

## Starting an R session

R is started by clicking on the relevant icon (M\$Win) or writing “R” on your console (linux). The first procedure starts R on a default directory the later starts R on the directory where you are.

One of the most common mistakes using R in M\$Win is to start R using the icon and be lost on one’s directory tree. In particular trying to load data or source files that are not on the working directory. To learn more about R read the [manual](#). To check where you are type `getwd()`. Now define your working dir using `setwd("c:\\mydir")`. It’s a good strategy to use a low level working dir to avoid having to type long lines of code.

## Using the installed packages

Installing a package in R merely makes it available on the local harddisk. If you want to use a package in a R-session, you need to load the package in memory by using the `library` command:

```
library("FLCore")
library("FLXSA")
library("FLSTF")
```

**Note:** you can only install packages that are not already loaded into memory. If you are working in a R-session and a new version of e.g. FLCore comes up that you want to install, you first have to quit the R-session and start a new one. In that new R-session you can install the new FLCore package.

To get an overview of the active libraries, type:

```
library()
```

## Sourcing FLR code

When code is under development, there may not be ready-made packages available. Nevertheless, the basic functionality of a package is usually contained in the R-source files. These can be loaded directly into the R-session, using the `source` command.

```
source("mycode.r")
```

## R tips and tricks

General introductions to R can be found here:

- Emmanuel Paradis: [R for beginners](#) (pdf)
- Thomas Lumley: [two day course in R](#) (html/pdf)
- Quick reference card: <http://www.rpad.org/Rpad/R-refcard.pdf>

The following general introduction to R was developed by Philip Grosjean and could be used as a starting point for FLR users/developers.

```
# set and retrieve working dir
# setwd() sets the working dir where the workspace and Rhistory are stored
# note that pathname uses forward slashes or double backward slashes, rather
# than
# single backward slashes, this is because R is based
# on ansi C that uses \ for output control of textstrings
setwd("c:\\mydir")
```

```

# getwd outputs the pathname of working directory
getwd()

# R is strongly vector orientated, following command creates vector a of length
5
b<- 1:5

# length() outputs length of vector b
length(b)

# typing only the name of the vector outputs its values
b

# Calculations are generally performed on all elements in vector thus: power of
two in all elements in vector b
b^2

# example get syntax of function call (use arg), in this case function log
# if one of the arguments is not followed by "=", then no default, in log case x
is obligatory,
# while base has default exp(1) , see output of args(log)
args(log)

# do natural log of all elements in b vector
log(b)

#other possibilities using log
log(x=b)
log(x=b, 10)
log(base=10,b)

# arguments for more complex function
args(plot.default)

# example for plot, lwd is the thickness of line used for plotting SYMBOL
representing
# observation
plot(b, lwd=2)

# example code over more lines.
log(
    10
)

# vector can also contain elements of type string: input strings in vector b
b<- c("a","b","c")
b

# for naming variables one is allowed to use digits, letters and dot, variable
has to start with letter
# variables are case sensitive (as c++, unix)

# assign values to variables is preferably done by "<-" sign. This is also
allowed within function
# thus in example, a is argument to x, with value 1
log(a<-1)

# use help and example for functions
help(log)
example(log)
?cbind

# open html help from code
help.start()

```

```

# help on non linear fitting using least squares
?nlm

# return all functions, of which name contains a string
# e.g. return all functions that have "log" in name
apropos(log)

# search all workspace and packages, note that nls is already loaded
# just as several other packages
search()

# example for function that is in package that is not loaded by default
example(boot)

#load bootstraplibrary using library command, do example and
# subsequently remove library
library(boot)
example(boot)
detach("package:boot")

#see sourcecode for function, eg boot
library(boot)
fix(boot)
detach("package:boot")

# possibilities for values
# Inf or -Inf for infinity
# NA missing value
# NaN not a number for instance for 0/0

a<- c(3,4, NA, Inf)
a

tt <- c(1, NA, 3, 4, NA, 6, NaN)
is.na(tt)

# NaN is distinct from NA
is.nan(tt)

# sum of vector a will in this case be defined as NA, since NA propagates
args(sum)

# note that na.rm will remove these and then sum will summarise all available
values in vector
sum(a)
sum(a, na.rm=T)

#string manipulation, don't yet know what is done
b<- c("yes","no", "maybe")
paste("is it",b)

paste("is it",b, sep=",")

paste("is it",b, sep=" ", collapse=" ")

# typeof() shows type of elements in vector
c <- c(TRUE, TRUE, FALSE)
typeof(c)

# true and false can be assigned by T and F
d<- c(T,T,F)

```

```

# is equal to sign is == just like c++
# is not c , you get vector with inverse of logical
!c

#subsetting vector using vector with logical values
a <- c(T, T, F, F, T)
b <- 1:5
# now the subset command, which is in vertical brackets, thus ,1 2 and 5
# are kept
b[a]

#subset such that you get all values larger than two
b[b>2]

# subset the vector by the index, so now 1 to 3
b[1:3]

#or subset vector by index, given by a list of numbers
b[c(1,2,1,3,4,1,1,5)]

#or give part of vector except for a number (in this case 3)
b[-3]

# convert vector of integer to vector of logical (just for the example)
# I concatenated 0 to beginning of vector
b <- c(0,b)
lb <- as.logical(b)

#recycling rule when doing vector arithmetics, after third element in b, jump
back to first element in b
a <- 1:6
b <- 1:3
a + b

# Adding 2 to a numeric vector using the conventional programming approach - The
slow method
x <- c(2,3,4,5)
for (i in 1:length(x)){
  x[i] <- x[i] +2
}

# Using vectorised arithmetic one can simple type - The faster method
x <- c(2,3,4,5)
x + 2

# However, beware of automatic recursion.
a + c(2,6,6,3)

#define two-dimensional array (matrix)
a <- matrix(rnorm(16),nrow=4)
a

#subset value from matrix, or vector defined by row or column
a[2,3]
a[2,]
#number of columns or rows
a[2:3,]

# there is argument in matrix function on how the matrix is filled
# by row or by column, differences and args
a <- matrix(1:16,nrow=4)
a <- matrix(1:16,nrow=4, byrow=TRUE)

