *de facto* standard for computing in the Statistics research community.
- It is documented in many books, the best known of which are by Venables and Ripley (*Modern Applied Statistics with S-PLUS* and *S Programming*) and by Chambers (*Programming with Data*).

Design Goals for S

- To provide an environment for interactive computing with data.
- To allow users to transition easily into programmers as the need arises.
- To provide both exploratory and presentation graphics.
- To avoid reinventing existing tools.

Characteristics of S

- It is an interactive language based on functions and function calls.
- It provides an environment with persistent, self-describing objects. In particular, functions are first-class objects.
- It has a wide variety of graphics device drivers and controls.
- It provides interfaces to existing code and systems.

Implementations of S

- **S-PLUS**, sold by Insightful Corp. (formerly MathSoft, Inc.) and based on their exclusive license of the original **S** source from Lucent Technologies.
- **R**, an Open Source project conforming for the most part to the published descriptions of **S**.
  - Initially developed by Ross Ihaka and Robert Gentleman at the University of Auckland
  - Now developed and maintained by an widely-dispersed, international group of volunteers from academia and industry.
  - Operates through web sites (www.r-project.org), archives (cran.r-project.org) e-mail lists, CVS sites, rsync sites.

What is R?

- An Open Source implementation of John Chambers' award-winning **S** language
- A language and environment for data analysis and graphics
- A means of technology transfer through packages
- A flexible data exchange mechanism accessing:
  - text files and saved **R** workspaces
  - **S-PLUS** data objects, **SAS** XPORT datasets, **SPSS** saved datasets, Minitab worksheets, . . .
  - relational databases – ODBC, PostgreSQL, MySQL
- An embeddable extension language
- Part of the GNU (GNU's Not Unix) software system.
- Just install it and try it. Comes with a money-back guarantee.

How do I get R?

- The informational web site `http://www.r-project.org/`
- **CRAN** - the Comprehensive R Archive Network
  - The primary site is `http://cran.r-project.org/`
  - Mirror sites are available for many countries, e.g. `http://cran.dk.r-project.org/`.
- **CRAN** sites have binary distributions for Windows 95, 98, ME, NT4 and 2000 on Intel, for the Macintosh (System 8.6 to 9.1 and MacOS X), and for several Linux distributions.
- New releases occur frequently - about every 3 months.

Installing R

Windows Download and run the installer, *SetupR.exe*.

Macintosh The distribution consists of one binhexed (.hxq) file that you expand using standard tools.

Linux RPM files are available for RedHat, SuSE, and Mandrake. Deb files are available for Debian. Under Debian you can list a **CRAN** archive in */etc/apt/sources.list* for automatic updates.

Unix Download and expand the compressed tar file of the sources. Run *./configure* then *make; make check; make install*

A sample session

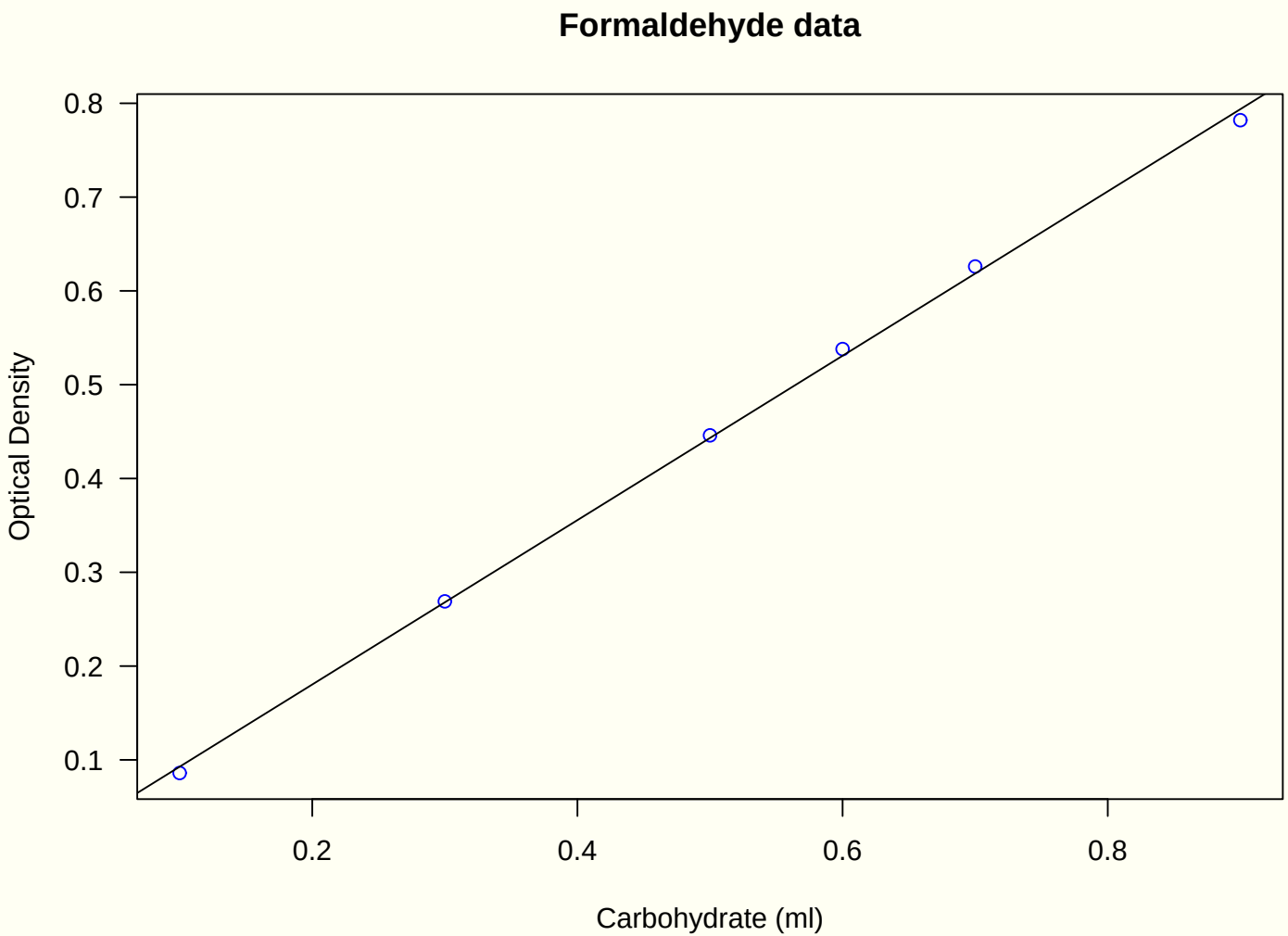
```
> demo(graphics)   # demonstrates some of the graphics capabilities
> data()           # list the available data sets
Data sets in package ‘base’:

Formaldehyde    Determination of Formaldehyde concentration
HairEyeColor    Hair and eye color of statistics students
InsectSprays    Effectiveness of insect sprays
LifeCycleSavings
                Intercountry life-cycle savings data
OrchardSprays   Potency of orchard sprays
PlantGrowth     Results from an experiment on plant growth
Titanic         Survival of passengers on the Titanic
ToothGrowth     The effect of vitamin C on tooth growth in guinea pigs
UCBAdmissions   Student admissions at UC Berkeley
USArrests       Violent crime statistics for the USA
...
```

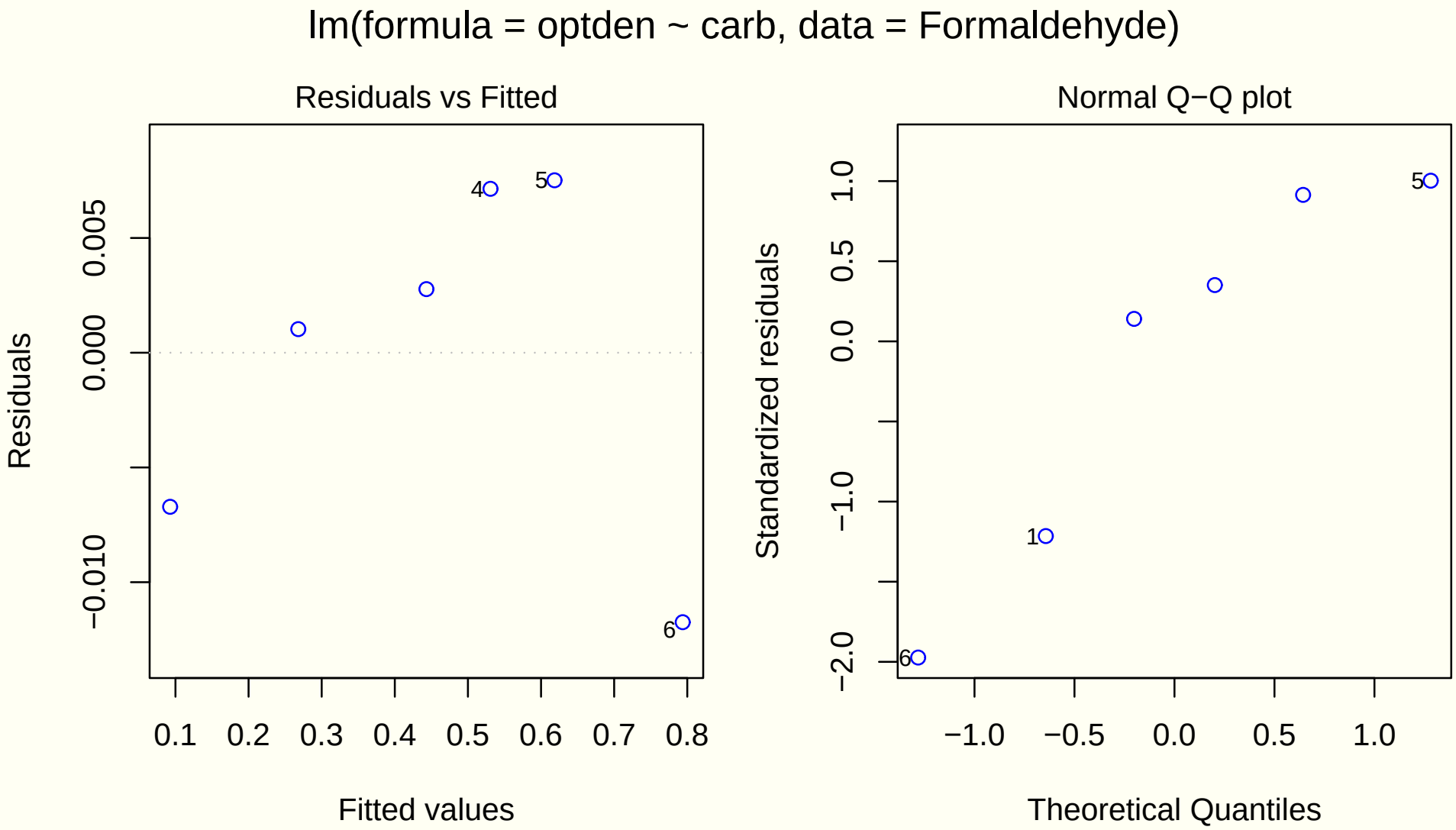
Functions and data sets are documented

```
> example(Formaldehyde)    # run the example for this data set
Frmldh> data(Formaldehyde)
Frmldh> plot(optden ~ carb, data = Formaldehyde, xlab = "Carbohydrate
    ylab = "Optical Density", main = "Formaldehyde data", col = 4,
    las = 1)
Frmldh> abline(fm1 <- lm(optden ~ carb, data = Formaldehyde))
Frmldh> summary(fm1)
Call: lm(formula = optden ~ carb, data = Formaldehyde)
Residuals: 1         2         3         4         5         6
   -0.006714  0.001029  0.002771  0.007143  0.007514 -0.011743
Coefficients: Estimate Std. Error t value Pr(>|t|)
(Intercept)   0.005086   0.007834   0.649    0.552
carb          0.876286   0.013535  64.744 3.41e-07 ***
Signif. codes:  0  '***'  0.001  '**'  0.01  '*'  0.05  '.'  0.1  ' '
Residual standard error: 0.008649 on 4 degrees of freedom
Multiple R-Squared: 0.999,      Adjusted R-squared: 0.9988
F-statistic:  4192 on 1 and 4 degrees of freedom,       p-value: 3.409
Frmldh> plot(fm1)            # diagnostic plots
```

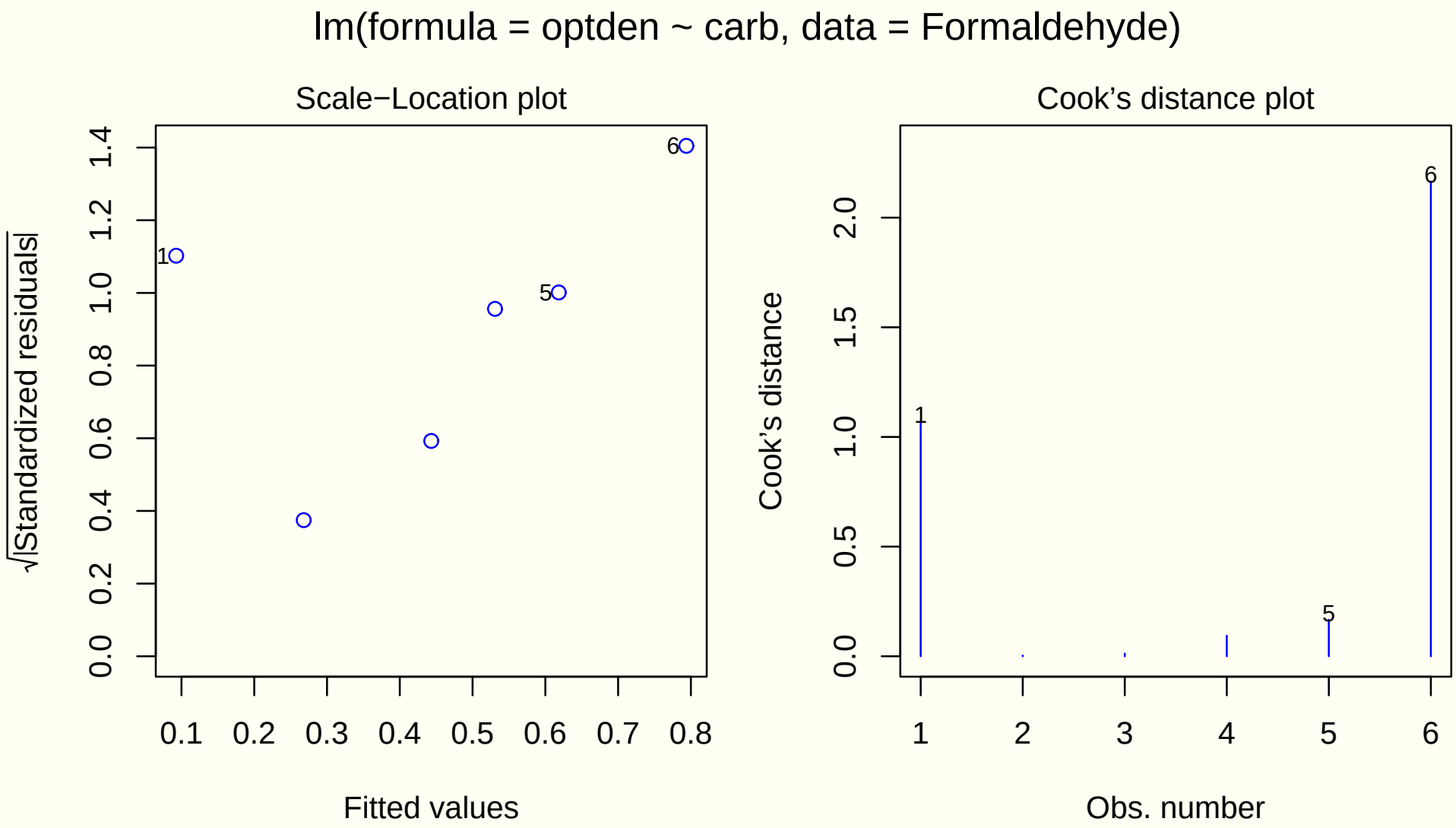
Formaldehyde - plot 1



Formaldehyde - plot 2



Formaldehyde - plot 3



9 basic R functions you should know

Typing a function name and <RET> simply displays the function. To invoke the function you must include an argument list in (), even if the list is empty. That is, use q()<RET>, not just q<RET>.

q Quit the session

help Get help on a function or object

help.start Allow the use of a web browser for reading help

example Run the example from the help page for an object

data List the available data sets or import a data set

library List available packages or attach a package

objects List the objects in the workspace

summary Summarize an object

str Show the low-level structure of an object

Sources of information about R

- The web site `http://www.r-project.org/` and **CRAN**
- The frequently asked questions (FAQ) list at `http://www.ci.tuwien.ac.at/~hornik/R/`, which is mirrored at `http://cran.r-project.org/doc/FAQ/`. Most of the contents of this section (and many other topics) are covered in the FAQ.
- The manuals in the documentation directory `http://cran.r-project.org/doc/manuals`. See especially *R-intro.pdf* and *R-data.pdf*.

