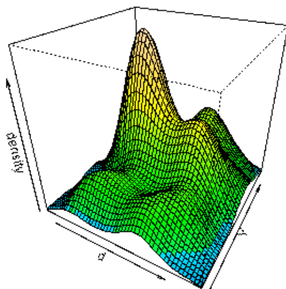
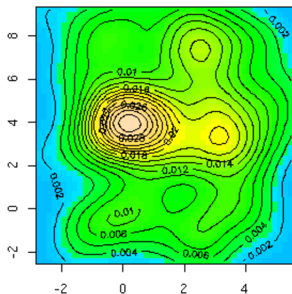
An Introduction to R - II

Functions and graphics

Olivier Flores





Scripting I

Scripting is the process of writing commands in external text files (= a script) with extension **.r** or **.R**

- Purposes:
  1. sequences of commands executed in a raw,
  2. keep trace of a working session,
  3. write personal functions that can be called any time
- Scripting requires a (good) text editor
  (avoid Notepad and Microft affiliated)
  For Windows,
  - PSPad: simple and free (`http://www.pspad.com/fr/`)
  - WinEdt: can be integrated with R, but **not free**,
  - **Tinn-R**: free and deeply integrated with R
    (can also be used with LaTeX)
    `http://www.sciviews.org/Tinn-R/`
  - ...

Scripting II

A simple example of script (to be adapted. . .)

```
# read data from file (suppose a data frame with 4 columns)
x=read.table(file="data.txt")
# check dimensions
print(dim(x))
# change columns names
names(x) =c("number","block","family","height","dbh")
# print summary of data
print(summary(x))
# check number of values in the factors and put result in y
y=apply(x[,c(2,3)], 2, table)
print(y)
```

Note: The command `table` creates cross contingency tables

Scripting III

- Save the script in a `example.r` file,
- Launch (=source) the file from R:
  `> source(file_path/example.r) # file_path not needed if example.r in working directory`
- Watch the results!

Working with functions I

A **function** is like a computing machine which uses variables (the *arguments*) in a suite of commands (the *body*) and return some results (the *value*: note that not all functions return a result).

1. **Handling existing functions**
   - Get help!! `> ?function; help(function)`
   - List of arguments with defaults: `> args(function)`
   - See the body of a function:
     `> body(function)` or simply `> function`
     The body cannot be seen for all functions (*Internals*)

Working with functions II

2. **Writing functions**

   Functions are declared with a unique key-word: **function**

   - General declaration:

     ```
     function_name = function(arguments)
     {
           commands
           return(value) # optional*
     } #
     ```

     *If no *return*, the function returns the last evaluated expression
   - Functions can be:
     - saved in a text file (**.R** or **.r**) for further use
     - declared "on the fly" for immediate use (*function_ name* optional in this case):

       ```
       > lapply(L, function(x) 1/length(x)) # where L is a list
       ```

Working with functions III

A script for a simple function

```
test = function(x)
{
    if(is.numeric(x))
    {
        m = mean(x)
        return(m)
    }
    else
      print("Cannot compute mean for non-numerical vector")
}
```

Working with functions IV

- Save the script in a `test.r` file
- Source the file from R:
  `> source(`*file_path/test.r*`)`
- If no error message, the function is now present as an object in the workspace:
  `> ls() # verify that the object `*test*` does exist`
- ```
  > x=c(2,6,7,-8,9)
  > test(x)
  [1] 3.2
  > x=factor(x)
  > test(x)
  [1] "Cannot compute mean for non-numerical vector"
  ```
  ```

Working with functions V

- Comment your scripts using the character #,
- Debug your functions
  1. Use `browser()` to indicate a pause in the execution