```

```

#what kind of datastructure is "a" (output tells number of rows and columns)
attributes(a)

# we can remove dimensions, getting vector rather than matrix
attr(a,"dim") <- NULL
attributes(a)
a

#reshape vector in three dimensional array, note that you cannot get out of
vector bounds, this is protected
attr(a,"dim") <- c(2,4,2)
a

#create list of fish (list has more than one variable with different type)
a<- list(fish="toto",age=3, good.shape=FALSE)
a

# access fish part of "a" list, different methods (JJP: are there differences?)
a$fish
a[["fish"]]
a[[1]]

# vectors in list do need need to be of same type
a<- list(fish="toto",age=1:3, good.shape=FALSE)
a

#dataframe, all vectors in dataframe need to be of same length, this is more or
less the sas dataset or excel sheet
#first create list with equal vector length, then make dataframe
a<- list(fish=c("toto", "toto2","toto3") ,age=1:3, good.shape= c(F,T,T))
a
b <- as.data.frame(a)
b

# one is also able to make dataframe at once. NOTE: the c in name=c("sole")
allows one to give list of strings
f <- data.frame( age=1:5, name=c("sole","plaice","cod","turbot","brill"))

#substract from dataframe (you get vector), different examples:
b$age
b[,2]
b[2:3,]

# get class
class(b)

#give comment to list
attr(b,"rem") <- "This is my dataframe"

#what does unclass do?
unclass(b)

# show and edit dataframe b as spreadsheet
fix(b)

#export dataframe, by default if no path, then file is put in setwd() dir
write.table(b, file="c:\\mydir\\b.txt")
args(write.table)

#import data.frame from textfile on disk
d<- read.table(file="c:\\mydir\\b.txt")
args(read.table)

#many other forms for reading data (e.g. .csv or delimited) see:

```

```

?read.table

# summary statistics
a<- rnorm(100)
mean(a)
sd(a)
plot(a)
hist(a)
boxplot(a)
boxplot(a, col="red")
qqnorm(a)
qqline(a)

# median and first and quarter of data
fivenum(a)

# stem and leave plot
stem(a)

# example of possibility
data(faithful)
attach(faithful)
hist(eruptions, seq(1.6, 5.2, 0.2), prob=T, col=7)
lines(density(eruptions,bw=0.1),lwd=2,col=4)
rug(eruptions)

#another dataset, plot is here adapted for the kind of dataset, b setting method
# in this case, nottem is a timeseries dataset. This is very similar to object
# oriented programming in C++, (this is probably class object with both data and
# functions
data(nottem)
plot(nottem)

#however, by defining plot.default, R can be forced to use default plot method
plot.default(nottem)

# using source you can "batch" a script running it completely, NOTE this is
fake reference
# to script file
source("C:\\mydir\\myscript.R")

#using seq() or by using (1:6)/6 you can make stepsize smaller
?seq
a <- seq(0,10, by=0.5)
b <- rnorm(length(a))
a
b

# define new function
# A function comprises 2 main components: the function arguments and a function
body.
# The function body contains error trapping, the main processes of the function
and the return statement.

cube <- function(x, na.rm=TRUE)
{
  if (na.rm == TRUE) x <- x[!is.na(a)]
  return(x^3)
}
a <- 1:5
cube(a)

```

## Using Apply and Sweep

```
# Loops can be replaced by a few calls to atomic computations.
# This not only speeds up your code but helps conceptually by moving towards a
"whole object" view.

kk <- array(rnorm(120), dim=c(8,5,3))
apply(kk, 2, mean)

apply(kk, c(2,3), mean)

# A family of apply functions exist to perform actions on different types of R
classes.
# See lapply for lists, tapply for dataframes and other incarnations sapply
rapply mapply etc.

# Sweep can be used very effectively with apply
# This example calculates the mean across the first dimension and subtracts it
from each
# element of the array

kk <- array(runif(12), dim=c(3,4))
sweep(kk, 1, apply(kk, 1, mean), "-")
```

## Plotting with lattice

Due to the multidimensionality of the fisheries data, reflected on the multidimensionality of FLQuant, FLR plots are mostly based on the lattice plots. This plots are *conditioning plots* which use the S formula syntax to define conditioning variables. For example if a dataframe has 3 variables, a,b and c, one can use the formula  $a \sim b \mid c$  to plot a against b conditioned on c. The result will be matrix of plots  $a \sim b$  for each value of c. If the conditioning variable is not set, the plot matrix will have a single plot. Lattice is a very powerful toolset but complex, see the lattice manual for more info.

```
# load library and data
require(lattice)
data(iris)
# see the first rows of data
head(iris)
# now plot
xyplot(Sepal.Length~Sepal.Width, data=iris)
# now conditioning on Species
xyplot(Sepal.Length~Sepal.Width|Species, data=iris)
# it is possible to use more than one conditioning variable (note the recycling
of v)
v <- rep(letters[1:2],rep(25,2))
xyplot(Sepal.Length~Sepal.Width|Species*v, data=iris)
# play with the several plots provided by lattice
dotplot(Sepal.Length~Sepal.Width|Species*v, data=iris)
barchart(Sepal.Length~Sepal.Width|Species*v, data=iris)
bwplot(Sepal.Length~Species|v, data=iris)
```