Types of data in R

- The basic data object is a vector of elements of type:

  numeric numbers - either floating point or integer

  character each element is a character string

  logical each element is *TRUE* or *FALSE*

  list elements can be any type of object, including other lists

- Components of the **S** language, such as functions, are also vectors.
- Any vector can include the missing data marker *NA* as an element.
- All vectors have a length and a mode. The functions *length* and *mode* return this information as does the *str* function.
- A structure consists of a data object plus additional information. Matrices (or arrays, in general) and time series are examples of structures.

Generating simple vectors

- The assignment operator in **R** is the two-character sequence '<-'. (An alternative is available but its use is **strongly** discouraged.)
- Any type of vector can be created explicitly with the *c* (concatenation) function.
- Numeric vectors can be generated with the *seq* function or the sequence operator ':'
- Pseudo-random samples from various distributions can be created. The function names have the pattern '*r*<distname>', such as *runif* for a uniform distribution or *rnorm* for a normal distribution.

Numeric vectors

```
> vv <- c(1, 3.14159, 17)   # assign a vector to the name vv
> vv                        # display the object named vv
[1]  1.00000  3.14159 17.00000
> length(vv)                # display the length of vv
[1] 3
> mode(vv)                  # display the mode of vv
[1] "numeric"
> str(vv)                   # a more concise description
 num [1:3]  1.00  3.14 17.00
> v1 <- 1:5                 # generate a sequence (of integers)
> str(v1)
 int [1:5] 1 2 3 4 5
> mode(v1)                  # for integers the mode is "numeric"
[1] "numeric"
> pi                    # ratio of the circumference to the diameter
[1] 3.141593
> v2 <- seq(-pi, pi, len = 11)# generate a sequence from -pi to pi
> str(v2)
 num [1:11] -3.142 -2.513 -1.885 -1.257 -0.628 ...
```

Generating pseudo-random data

```
> rn <- rnorm(100)           # 100 samples from a normal(0, 1) dist'n
> str(rn)
 num [1:100] -0.258  0.476 -0.110  1.410  0.635 ...
> stem(rn)                   # stem-and-leaf plot
  The decimal point is at the |
  -2 | 54310
  -1 | 887554444321111
  -0 | 9999877766655544443333222211111110
   0 | 000112223345555556668899 9
   1 | 00113466899
   2 | 1112346
> rp <- rpois(500, lambda = 4.3) # a sample from a Poisson dist'n
> table(rp)                  # the frequency table
rp
 0  1  2  3  4  5  6  7  8  9 10 11 13
 8 24 64 91 90 77 72 33 23 10  4  2  2
> mean(rp)                   # the mean (average) is as expected
[1] 4.39
```

Character and logical vectors

```
> LETTERS                    # the upper case letters
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O"
[16] "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
> mode(LETTERS)
[1] "character"
> str(LETTERS)
 chr [1:26] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" ...
> kids <- c("Barbara", "Michael") # elements are entire strings
> str(kids)
 chr [1:2] "Barbara" "Michael"
> kids < "C"                 # string comparison is lexicographic
[1]  TRUE FALSE
> v3 <- LETTERS[1:5]         # a subset of a vector
> v3
[1] "A" "B" "C" "D" "E"
> v3 == "A"                  # equality comparison uses "==", as in C
[1]  TRUE FALSE FALSE FALSE FALSE
> mode(v3 == "A")
[1] "logical"
```

Named elements, patterned data

```
> con <- c(e = exp(1), pi = pi, twopi = 2*pi)
> con                              # well-known constants
       e       pi    twopi
2.718282 3.141593 6.283185
> str(con)
 Named num [1:3] 2.72 3.14 6.28
 - attr(*, "names")= chr [1:3] "e" "pi" "twopi"
> names(con)                       # query the names
[1] "e"     "pi"    "twopi"
> names(con) <- c("e", "pi", "2pi") # assign new names
> con
       e       pi      2pi
2.718282 3.141593 6.283185
> rep(1:2, 5)                      # repeat a vector
 [1] 1 2 1 2 1 2 1 2 1 2
> rep(1:2, c(5, 5))                # repeat elements of a vector
 [1] 1 1 1 1 1 2 2 2 2 2
```

Factors

Qualitative data that can assume only a discrete set of values are represented by a *factor*.

```
> trt <- factor(rep(c("Control", "Treated"), c(3, 4)))
> str(trt)
 Factor w/ 2 levels "Control","Treated": 1 1 1 2 2 2 2
> summary(trt)                    # summary gives a frequency table
Control Treated
      3       4
```

If the levels of a factor are numeric (e.g. the treatments are labelled “1”, “2”, and “3”) it is important to ensure that the data are actually stored as a factor and not as numeric data. Always check this by using *summary*.

Data frames

A *data.frame* is the basic **S** structure for a “data set” that can be represented as a set of observations (rows) on several variables (columns). Most of the data sets you see listed in the output of *data()* are data frames. They are similar to a **SAS** or an **SPSS** data set.

```
> data(Formaldehyde)
> str(Formaldehyde)
'data.frame':   6 obs. of  2 variables:
 $ carb  : num  0.1 0.3 0.5 0.6 0.7 0.9
 $ optden: num  0.086 0.269 0.446 0.538 0.626 0.782
> summary(Formaldehyde)
      carb           optden
 Min.   :0.1000   Min.   :0.0860
 1st Qu.:0.3500   1st Qu.:0.3132
 Median :0.5500   Median :0.4920
 Mean   :0.5167   Mean   :0.4578
 3rd Qu.:0.6750   3rd Qu.:0.6040
 Max.   :0.9000   Max.   :0.7820
```

Data frames (cont'd)

Columns in data frames are usually numeric variables or factors. Always check a data frame using *summary* to ensure that variables that should be factors are factors. Factors are summarized by frequency tables.

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

Using read.table

The *read.table* function reads a table of data from a text file and returns a data frame. Column headers, if available, are used to name the variables. If you have a choice of file format (say when writing a data file from a spreadsheet) use a tab-separated format.

```
> write.table(iris, file = "iris.txt", sep = "\t",
+       quote = FALSE, row.names = FALSE)  # create the sample data
> file.show("iris.txt")                    # view the file
Sepal.Length    Sepal.Width     Petal.Length    Petal.Width     Specie
5.1     3.5     1.4     0.2     setosa
...
5.9     3.0     5.1     1.8     virginica
> iris.new <- read.table(file = "iris.txt", header = TRUE, sep = "\t")
> summary(iris.new)
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
...
       Species
 setosa    :50
 versicolor:50
 virginica :50
```

Extracting subsets

One of the keys to mastering the **S** language is learning to use the extraction (or subset) operators effectively.

A typical message on the R-help email list asks:

> Is there a better solution to select rows from a data.frame than by iterating through the whole set and evaluating every case one by one?
>
> Is there maybe something like:

```
d<-data.frame(...)
maleOver40<-select.data.frame(d,"( sex=m or sex=M) and age > 40)")
```

> Of course, I could use direct database-requests, but this would require my data to be stored in a database...

The general subset operator

- The "*[*" operator is the general extraction operator. It creates an object of the same mode as the object to which it is applied.

  In the expression **`x[i]`** several forms of indices **`i`** can be used:

  positive integers - indicate the positions of the elements to extract. The first position is numbered 1.

  negative integers - indicate all elements except those at indices numbered **`-i`**. That is, **`x[-1]`** means “drop the first element of **`x`**”.

  logical vectors - if **`i`** is a logical vector of the same length as **`x`** then the elements of **`x`** corresponding to **`TRUE`** in **`i`** are returned.

  character variables - are matched against the names of elements of **`x`**.

Examples of the general subset

```
> str(rn)                              # our pseudo-random sample
 num [1:100] -0.258  0.476 -0.110  1.410  0.635 ...
> rn[1:5]                              # the first 5 elements of rn
[1] -0.2584713  0.4760666 -0.1101059  1.4095607  0.6353282
> rn[96:100]                           # the last 5 elements
[1]  0.5211764  1.0264300 -0.9405918 -0.1354008  2.3525296
> rn[-(1:95)]                          # another way of extracting those
[1]  0.5211764  1.0264300 -0.9405918 -0.1354008  2.3525296
> ## The parentheses in that last expression are important.
> ## Try it without them.
> str(rn[rn > 0])                      # only the positive values
 num [1:44] 0.476 1.410 0.635 0.813 2.323 ...
> names(con)                           # our vector of constants
[1] "e"   "pi"  "2pi"
> con[c("e", "pi")]                    # extracting by name
       e       pi
2.718282 3.141593
```

Extracting single elements

The **"["** operator returns an object of the same mode as the object to which it is applied. The **"[["** and **"$"** operators extract single elements in their native mode. The distinction is like that between "an element of a set" (what **"[["** produces) and "a subset of size 1 from a set", (what **"["** produces).

```
> li <- list(pi = pi, e = exp(1)) # a list with two numeric elements
> mode(li)
[1] "list"
> mode(li[1])                     # still a list
[1] "list"
> mode(li[[1]])                   # this is numeric
[1] "numeric"
> li[1]                           # this prints as a list
$pi
[1] 3.141593
> sqrt(li[1])                     # no square root of a list
Error in sqrt(li[1]) : Non-numeric argument to mathematical function
> sqrt(li[[1]])                   # square roots of numbers are ok
[1] 1.772454
```

Subsets applied to data frames

Data frames are most naturally regarded as a rectangular structure. We can use **"["** to extract subsets of rows or subsets of columns or both. For this, two indexing expressions are used. Omitting an indexing expression for the rows (or columns) means to use all the rows (or columns).

```
> dim(iris)                          # iris data has 150 obs. on 5 variables
[1] 150   5
> summary(iris[, c(1,2,5)])          # only Sepal dimensions and Species
  Sepal.Length    Sepal.Width           Species
 Min.   :4.300   Min.   :2.000   setosa    :50
 1st Qu.:5.100   1st Qu.:2.800   versicolor:50
 Median :5.800   Median :3.000   virginica :50
 Mean   :5.843   Mean   :3.057
 3rd Qu.:6.400   3rd Qu.:3.300
 Max.   :7.900   Max.   :4.400
> dim(iris[iris$Species == "setosa", ]) # the 50 setosa observations
[1] 50  5
```

‘subset’ and the %in% operator

Extracting rows from a data frame based on values of some of the variables is common. Writing expressions like **`iris[iris$Species == "setosa", ]`** can become tedious. The *subset* function is an alternative.

```
> dim(subset(iris, Species == "setosa"))
[1] 50  5
```

Sometimes we want to select for several possible values of a variable. A general function *match* returns the indices generated by matching one set of values against another another. It is often used to find out which values are present in another set. The operator *%in%* phrases this more succinctly.

```
> dim(subset(iris, Species %in% c("setosa", "versicolor")))
[1] 100   5
```

The selection question from R-help

Recall the question asked on the R-help email list

Is there maybe something like:

```
d<-data.frame(...)
maleOver40<-select.data.frame(d, "( sex=m or sex=M) and age > 40)'
```

Peter Dalgaard answered

```
> maleOver40 <- subset(d, sex %in% c('m', 'M') & age > 40)
```

Subsets that are larger than the original

The extraction operator *[* is more general than a subset operator. By repeating indices we produce “subsets” that are larger than the original.

```
> c("Yes", "No")[rep(1:2, 5)]    # same as rep(c("Yes", "No"), 5)
 [1] "Yes" "No"  "Yes" "No"  "Yes" "No"  "Yes" "No"  "Yes" "No"
```

This is used to merge data sets (or to join data tables, in the relational database terminology). For example, the *nlme* package contains data on mathematics achievement scores. The student-level table records the school, the test score, and some demographic information. A companion table records school-level information.

```
> data(package = "nlme")
Data sets in package ‘nlme’:
...
MathAchSchool              School demographic data for MathAchieve
MathAchieve                Mathematics achievement scores
```

More on single element extraction

The **"[["** and **"$"** operators both extract a single element in its native mode. The **"[["** operator requires an index position or a (quoted) name. The **"$"** operator only works on lists and requires a name without quotes. The expression **x$nm** is essentially a short form for **x[["nm"]]**.

```
> data(Formaldehyde); str(Formaldehyde)
'data.frame':   6 obs. of  2 variables:
 $ carb  : num  0.1 0.3 0.5 0.6 0.7 0.9
 $ optden: num  0.086 0.269 0.446 0.538 0.626 0.782
> Formaldehyde$optden
[1] 0.086 0.269 0.446 0.538 0.626 0.782
> Formaldehyde[["optden"]]                # same thing
[1] 0.086 0.269 0.446 0.538 0.626 0.782
> Formaldehyde[[2]]                       # same again
[1] 0.086 0.269 0.446 0.538 0.626 0.782
```

Single element extraction & data frames

Notice that we can use only one index without extra commas to access a column in a data frame.

```
> Formaldehyde[[2]]                          # second column
[1] 0.086 0.269 0.446 0.538 0.626 0.782
> Formaldehyde[ , 2]                         # also the second column
[1] 0.086 0.269 0.446 0.538 0.626 0.782
```

Single element extraction returns a column because a data frame is in fact a list of columns, as indicated by the result of **`str(Formaldehyde)`**. The **`[row,col]`** form of indexing shown previously is a special indexing method that makes a data frame appear to be rectangular.

NA - the missing data marker

The codes **NA** (not available) and **NaN** (not a number) indicates a missing data value in a vector or other data structure. Both are called NA's. An **NA** can be part of the original data, or it can be the result of operations on other data where the result is undefined, or it can be assigned.

```
> rrn <- rnorm(100); lrn <- log(rn)
Warning message:
NaNs produced in: log(x)
 num [1:100]  0.466  0.213 -0.644 -0.497  2.830 ...
 num [1:100] -1.32   NaN   NaN -1.35   NaN ...
> rrn <- rnorm(100); lrn <- log(rrn) # logs of random number
Warning message:
NaNs produced in: log(x)
> str(rrn)                           # both positive and negative
 num [1:100] -0.374 -0.695  0.337 -0.496 -1.851 ...
> str(lrn)                           # log of negative no. not defined
 num [1:100]   NaN   NaN -1.09   NaN   NaN ...
```

Use is.na to check for missing data

Note that we check for missing data with is.na. This is the **only** way to detect missing data. A common mistake is trying to check for missing data with expressions like **`x == NA`**. This doesn't work as expected. Missing values propagate in operations, including comparison operations. Comparing another value to **NA** always produces an **NA**

```
> str(1 + lrn)                          # missing values propagate
 num [1:100]     NaN     NaN -0.0892     NaN     NaN ...
> lrn.msng <- lrn == NA                 # a common mistake
> str(lrn.msng)                         # not the expected result
 logi [1:100] NA NA NA NA NA NA NA NA NA NA NA NA ...
> lrn.msng <- is.na(lrn)                # this is how you do it
> str(lrn.msng)
 logi [1:100]  TRUE  TRUE FALSE  TRUE  TRUE FALSE ...
```

Summaries of data that have NA's

Applying a summary function, such as **`mean`**, **`median`**, or **`var`**, to data with any **`NA`**'s (or **`NaN`**'s) will return **`NA`** (or **`NaN`**).

If you want the value of the summary function after excluding the NA's, you must exclude the NA's then do the summary. Several summary functions allow an argument **`na.rm = TRUE`** that causes this to be done automatically.

```
> mean(lrn)                          # missing data propagate in summaries
[1] NaN
> mean(lrn[!is.na(lrn)])             # the complicated way
[1] -0.6182534
> mean(lrn, na.rm = TRUE)            # the simpler way
[1] -0.6182534
```

The recycling rule

If `a` and `b` are vectors of the same length n then `a*b` is the elelement-by-element product.

What if they are not the same length?

- It is clear that `2*b` should be the vector whose elements are twice those of `b`, so the `2` must be repeated n times.
- Generalising this, the shorter of the two arguments is always repeated to make it as long as the longer argument. If this is not an exact multiple a warning is given.

```
> 1+1:6
[1] 2 3 4 5 6 7
> 1:2+1:6
[1] 2 4 4 6 6 8
> 1:4+1:6
[1] 2 4 6 8 6 8
Warning message:
longer object length is not a multiple of shorter object length in: 1:
```

Matrices and other arrays

R provides several linear algebra operations on matrices. Although matrices and data frames seem similar, their underlying structure is different. Matrices are homogeneous (i.e. all elements are the same type) whereas data frames can be heterogeneous with elements of different types - numeric, factors, ordered factors, . . .