```
test = function(x)
{
        if(is.numeric(x))
        {
                m = mean(x)
                browser()
                return(m)
        }
        else
            print("Cannot compute mean for non-numerical vector")
}
```

Resource the function and call it again...

Working with functions VI

2. - Use `debug(function_name)` to execute the function step by step
   `> debug(test)` and then use commands `ls(),...`
   or use `n, c, where, Q`:
   - `n` (or just return): advance to the next step,
   - `c`: continue to the end of the current context:
     e.g. to the end of the loop if within a loop, or to the end of the function
   - `where`: print a trace of all active function calls,
   - `Q`: exit the browser and the current evaluation and return to the top-level prompt.

   - Use `undebug(function_name)` to stop debugging and return to normal execution
   - Use `traceback()` to print the sequence of calls that lead to the last error

Programming with R I

R provides commands to control the flow of a function:

1. **Conditional execution**:

   ```
   > if (expr_1) expr_2 else expr_3
   ```

   - *expr_1* must evaluate to a single logical value (boolean)
     Examples: `x<0 && y>0` means *x negative AND y positive*
     `x<0 || y>0` means *x negative OR y positive*
   - *expr_2* and *expr_3* can be composed of several expressions between brackets and semi-coma: `{...; ...; ...}`

   See also the function `ifelse`

Programming with R II

2 Repetitive execution

- *for* loops: `> for (name in expr_1) expr_2`
  - `name` is the loop variable (change at each loop cycle),
  - `expr_1` is a vector expression (a sequence like `1:10`),
  - `expr_2` is a group expression which is evaluated as `name` varies in the range specified by `expr_1`
- *repeat* and *while* loops
  - `repeat expr`
  - `while(condition) expr`

End loops with the statement `break` (the only way to stop a *repeat* loop)

Skipped a cycle with the statement `next`: (causes the loop to skip to the next cycle)

Most of the time, loops can be avoided with R which provides function to do effective repetitive execution (`apply, lapply,...`)

Packages I

- Packages are collections of functions, datasets and documentation for particular tasks.
- R comes with some default packages: *base, stats,...*
- Various other packages are available on the CRAN website (`http://cran.r-project.org/`):
  - **ade4**: analysis of environmental (multivariate) data,
  - **ape**: Analysis of Phylogenetics and Evolution,
  - **sem**: Structural Equation Models,
  - **smatr**: (Standardised) Major Axis estimation and Testing Routines,
  - **spatstat**: spatial point pattern analysis, model-fitting, simulation and tests,
  - **vegan**: Community Ecology Package,
  - ...
- Packages can be installed directly from R from the Menu `Packages` (if a Internet connection is available...)

Packages II

- Once installed, packages must be loaded in R with the function *library*:
  ```
  > library(package_name)
  ```
- Once loaded, packages are attached and can be seen with the *search* command:
  ```
  > search()
  ```
- The list of all objects in a package can be printed by:
  ```
  > ls(package:package_name)
  ```
  or `ls(pos = position in the "search" list)`

Graphical procedures I

R offers possibilities that allow users to:

- produce numerous types of graphs for many types of data
- create almost any new type by customizing symbols, text, legends, colors,. . .
- interact with graphics (to identify points for instance)

Advantage and drawback: **many** potential graphical parameters can be modified.

Graphical procedures II

Two types of graphical functions:

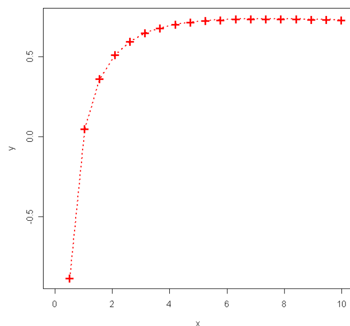
1. **High-level** functions, used to create new graphs of pre-defined types: scatter plots ($y = f(x)$), histograms, barplots, 3D-plots,...
2. **Low-level** functions, used to add features or information on an already drawn graph: points, lines, legends, text,...

High-level functions I

- Generic scatter-plotting

  `plot()`: the resulting plots depends on the class of the object

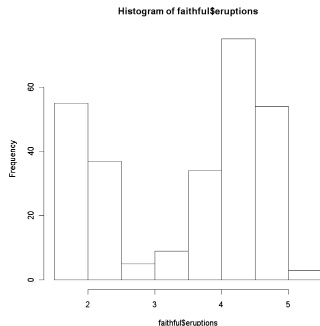
```
> x=seq(0, 100, by = 1); y = sqrt(x)*log(x)
> plot(x,y,type="o",pch="+",lty=2,lwd=2,col="red")
```



High-level functions II

- Histograms