Matrices are created with the **`matrix`** or **`array`** functions. They are stored in *column major* ordering, which means the first column, followed by the second column, ... The **`array`** function can be used to create multi-dimensional arrays.

```
> str(rmat <- matrix(rnorm(12), nrow = 4)) # matrix of 4 rows and 3 co
 num [1:4, 1:3] -0.553  1.345 -1.359  1.843  1.622 ...
> rmat                                    # column-major ordering
           [,1]       [,2]        [,3]
[1,] -0.552920  1.6217560 -0.04442308
[2,]  1.345163  1.0104360 -0.80445109
[3,] -1.359278 -0.4599424  0.50111019
[4,]  1.842592  0.0521774 -0.70017661
```

Graphical capabilities

One of the strengths of the **S** language is graphics.

- Simple, exploratory graphics are easy to produce.
- Publication quality graphics can be created.
- Several device drivers are available including:
  - On-screen graphics - either Windows, or X11, or Macintosh
  - postscript - PostScript graphics commands
  - pdf - Adobe Portable Document Format
  - png - PNG bitmap device (like .gif but free of software patents)
  - jpeg - JPEG bitmap
  - WMF - Windows meta-file (Windows only)

Declaring graphics devices

The on-screen devices are the most commonly used. For publication-quality graphics the postscript, pdf, or WMF devices are preferred because they produce scalable images. Use bitmap devices only when there is no alternative.

The preferred sequence is to specify a graphics device then call graphics functions. If you do not specify a device first, the on-screen device is started.

A contributed **R** package called **`lattice`** provides Trellis graphics functions. When using lattice it is important to declare the device using **`trellis.device`** before issuing graphics commands.

Types of graphics functions

High-level - functions such as **`plot`**, **`hist`**, **`boxplot`**, or **`pairs`** that produce an entire plot or initialize a plot.

low-level - functions that add to an existing plot created with a high-level plotting function. Examples are **`points`**, **`lines`**, **`text`**, **`axis`**, . . .

Trellis functions - functions such as **`xyplot`**, **`bwplot`**, or **`histogram`** that can produce an entire multipanel display in a single call.

After creating a new plot with a high-level plotting function, you can add to the plot by making calls to low-level plotting functions. You cannot, however, do this after a trellis function call.

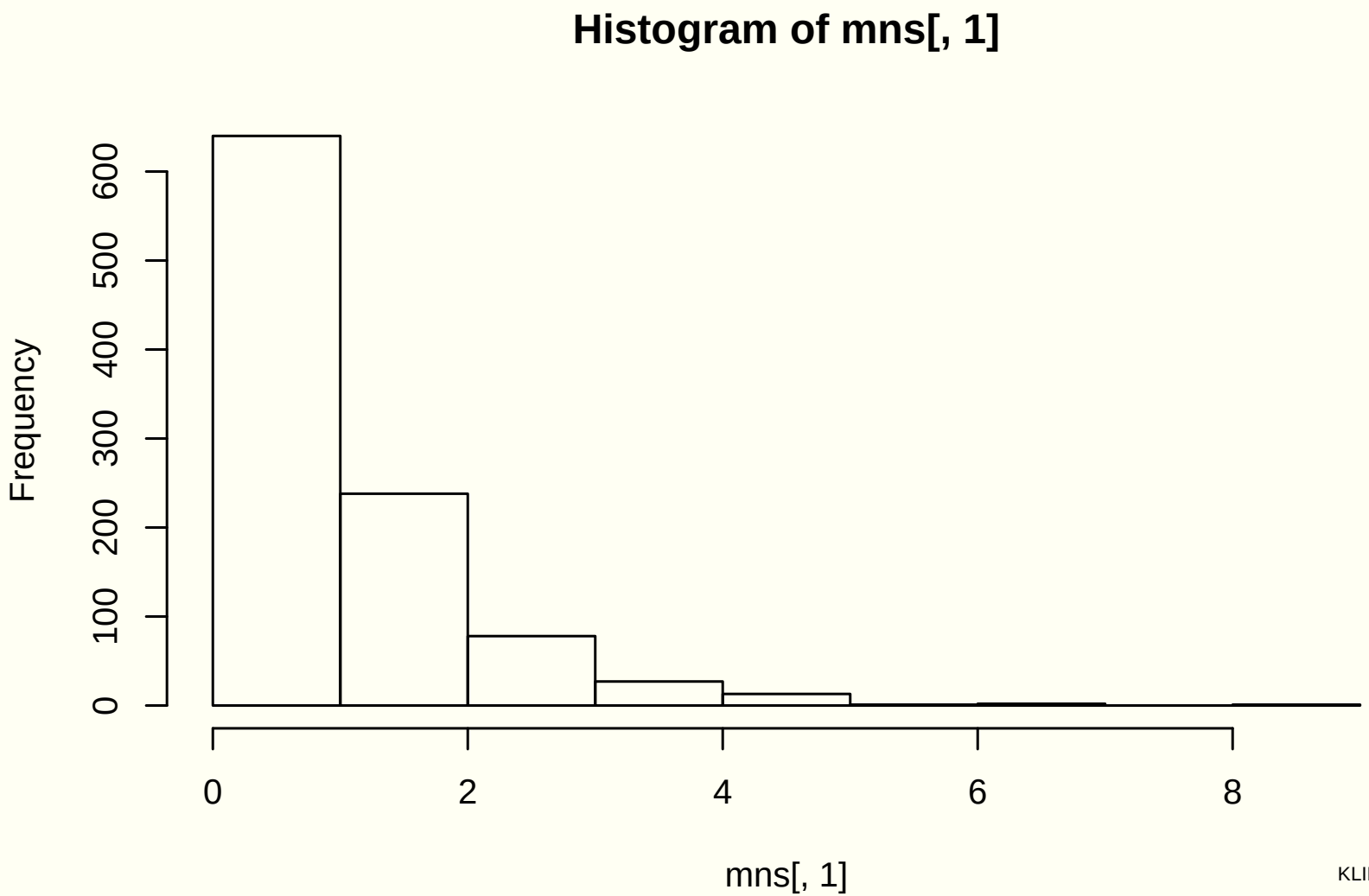
An example

It is common to illustrate the central-limit effect by computing means of samples from a nonsymmetric distribution and showing that the distribution of the mean tends to a normal distribution as the sample size increases. This is best illustrated with graphical displays.

```
> # generate the samples as a matrix
> rmt <- matrix(rexp(1000 * 16), nrow = 16)
> mns <-              # Apply the mean function to columns
+    cbind(rmt[ 1, ],                      # means of samples of 1
+          apply(rmt[ 1:4, ], 2, mean), # means of samples of 4
+          apply(rmt[ 1:16, ], 2, mean) # means of samples of 16
+          )
> meds <-             # Apply the median function to columns
+    cbind(rmt[ 1, ],                        # medians of samples of 1
+          apply(rmt[ 1:4, ], 2, median), # medians of samples of 4
+          apply(rmt[ 1:16, ], 2, median) # medians of samples of 16
+          )
```

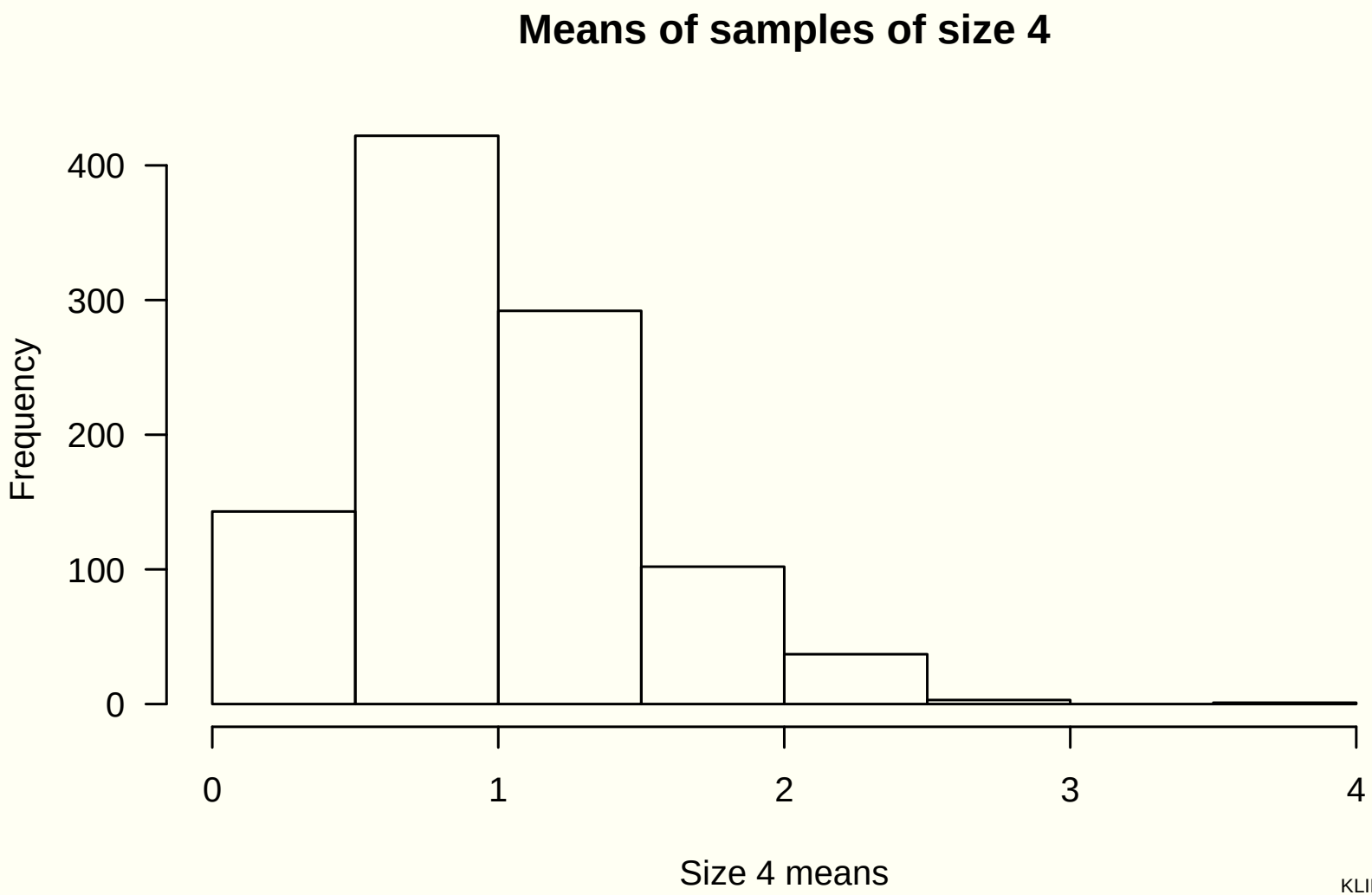
Using high-level plotting functions

```
> hist(mns[, 1])          # a histogram of the means of samples of 1
```



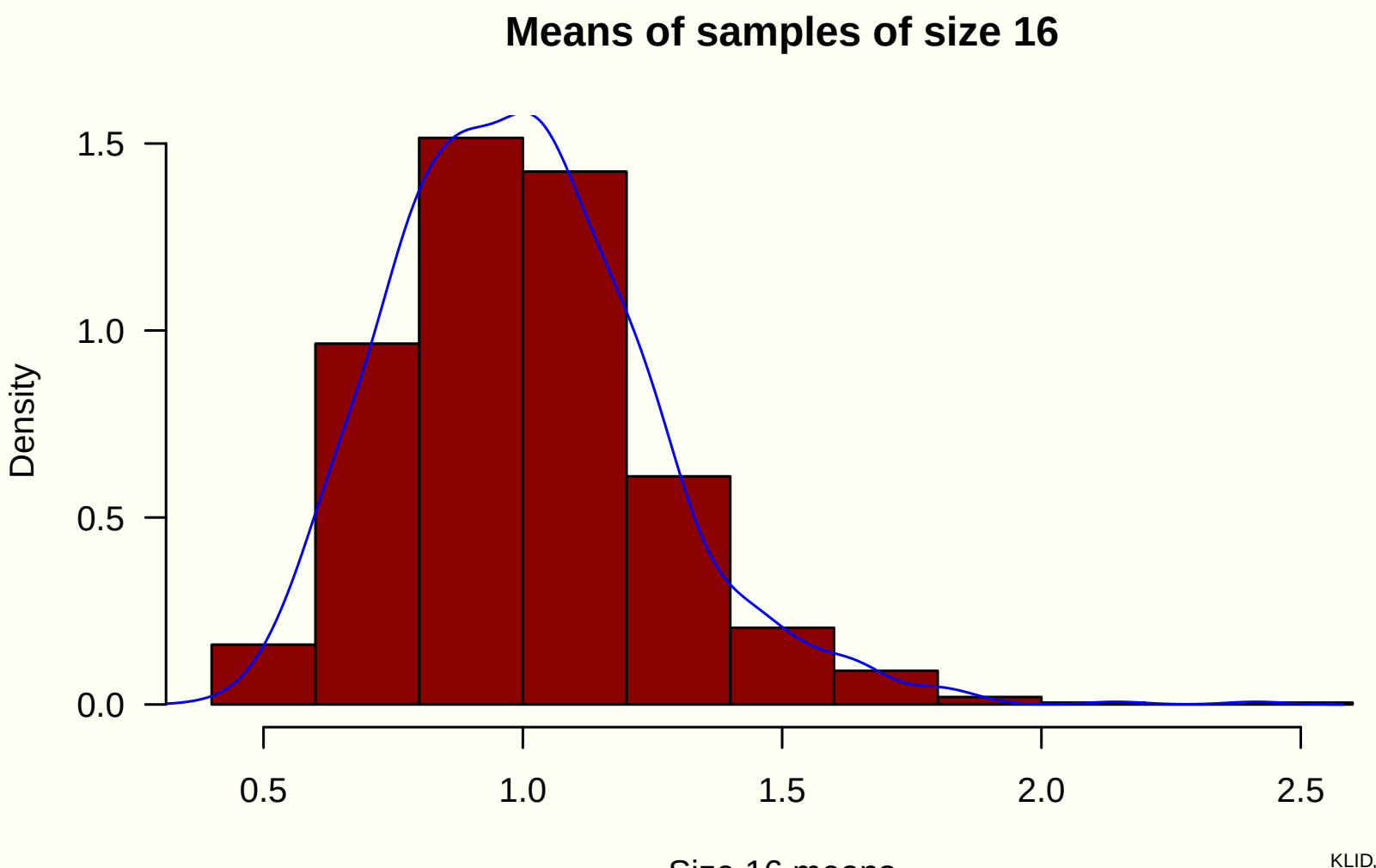
Enhancing high-level plots

```
> hist(mns[,2], main = "Means of samples of size 4",
+      xlab = "Size 4 means", las = 1)  # sets the axis label style
```



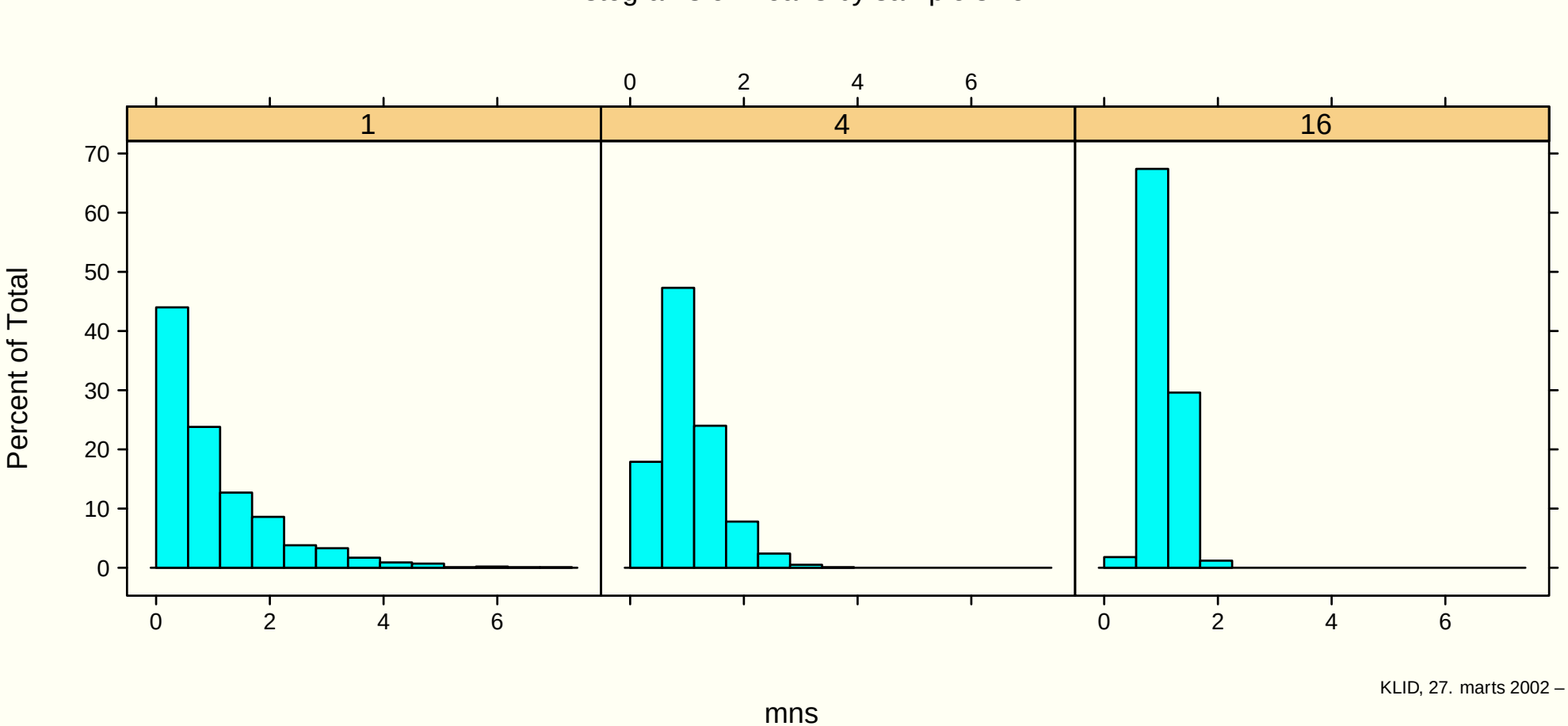
Using low-level graphics functions

```
> hist(mns[,3], main = "Means of samples of size 16",
+      xlab = "Size 16 means", las = 1, col = "darkred", prob = TRUE)
> lines(density(mns[,3]), col = "blue")
```



Using lattice graphics

```
> library(lattice)
> histogram(~ mns | ssz, data = data.frame(mns = c(mns),
+                           ssz = gl(3, 1000, labels = c("1", "4", "16"))),
+           layout = c(3, 1), main = "Histograms of means by sample si
```



Using the formula-data specification

Most of the high-level **R** graphics functions allow a formula-data specification for the plot. In trellis-style graphics from the lattice package the formula-data specification is the only way to specify a plot.

A *formula* in **S** is indicated by the $\sim$ character. Because formulas are used to specify statistical models, this character is often read as “is modelled as”. The second argument in a formula-data specification is usually a data frame with variables corresponding to the names in the formula.

To use the formula-data specification, first construct a data frame with the data to be plotted. The preferred form is to “stack” all the data into a single column with accompanying columns that indicate the groups of observations.

Stacking the simulation data

Recall the simulated data of means and medians of samples of size 1, 4, and 16 from an exponential distribution. We arrange this into a data frame with 6000 rows and three columns - the simulated data, the sample size being simulated, and an indicator of mean or median.

The *gl* function can be used to generate patterned data like the sample size and the type of simulation.

```
> alldat <- data.frame(sim = c(mns, meds),
            ssz = gl(3, 1000, len = 6000, labels = c("1", "4", "16")),
            type = gl(2, 3000, labels = c("Mean", "Median")))
> str(alldat)
'data.frame':   6000 obs. of  3 variables:
 $ sim : num  0.934 0.200 1.866 1.074 1.283 ...
 $ ssz : Factor w/ 3 levels "1","4","16": 1 1 1 1 1 1 1 1 1 1 ...
 $ type: Factor w/ 2 levels "Mean","Median": 1 1 1 1 1 1 1 1 1 1 ...
```

Formulas specifying plots

The general form of a formula specifying a plot is

```
y ~ x | g
```

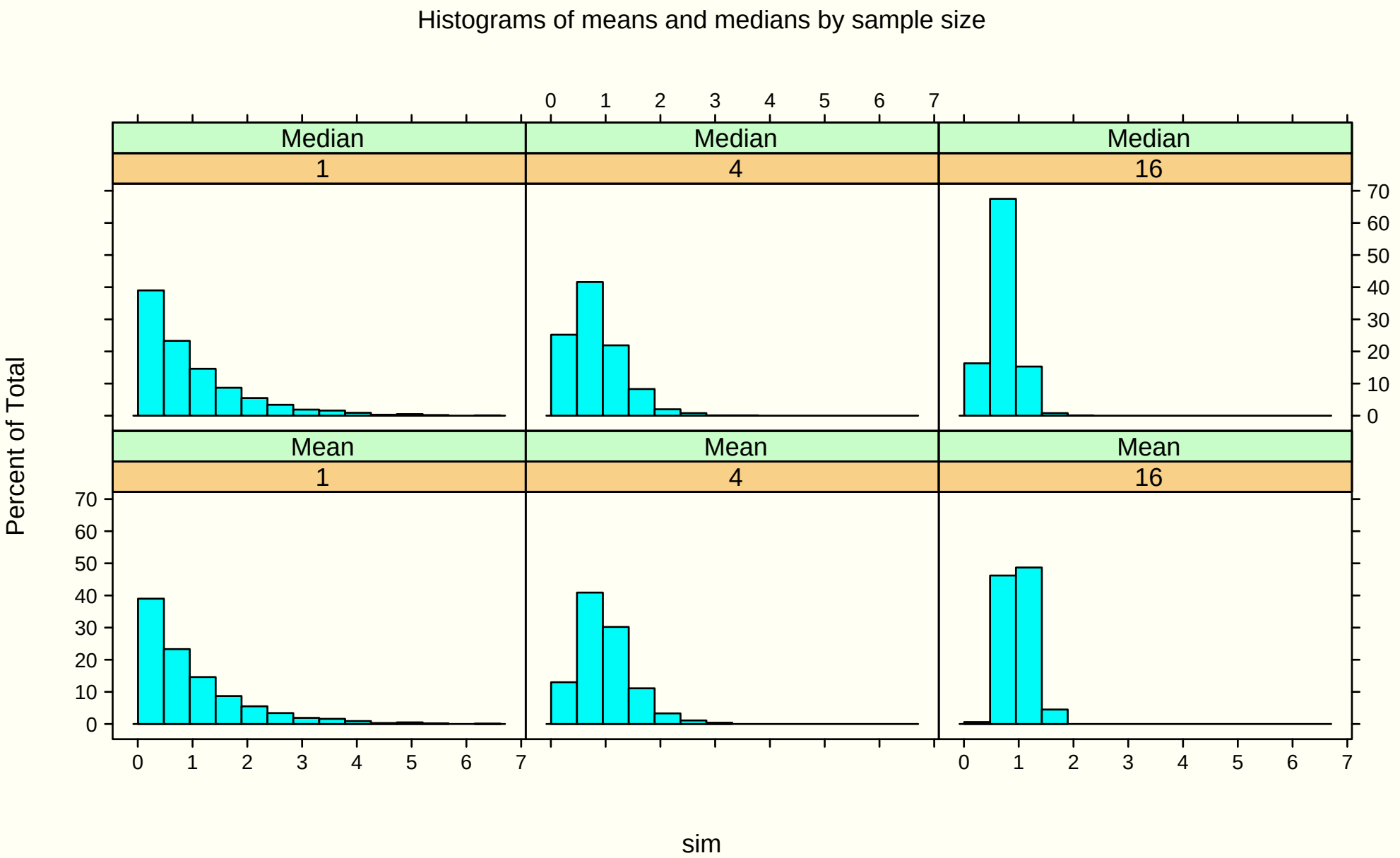
where **y** is assigned to the vertical axis, **x** is assigned to the horizontal axis, and **g** is a grouping factor or expression.

For special cases like the histogram, the vertical axis is pre-specified. In these cases we use a one-sided formula where **y** is omitted.

In trellis graphics functions the grouping expression can include multiple factors separated by an arithmetic operator - often the ***** because this indicates “crossing” the factors. For example,

```
histogram(~ sim | ssz * type, data = alldat,
    layout = c(3, 2),
    main = "Histograms of means and medians by sample size")
```

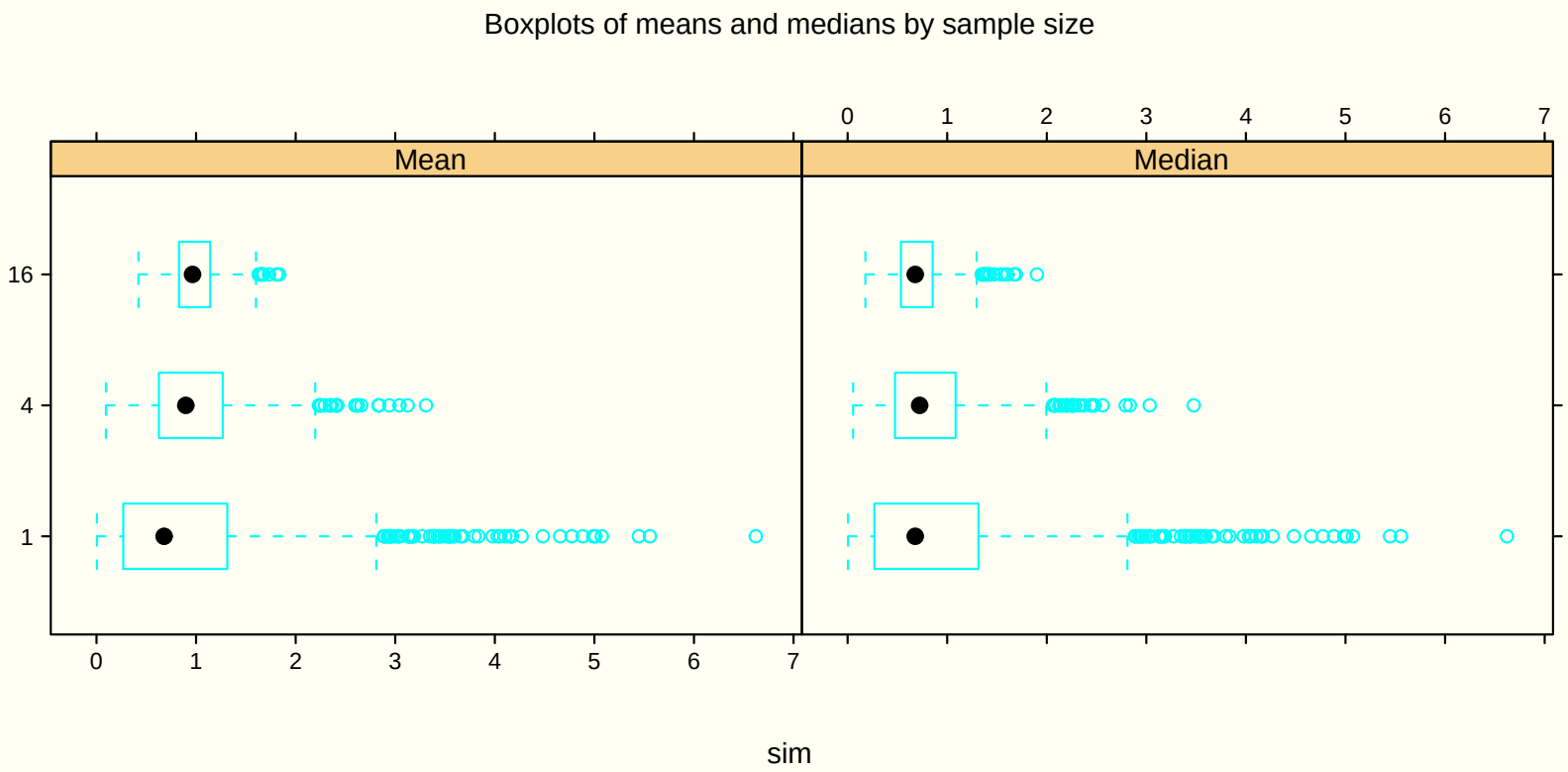
Example of multiple grouping factors



Boxplots

Another way of comparing distributions of sample data is with a **`boxplot`** (high-level graphics) or **`bwplot`** (lattice).

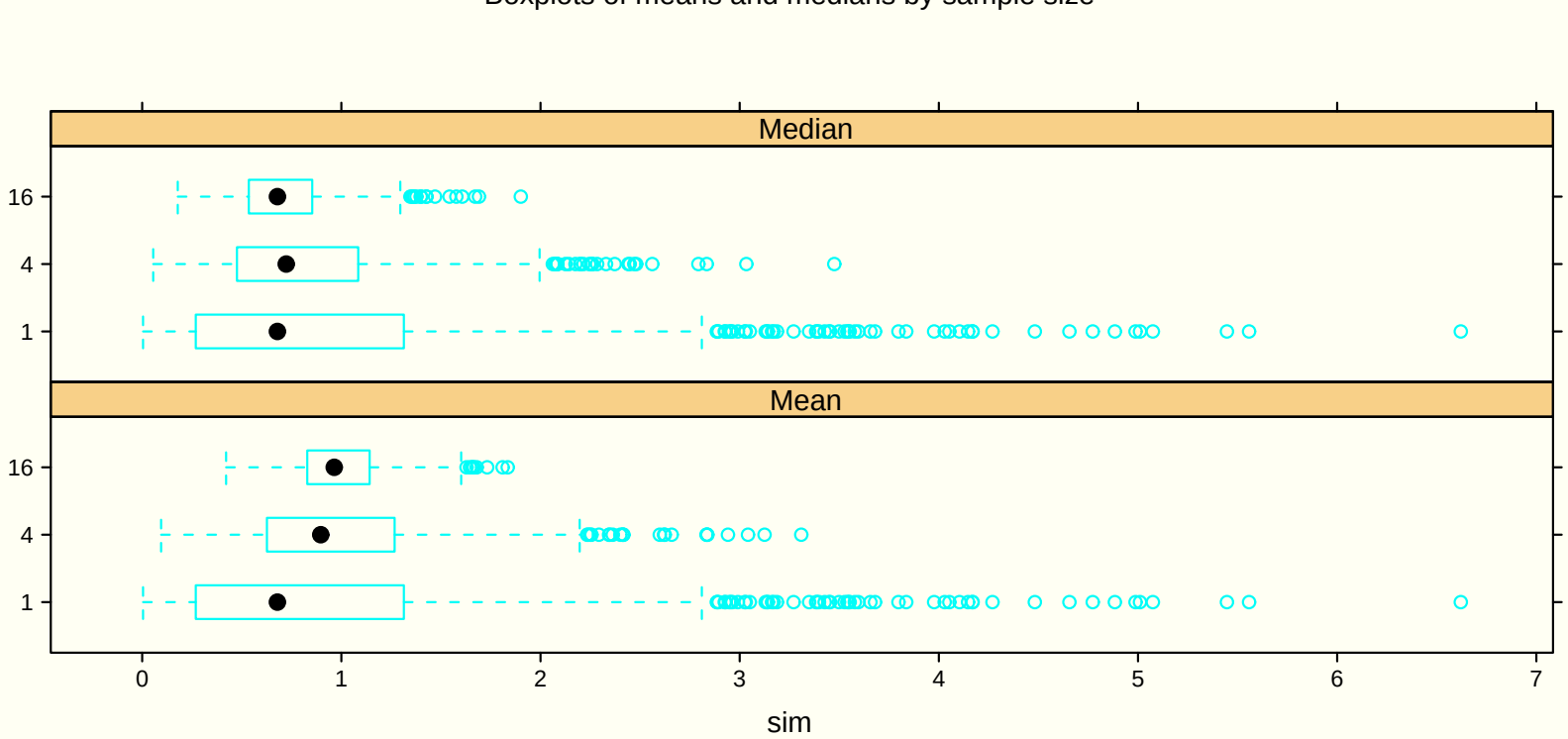
```
bwplot(ssz ~ sim | type, data = alldat,
       main = "Boxplots of means and medians by sample size")
```



Changing the lattice layout

The **`layout`** argument in lattice is used to rearrange the panels. It is a two-component or three-component integer vector in the order **`(columns, rows, pages)`** (not (rows, columns, pages)).