```
hist(x,...)
> library(datasets) # load the datasets package
containing numerous datasets
> hist(faithful$eruptions)
```

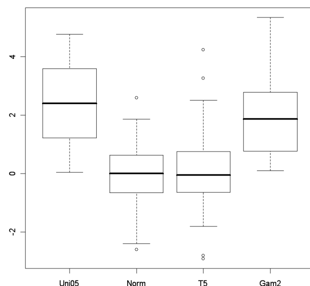


High-level functions III

- Boxplots

  `boxplot(`x`,...)`

```
>  mat <- cbind(Uni05 = (1:100)/21, Norm = rnorm(100),
T5 = rt(100, df = 5), Gam2 = rgamma(100, shape = 2))
> boxplot(data.frame(mat))
```

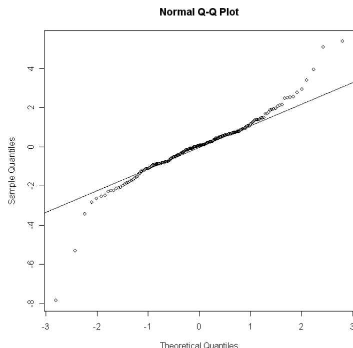


High-level functions IV

- Quantile-Quantile plots
  The quantile-quantile (q-q) plot is a graphical technique for determining if two data sets come from populations with a common distribution.
  A q-q plot is a plot of the quantiles of the first data set against the quantiles of the second data set.

High-level functions V

```
> y = rt(200, df = 3) # draws 200 random numbers from a Student t-distribution
> qqnorm(y) # plots a normal QQ plot of the values in y
> qqline(y) # adds a line which passes through the first and third quartiles
```

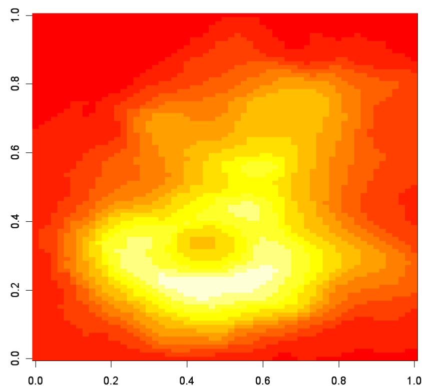


See also the function `qqplot`

High-level functions VI

- Images

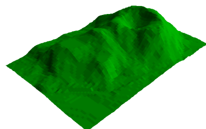
```
> image(t(volcano)[ncol(volcano):1,])
```



High-level functions VII

- 3D plots

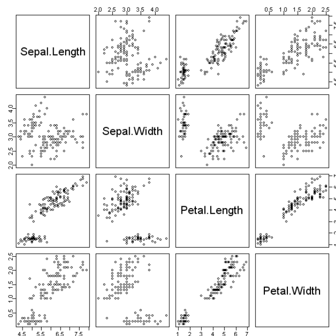
```
> z=2*volcano
> x=10*(1:nrow(z)); y=10*(1:ncol(z))
> persp(x, y, z, theta = 135, phi = 30, col = "green3",
scale = FALSE, ltheta = -120, shade = 0.75, border = NA,
box = FALSE)
```



High-level functions VIII

- Multivariate data plotting (see also package *ade4*)
  - `pairs(X)`: pairwise scatterplots of variables in matrix X

```
> pairs(iris[1:4])
```

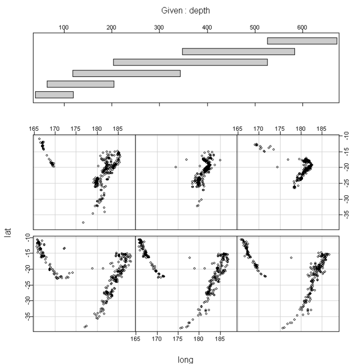


High-level functions IX

If we have few variables:

- `coplot(`$a \sim b|c$`)`: plot a (numeric) as a function b (numeric) for every level of c (factor) or for values of c (numeric) in intervals (given or computed automatically)