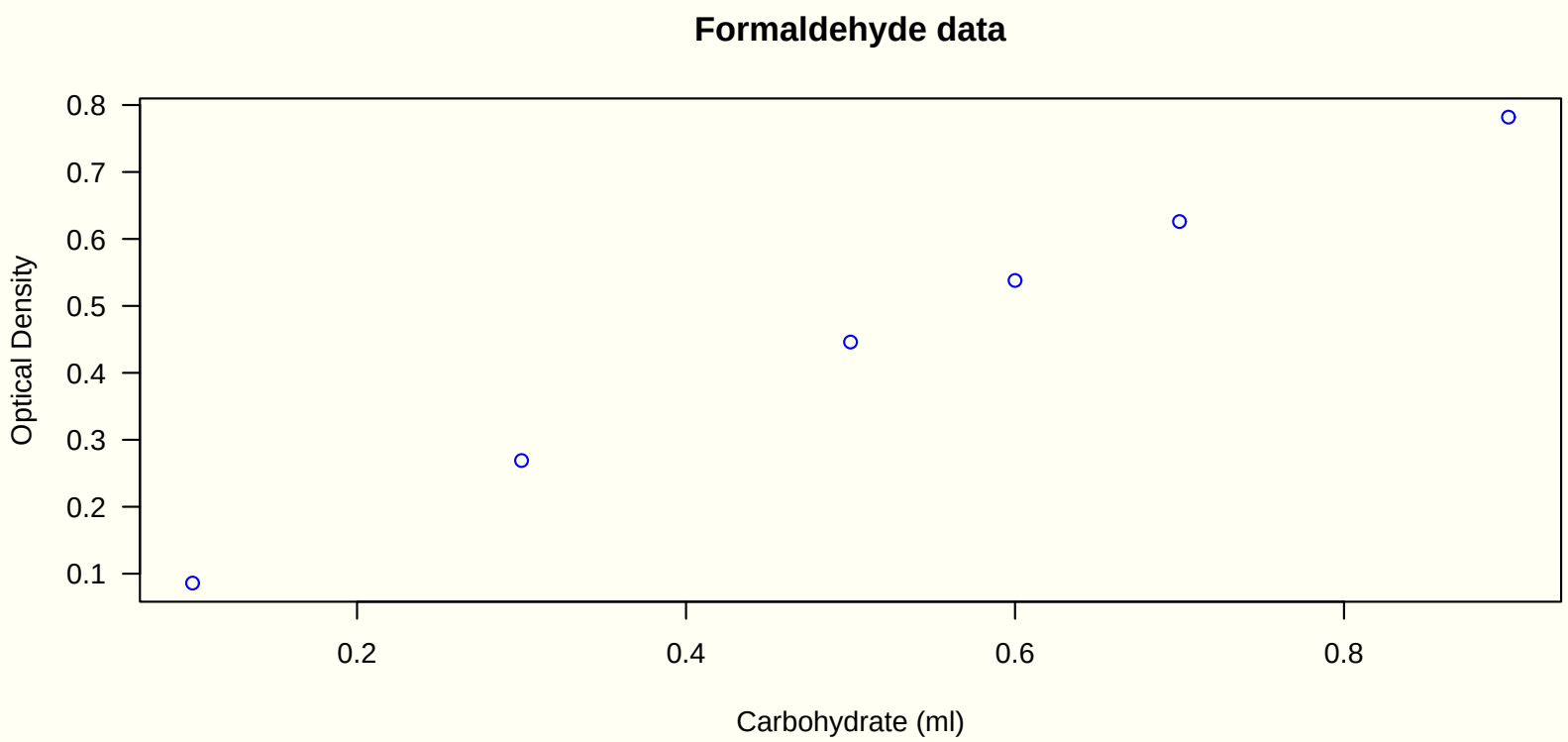
```
bwplot(ssz ~ sim | type, data = alldat, layout = c(1,2))
       main = "Boxplots of means and medians by sample size",
```



Scatter plots

A basic data plot is the scatter plot for x-y data. Recall

```
> data(Formaldehyde)
> plot(optden ~ carb, data = Formaldehyde, xlab = "Carbohydrate (ml)",
+    ylab = "Optical Density", main = "Formaldehyde data", col = 4,
+    las = 1)
```



Common additional graphics parameters

The call to **`plot`** on the previous slide included arguments

- **`xlab`** - x axis label
- **`ylab`** - y axis label
- **`main`** - main title on the plot
- **`col`** - color of the plotted points
- **`las`** - axis label style

Although not required it is common to use these arguments.

Usually it is easy to get a first cut at a data plot, which may be sufficient for exploratory graphics. Presentation graphics can require considerable experimentation with different settings, involving many cycles of editing and rerunning the code. Creating file of **R** code that can be edited and rerun using **`R BATCH`** is a good idea.

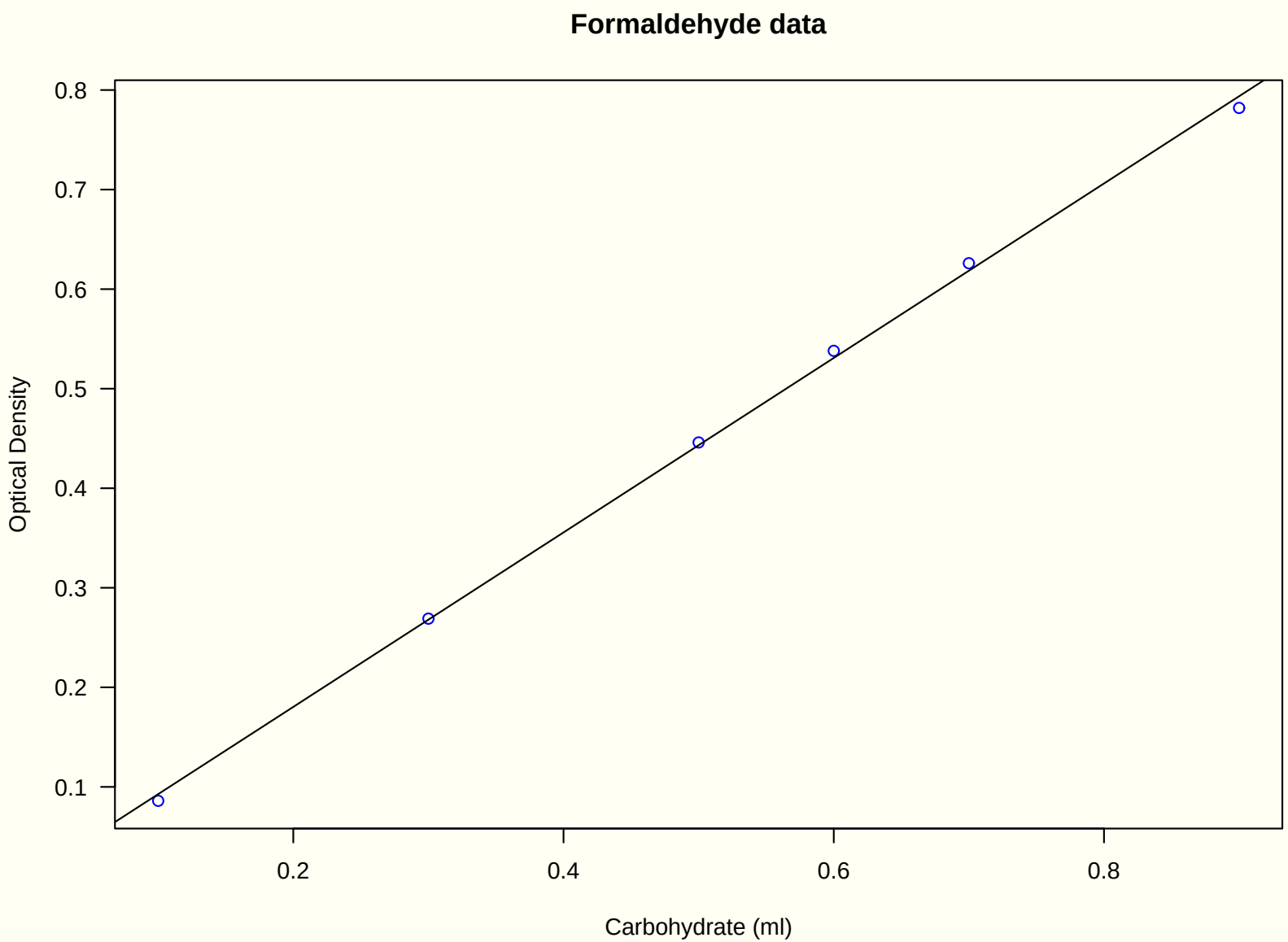
Adding lines to plots

We often use scatter plots to picture relationships between variables then procede to fit statistical models representing these relationships. The **`lines`** and **`abline`** functions can be used to add lines to a data plot depicting

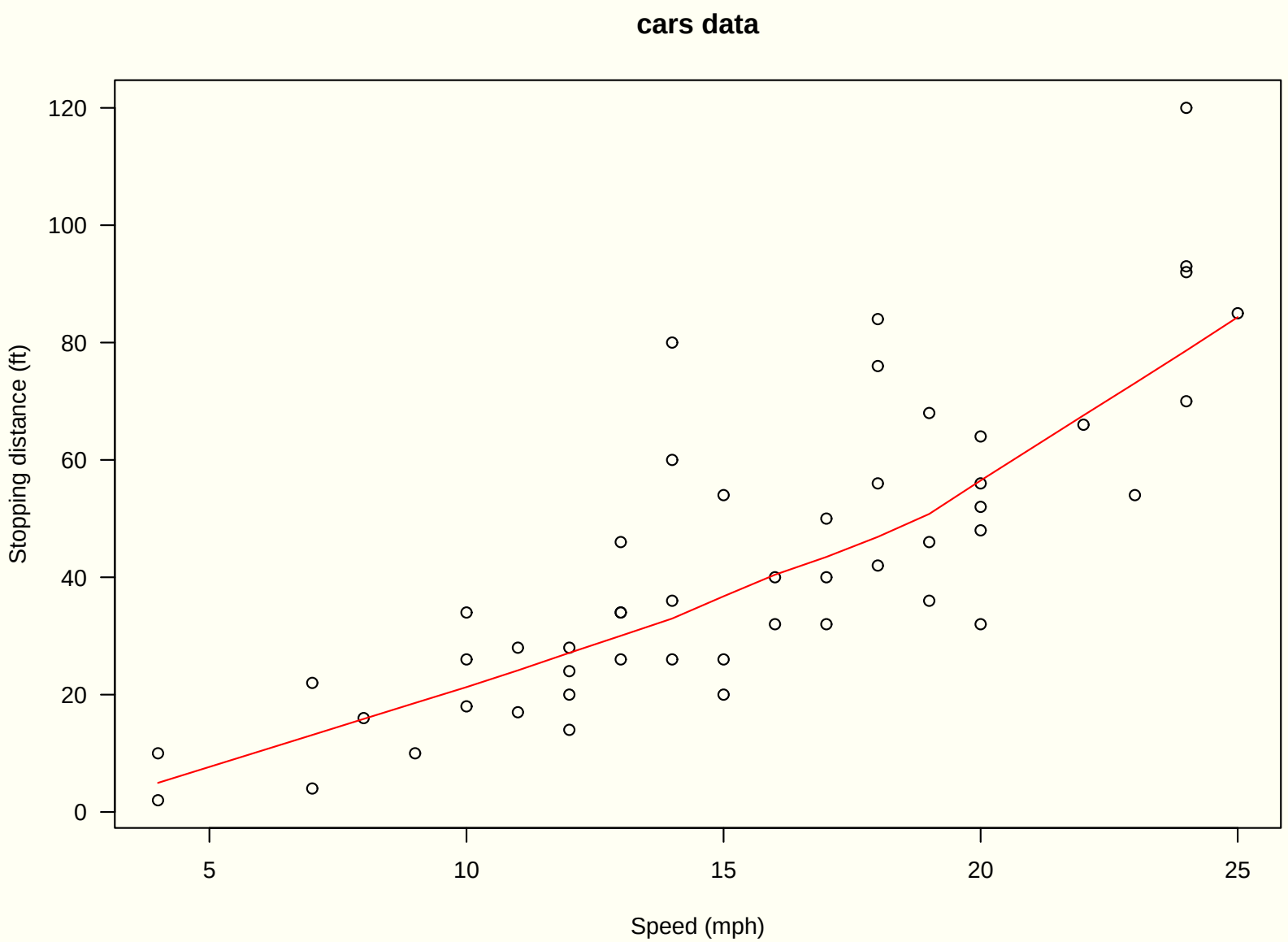
- **`scatterplot smoothers`** - a smooth curve generated from the **`y`** versus **`x`** data. This is added to enhance visualization of the **`y ~ x`** relationship.
- **`predictions from fitted models`** - a fitted simple linear regression model can be added directly to a plot with **`abline`**. For more complicated models, use **`predict`** to create **`(x, y)`** pairs to add to the plot with **`lines`**.

```
> abline(fm1 <- lm(optden ~ carb, data = Formaldehyde))
> data(cars); plot(cars, xlab = "Speed (mph)",
+   ylab = "Stopping distance (ft)", las = 1, main = "cars data")
> lines(lowess(cars$speed, cars$dist, f = 2/3, iter = 3), col = "red")
```

Adding a fitted model



Adding a smoothed curve



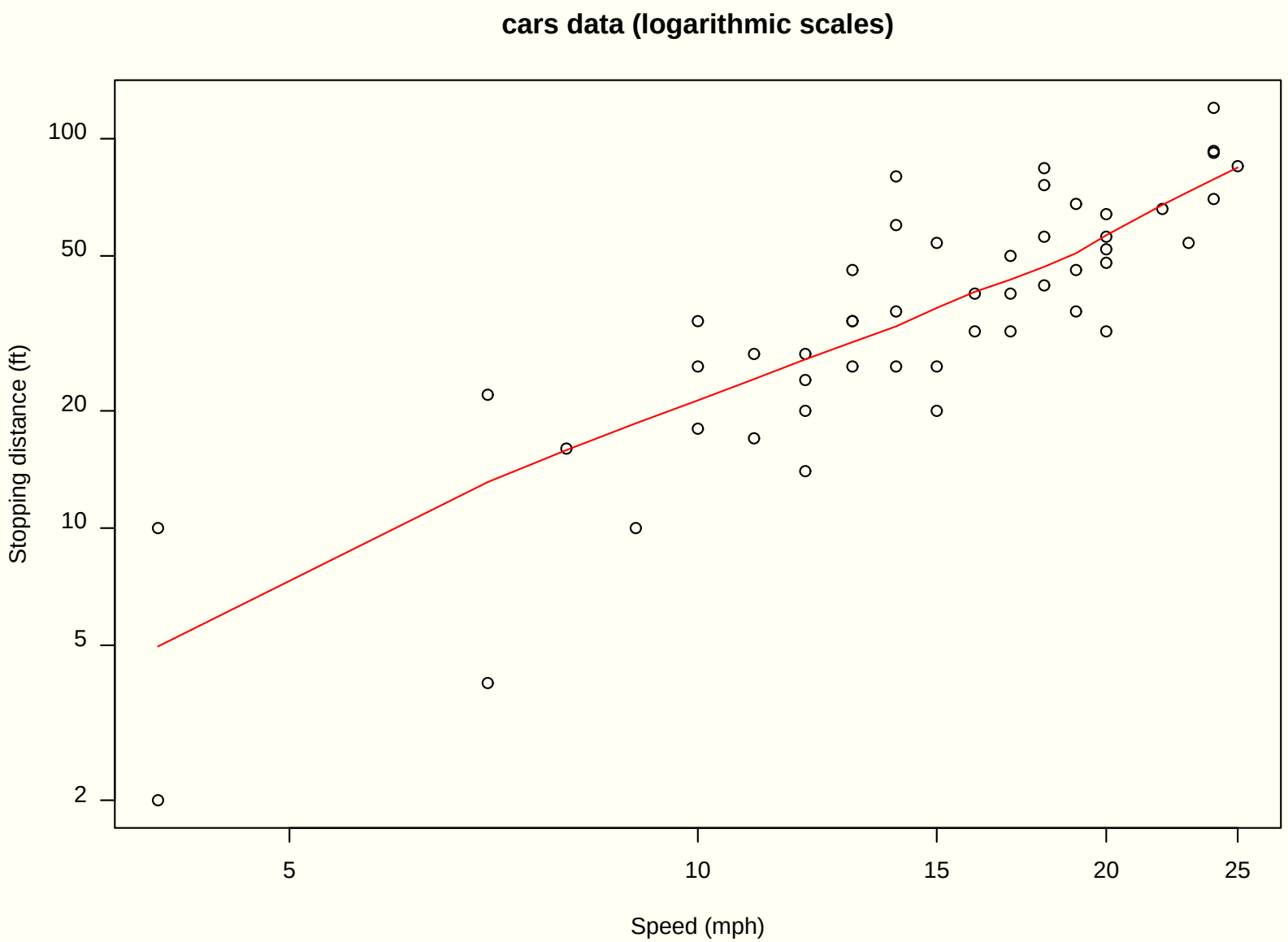
Logarithmic axes

The plot of the stopping distance versus speed for the **`cars`** data shows a curvilinear relationship. It also hints at increasing variance in the stopping distance as the speed (and the mean stopping distance) increase.