```
> coplot(lat ~ long | depth, data = quakes)
```

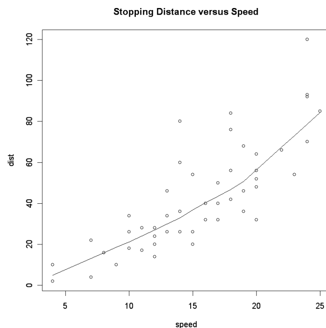


Low-level functions I

Low-level functions are used to add features on a graph

- Add points: `points(x, y)`
- Add lines: lines(x, y)

```
> plot(cars, main="Stopping Distance versus Speed")
> lines(lowess(cars))
```



Low-level functions II

You can also add:

- titles: `title(main,sub)`
- text: `text(x,y,labels)`
- axes: `axis(side,...)`
- straight lines: `abline(a=...,b=...,[h=...,v=...])`
- legends: `legend(x,y,legend,...)`

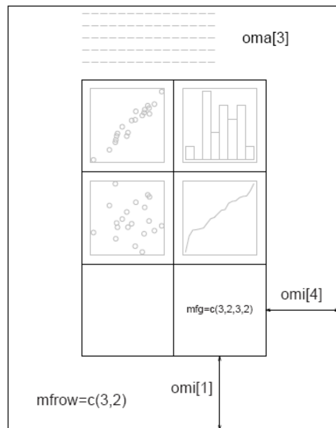
Graphical parameters I

A single plot is called a **figure** which comprises a *plot region* surrounded by margins where text can be written (axis labels, titles,. . .) and bounded by axes.

mar[3]
Plot region
mai[2]
mai[1]
Margin

Graphical parameters II

Graphs can be arranged in a $n \times m$ array of figures on single page. Each figure has its margin, and the array is surrounded by an outer margin



Graphical parameters III

Graphical parameters can be controlled in two different ways:

- High- and low-level plotting functions allows to control some parameters (see the `plot` example),
- The function `par()` can be used to set or query many graphical parameters (71): margin width, line type, thickness, figures layout,...

```
> old = par() # save the parameters
> par(mfrow = c(3,2),cex = 1.2, mgp = c(2,0.5,0),
+ xlog=T,...)
> ...# do some plotting
> par(op) # reset former parameters
```

Graphical parameters IV

Some useful parameters:

- mfcol: create an array for multiple figure plotting
- mar: margin width (4 values)
- lty: line type
- lwd: line width
- cex: character expansion (magnifying value) (see also *cex.lab, cex.main,...*)
- pch: plotting character,
- mfg: position of axis title, axis labels, axis line in margins
- col: plotting color
- las: style for axis labels (horizontal, vertical,...)
- ...

See the function layout for more complex layout than with mfcol

Interacting with graphs

Two main functions:

- `locator()`: allows to point with the mouse and prints the coordinates in the console,
- `identify()`: cliking near a point returns the index number of the point in the data

Graphics devices

One can easily produce graphics files (*.ps*, *.eps*, *.pdf*, *.png*, *.jpg*,...), to include in an report for instance

- `dev.print(file)`: copies the figure in the active graph window in a postscript *file*,
- `dev.copy2eps`: same as *dev.print* but produce *.eps* files
- `pdf`: same as above for PDF files

Or use the *File* menu in the graphic window for saving in other formats

Use the function `windows()` to create a new graphic window

References

This presentation is largely inspired by the manual:

An Introduction to R.
Notes on R: A Programming Environment for Data Analysis and Graphics
Version 2.4.0, 2006-10-03
by W. N. Venables, D. M. Smith and the R Development Core Team

Most of the graphical examples come from the help documentation on functions