In cases like this a logarithmic transformation can stabilize the variance and perhaps produce a simpler relationship. The argument **`log`** is used to request logarithmic axes. It can take the values **`"x"`**, **`"y"`**, or **`"xy"`**.

```
plot(cars, xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
     las = 1, log = "xy", main = "cars data (logarithmic scales)")
lines(lowess(cars$speed, cars$dist, f = 2/3, iter = 3), col = "red")
```

Cars data - logarithmic scales

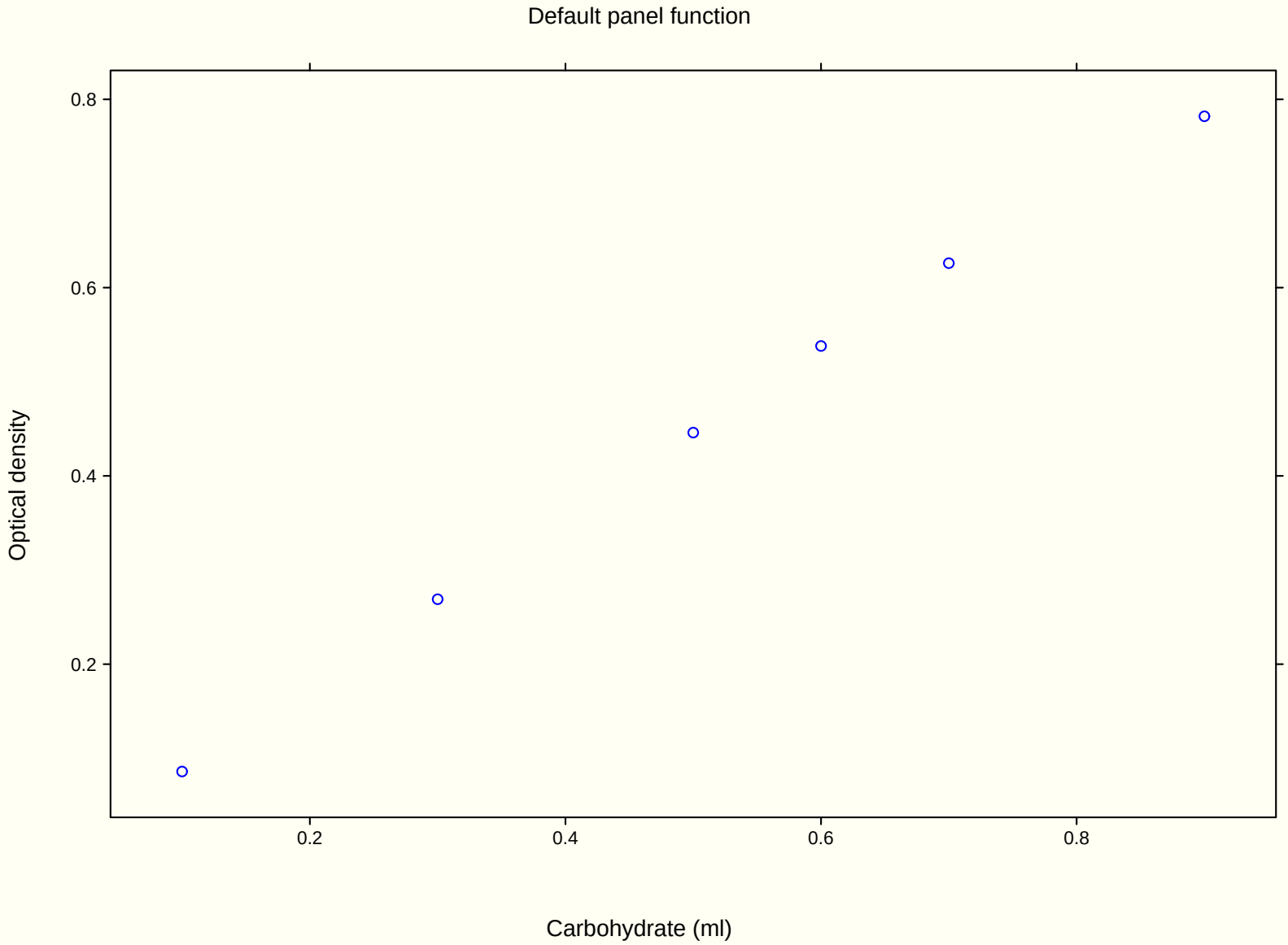


Scatterplots with lattice

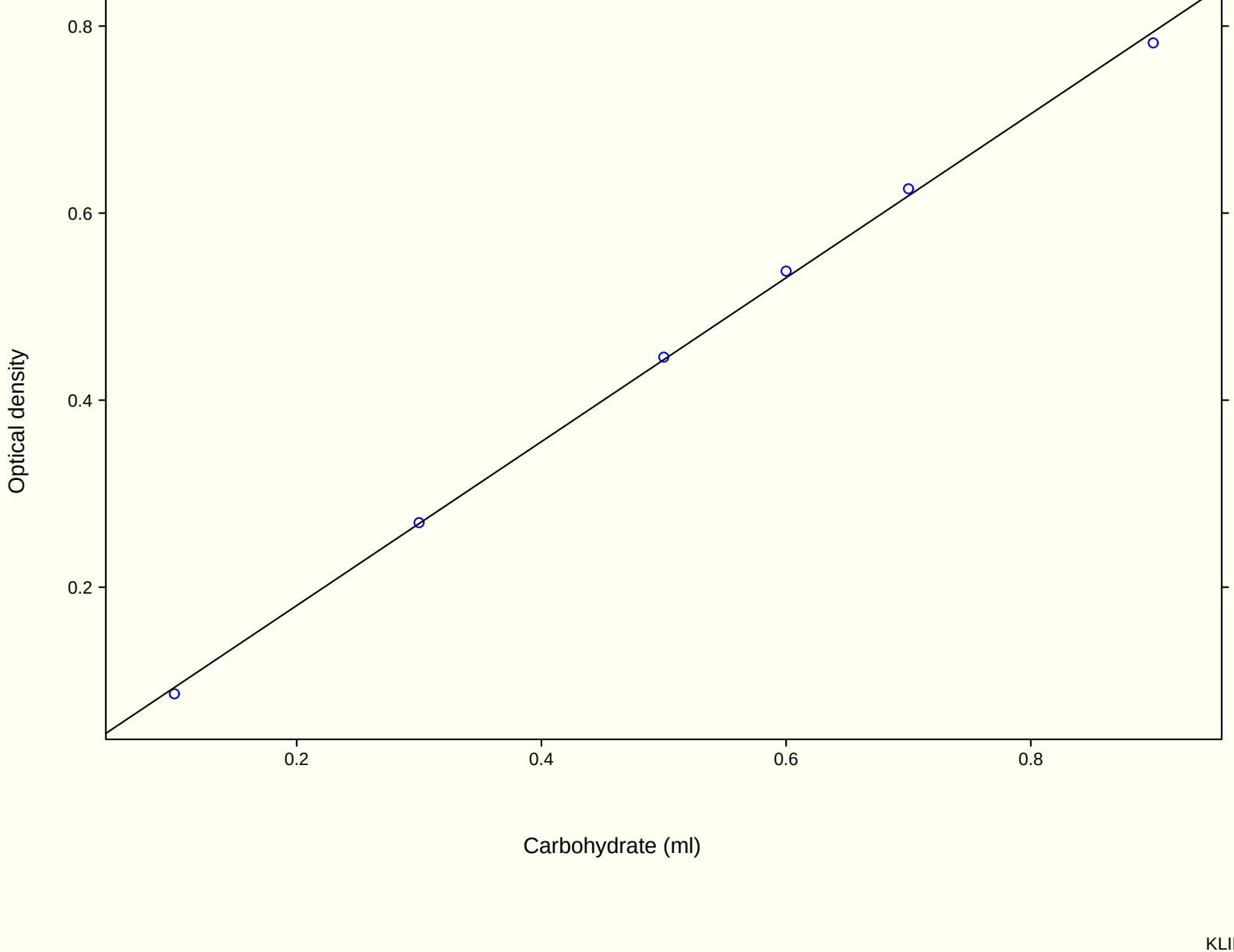
The lattice equivalent to **`plot`** is **`xyplot`**. You must use a formula-data specification with xyplot. Also, you must complete the plot in a single function call. Instead of using multiple calls to plot points then add scatterplot smoothers, etc., you describe all the actions for creating the plot in a *panel function* passed as the **`panel`** argument.

```
> xyplot(optden ~ carb, data = Formaldehyde, xlab = "Carbohydrate (ml)
+        ylab = "Optical density", main = "Default panel function")
> xyplot(optden ~ carb, data = Formaldehyde, xlab = "Carbohydrate (ml)
+        ylab = "Optical density",
+        panel = function(x, y) {panel.xyplot(x,y); panel.lmline(x,y)}
> xyplot(dist ~ speed, data = cars, xlab = "Speed (mph)",
+        ylab = "Stopping distance (ft)",
+        panel = function(x, y) {panel.xyplot(x,y); panel.loess(x,y)})
> xyplot(log(dist) ~ log(speed), data = cars, xlab = "log(Speed) (log(
+        ylab = "log(Stopping distance) (log(ft))",
+        panel = function(x, y) {panel.xyplot(x,y); panel.loess(x,y)})
```

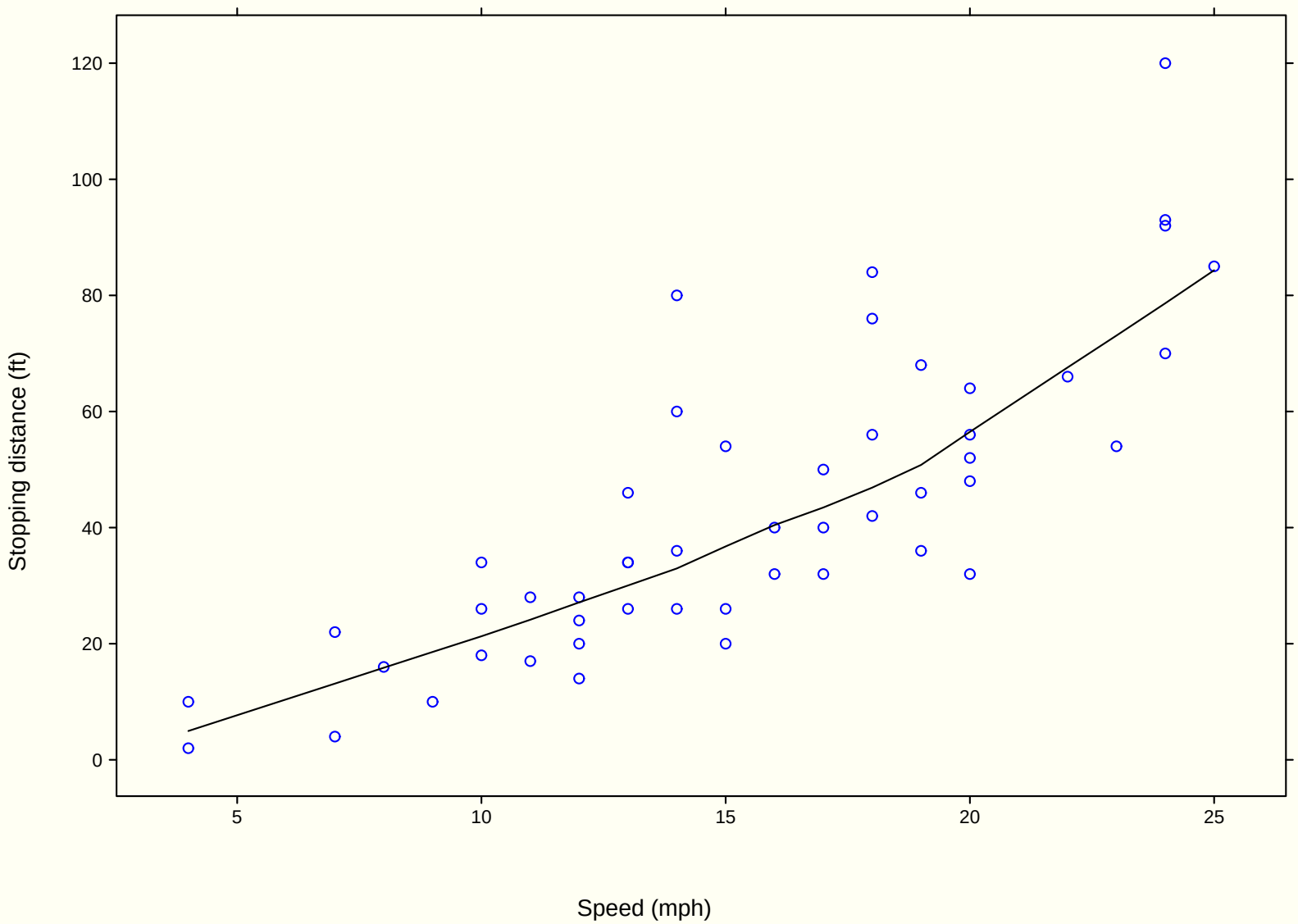
Formaldehyde xyplot with default panel



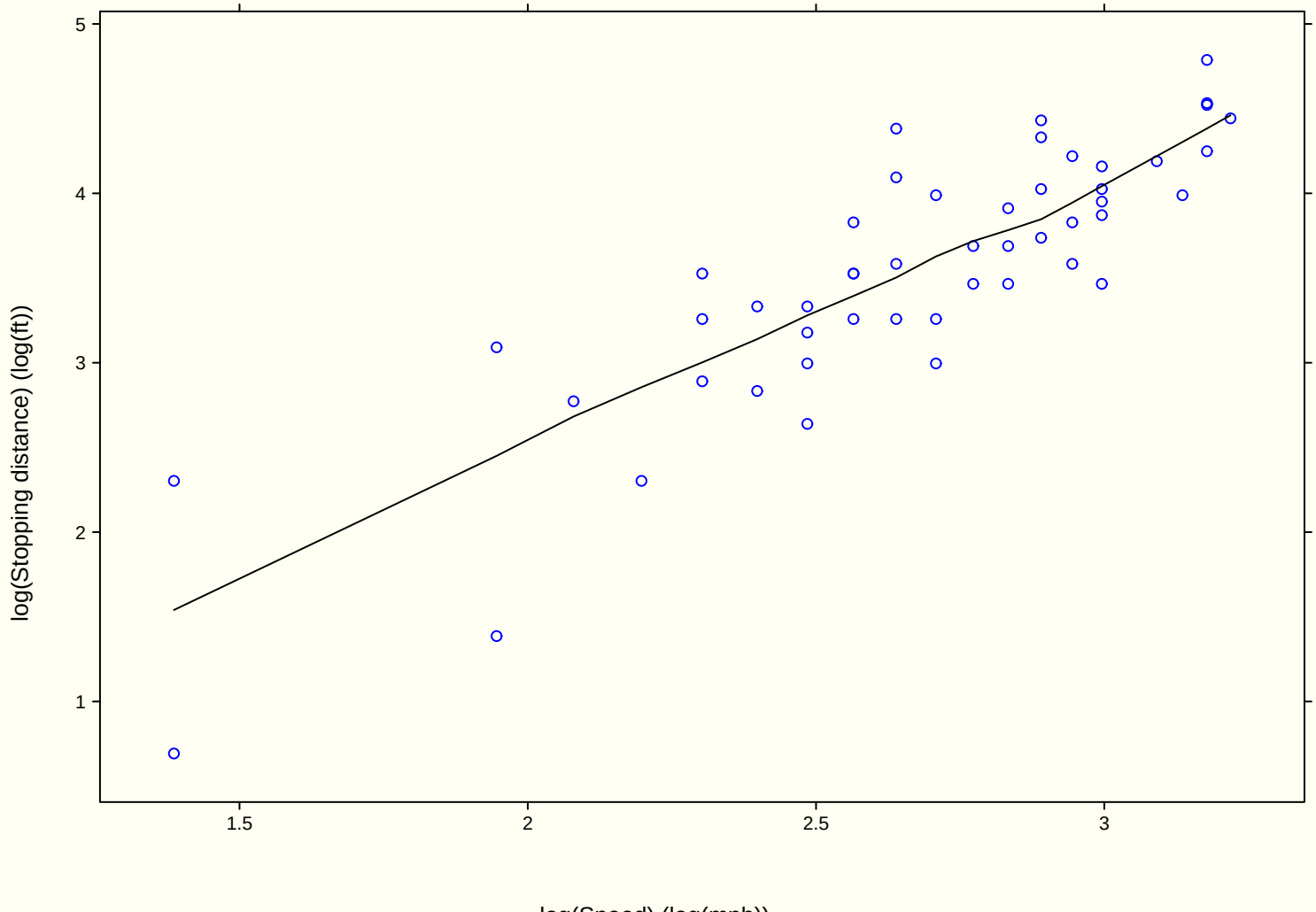
Formaldehyde xyplot with custom panel



Cars xyplot with custom panel



Cars xyplot on the logarithmic scale



There is a **`scales = list(log = TRUE)`** argument for **`xyplot`** but currently it has no effect.

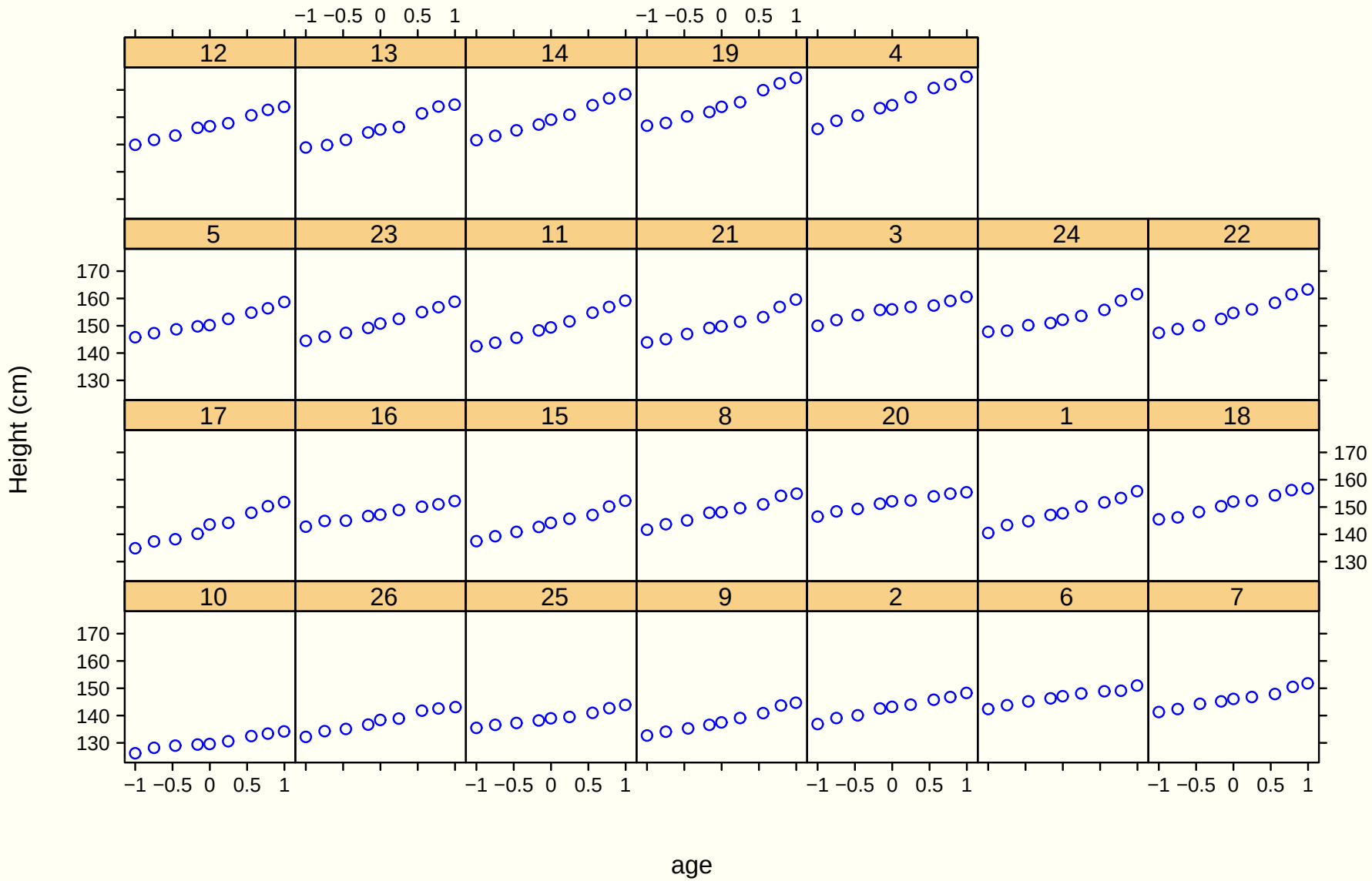
xyplots conditioned on a factor

One of the most powerful uses of trellis-style graphics is comparing patterns of responses across different groups. For example, the **Oxboys** data in the **nlme** package gives the height (cm) of some boys from Oxford, England. Each subject was measured several times throughout his adolescence. The time of measurement was converted to an arbitrary scale centered at the midpoint of the data.

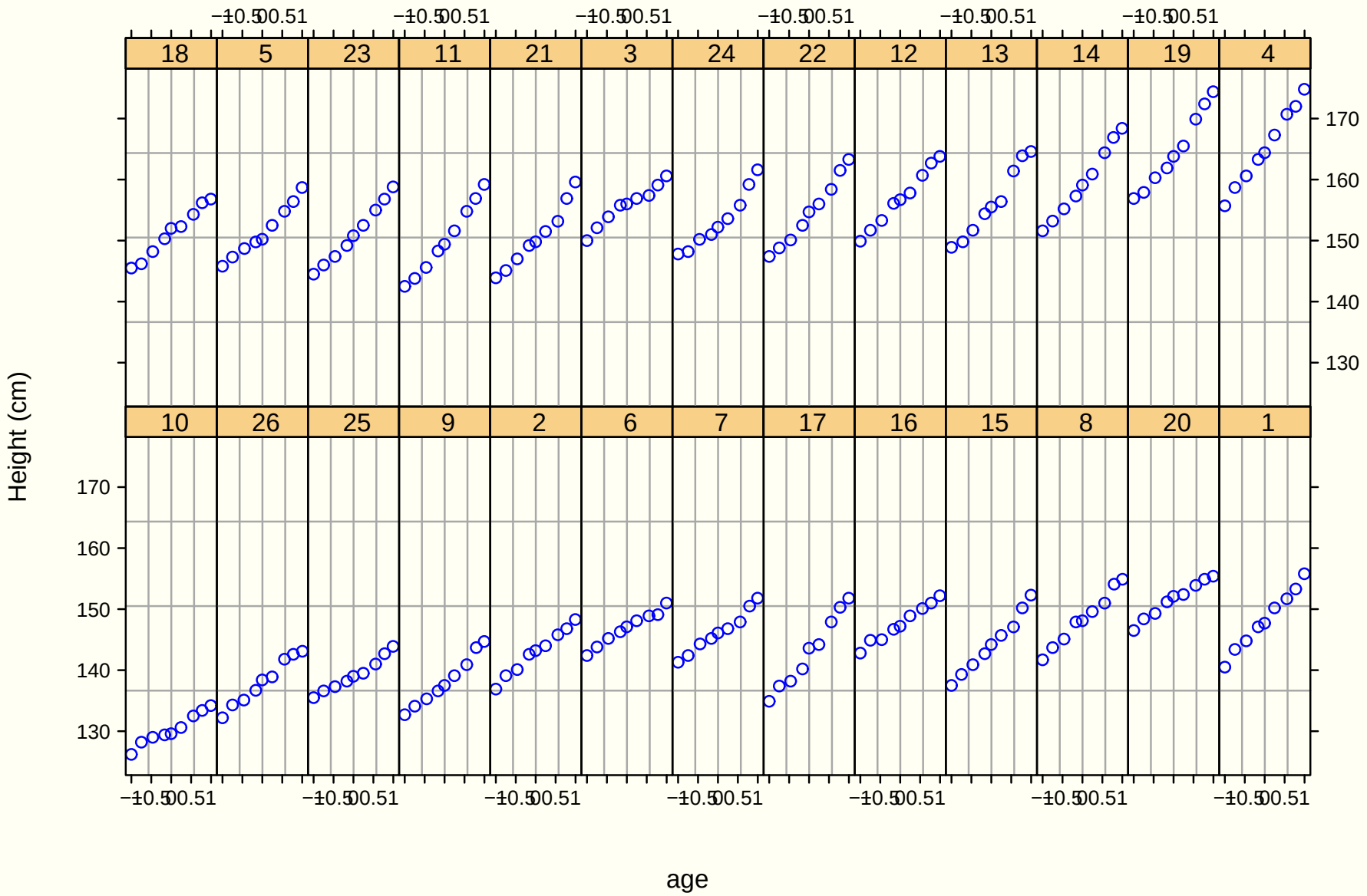
We wish to compare the growth patterns for these boys. To facilitate comparison, the data for each boy is plotted in a separate panel but scale of the axes is the same for each panel.

```
> data(Oxboys, package = nlme)
> xyplot(height ~ age | Subject, data = Oxboys, ylab = "Height (cm)")
> xyplot(height ~ age | Subject, data = Oxboys, ylab = "Height (cm)",
+        aspect = "xy",  # calculate an optimal aspect ratio
+        panel = function(x,y) {panel.grid(); panel.xyplot(x,y)})
```

Oxboys data, standard aspect ratio



Oxboys data, "xy" aspect ratio



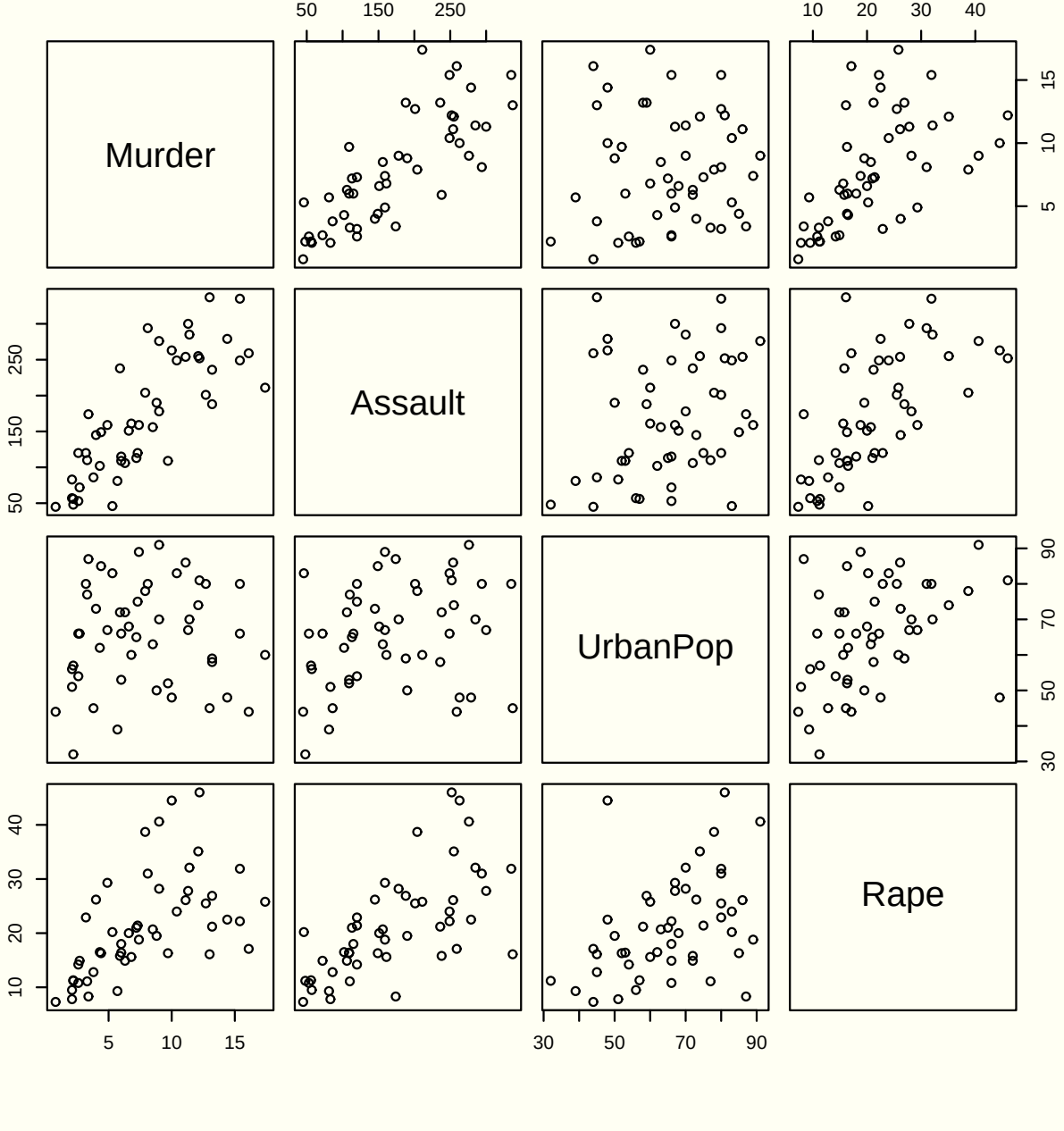
Scatterplot matrices

Scatterplot matrices are helpful in initial exploration of multivariate data. These provide scatterplots of all pairs of variables in the data and are produced by either the high-level graphics function **`pairs`** or the lattice function **`splom`**.

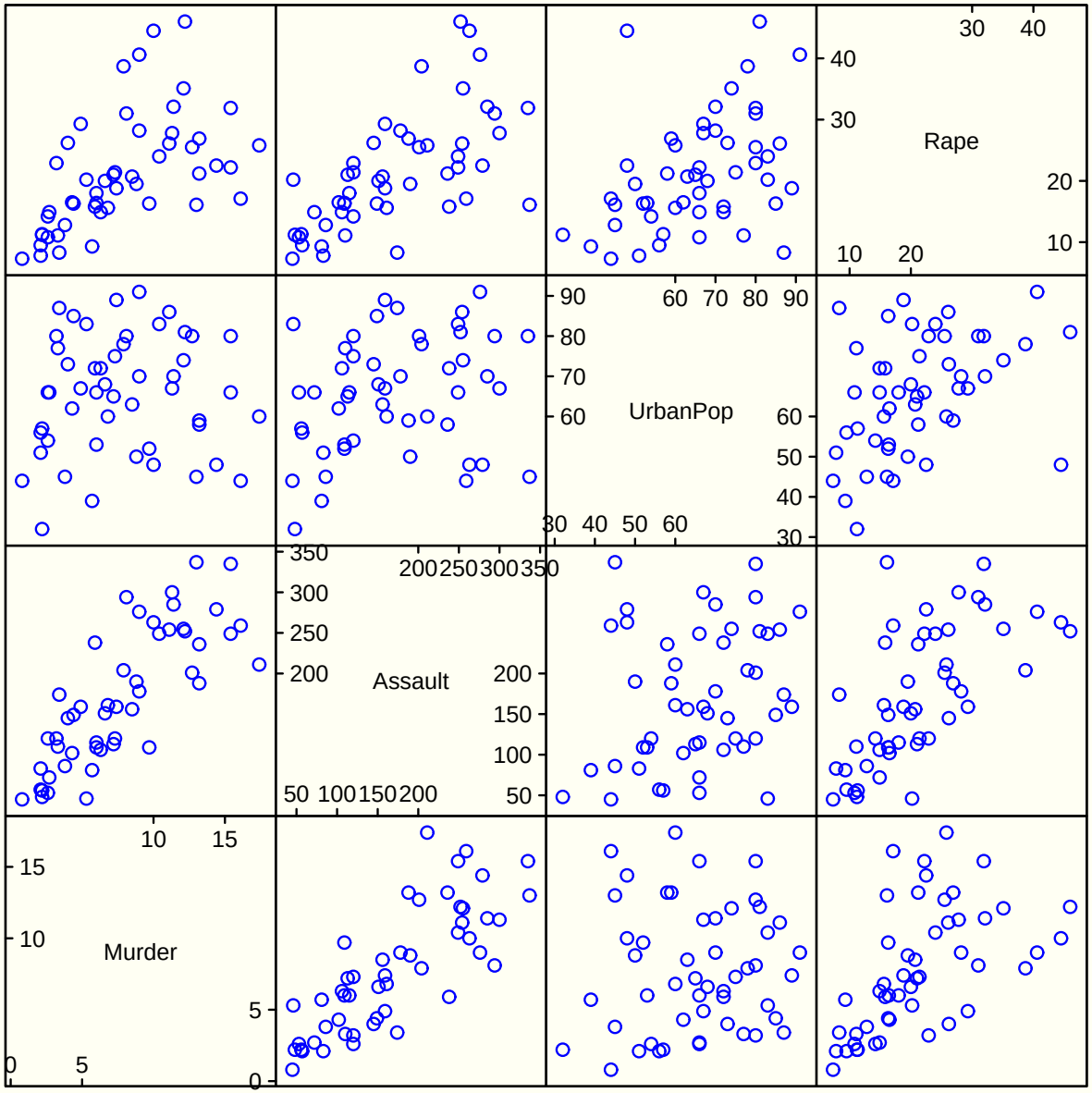
Both **`pairs`** and **`splom`** allow a panel function to be specified. Check the documentation for details.

```
> data(USArrests)
> pairs(USArrests)
> splom(~ USArrests)
> splom(~ USArrests,
+       panel = function(x,y) {panel.xyplot(x,y); panel.loess(x,y)})
```

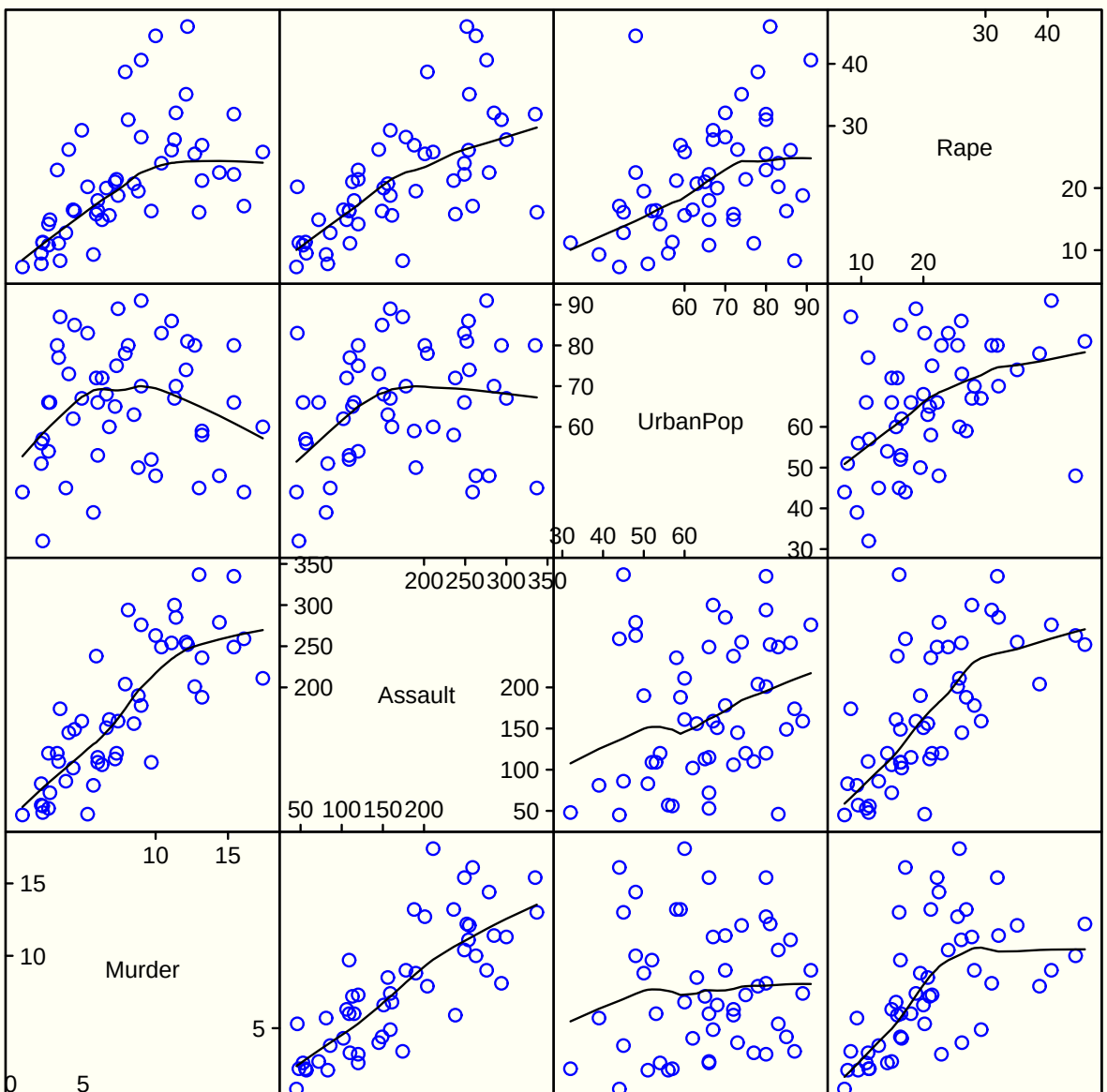
Pairs plot of USArrests data



Default splom plot of USArrests data



Enhanced splom plot of USArrests data

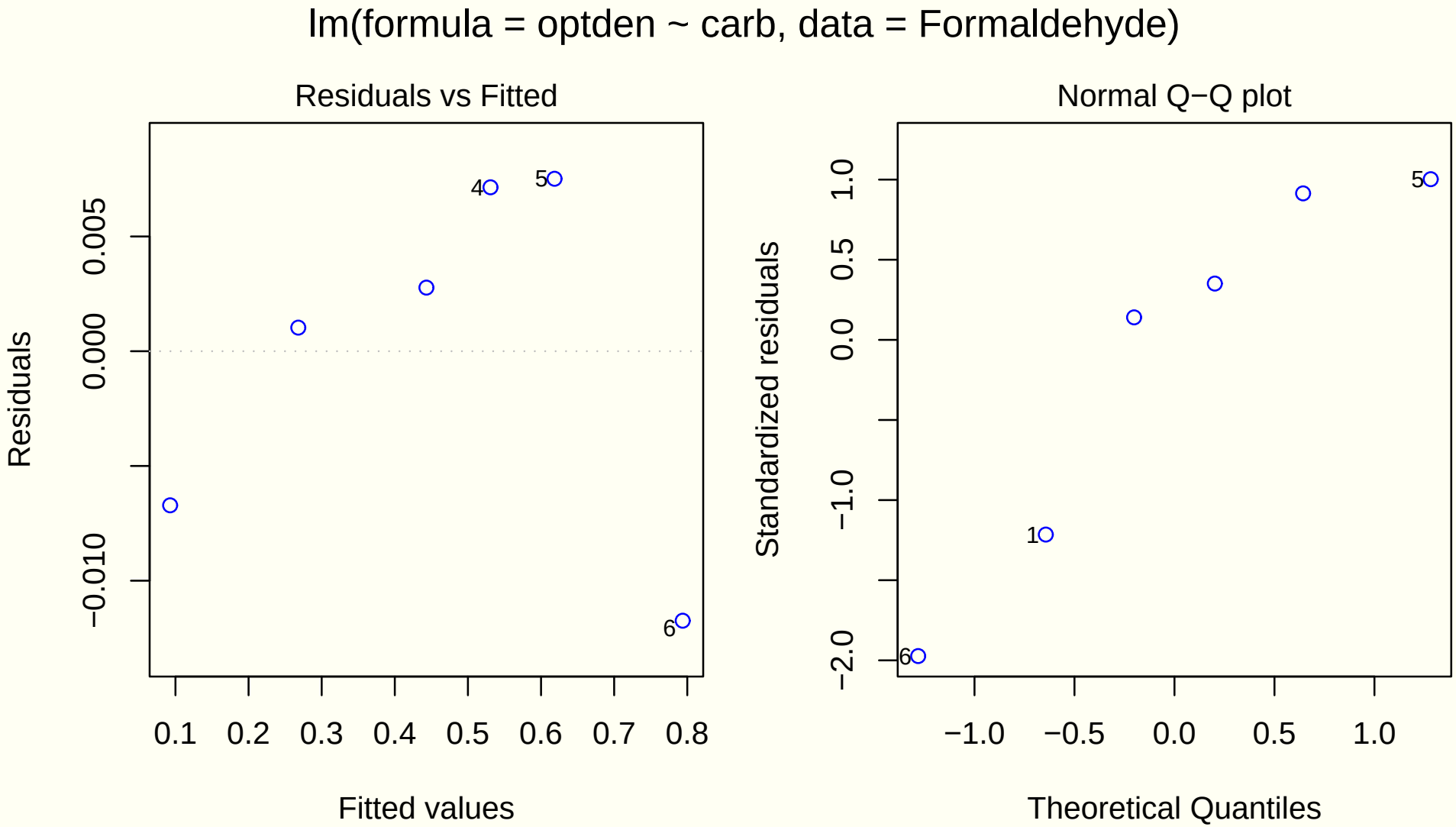


Multiple figures per page

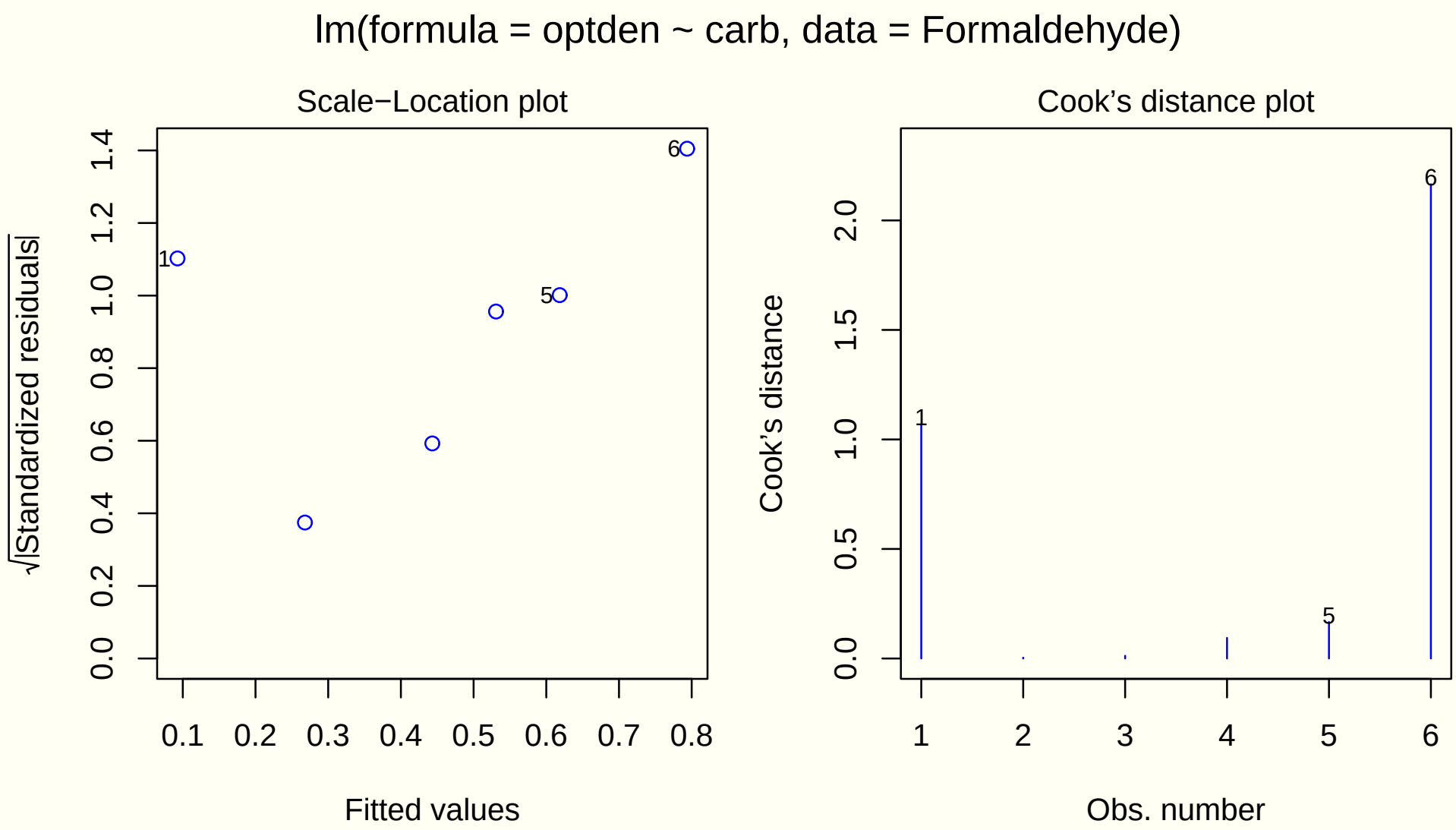
We have seen the use of the lattice package to produce multipanel plots where all the panels are related in some way. Occasionally we wish to put multiple unrelated figures on a page. In the high-level graphics, the **`par`** function can be used to set the graphics parameters **`mfrow`** (multiple figures created row-wise) or **`mfcol`** (multiple figures created columnwise) to do this.

```
> data(Formaldehyde)
> fm1 <- lm(optden ~ carb, data = Formaldehyde)
> par(mfrow = c(1,2))  # the next plot command produces 4 figures
> plot(fm1)
```

First two diagnostic plots



Second two diagnostic plots



Udviklingsmodellen for R

- Core Team (ca. 15 personer)
- Kvalitet er vigtigt
- Opnå gennem
  - Grundig testning (automatiserede metoder)
  - Konservativ politik for releases (2 per år), forgrenet udvikling
  - Hurtig fejlretning
- CVS – Concurrent Versioning System.
- Jitterbug
- Mailinglister

Kompatibilitet

- Unix/Linux – nemt (relativt)
- Windows – ikke helt så nemt
- Macintosh – Stefano Iacus/Jan deLeeuw: MacOS 8.6, OS X (Unix-agtig)
- Integreret kildetekst

Kontribuerede R pakker

- Hæve standarden for distribution af software, undgå “bitrot”
- Gøre pakker
  - Portable
  - Pålidelige
  - Lette at installere
  - Veldokumenterede
- Strategi
  - API (relativt) uforanderlig
  - Standardiseret pakkeformat
  - Forsyne programmører med værktøj til konsistenscheck
- CRAN – Comprehensive R Archive Network (pt. 154 pakker)

Grænseflader mellem sprog

- Kombinere software skrevet i forskelllige programmeringssprog
- **R** og **S-PLUS** har længe haft dynamisk linking af C og Fortran
- Andre sprog (Java, Python,. . .) kan være mere komplicerede
  - Kontrol over “event loop”
  - Objekthåndtering
- Eksempel: **R**-Tcl/Tk grænseflade

Tcl/Tk

```
button .a -text hello
pack .a
.a configure -command "puts Av!"
```

- Kompakt, shell-agtig struktur (Tcl sproget — Tool Command Language)
- “Geometry manager” `pack` anbringer `.a` i forældre-vindue
- *Widget command* `.a` med underkommandoer
- Indlejring i **R**: Få kommandoer til at ligne **R** funktioner
- Særlige problemer: Callbacks, variable

Databasegrænseflader

- Ikke alle datasæt håndteres hensigtsmæssigt af statistisk programmel med den traditionelle cases $\times$ variable data matrix.
  - Bloodflow-målinger over 5 minutter ved 10 Hz i 4 kanaler.
  - Bevægelser på bankkonti.
- Relationsdatabaser
  - Koblede “tabeller”
  - De facto standard Structured Query Language (SQL)
  - Flerbrugeradgang
  - Atomiske transaktioner
- Adgang fra **R**/**S**
  - Findes interfaces til Postgres, MySQL, mSQL, ODBC, Oracle
  - Proxy dataframes

Udfordringer

- Økonomiske tidsrækker
- Store databaser, “live” data
- Gen-ekspression microarray chips
- Implementation af statistiske modeller
  - Fx. hierarkiske varianskomponentmodeller
  - Minimere dataredundans
  - Undgå brug af internt lager

Sweave

- Literate programming, cf. Knuth (TeX)
- Blande kode og dokumentation
- Web, tangle, weave
- cweb, fweb, noweb
- Sweave (Friedrich Leisch, Wien)
  - Inkludere **R** kode i et dokument
  - Generere analyser, tabeller og figurer