

Programming with R

Bjørn-Helge Mevik

Research Infrastructure Services Group, USIT, UiO

RIS Course Week spring 2014

Introduction

Basic building blocks

Programming in R

Best practices

Moving on...

R prerequisites

- ▶ Basic calculation and data types
- ▶ Saving and loading data
- ▶ Using functions and running scripts

Overview of R

- ▶ A dialect of the language 'S'
- ▶ Syntax is C-like, but philosophy is functional
- ▶ Focus on matrices and vectors
- ▶ Free, open-source (GPL)
- ▶ Active user community with thousands of contributed packages
- ▶ Latest version: 3.0.3
- ▶ URL: <http://www.r-project.org/>

R features

- ▶ Scriptable and extensible
- ▶ Bindings to many other systems/languages, e.g., Python, Perl, Matlab, *SQL, Excel
- ▶ Dynamically typed
- ▶ Functional language, borrows ideas from lisp, but C-like syntax
- ▶ Supports object oriented programming (two types!)
- ▶ Designed to ease conversion from interactive usage to programming

Help!

This is probably the most important slide!

- ▶ `?mean` - help for a function
- ▶ `help.search("regression")` or simply `??regression` - search in your installed R
- ▶ `RSiteSearch("logistic")` - search the R web site
- ▶ `demo()` - list/run demos
- ▶ `vignette()` - list package vignettes
- ▶ `help.start()` - start help centre

Common data types

- ▶ Atomic types: *number, string, logical*

	type	1-dim	2-dim	> 2 dim
▶ Compound types:	same	vector	matrix	array
	mixed	list	data frame	

- ▶ Factors ('character vector with fixed set of values')

All types have a class: `class()`

Factors

- Factors are stored as a numeric vector, with special attributes for the levels:

```
x <- factor(rep(c("white", "black"), 20))  
x  
print.default(x)  
attributes(x)  
str(x)
```

- Special case: ordered factor; handled differently in models:
`ordered(rep(c("white", "black"), 20))`

Special values

- ▶ R has a few special values:
 - `NA` Missing value, `is.na()`
 - `NaN` Not a Number ('0/0'), `is.na()` and `is.nan()`
 - `-Inf`, `Inf` Infinite number ('1/0'), `is.finite()`
- ▶ Many functions have an argument `na.rm` to ignore `NAs` and `NaNs`:
`mean(x, na.rm = TRUE)`

Names and indexing

All compound types can have names:

- ▶ `x <- c(a = 0, b = pi, c = exp(1))`
- ▶ `y <- list(house = "yellow", car = "blue")`
- ▶ `z <- matrix(1:4, ncol = 2, dimnames = list(c("a", "b"), c("first", "second")))`

They can be used in indexing:

- ▶ `x["a"]`
- ▶ `x[c("a", "b")]`
- ▶ `y$house`
- ▶ `z[, "second"]`

Common functions

Matrix functions:

`A %*% B`

Matrix product (note: `A * B` is *element wise* product)

`t(A)`

Transpose of matrix

`crossprod(A, B),`

Fast versions of `t(A) %*% B` and

`tcrossprod(A, B)`

`A %*% t(B)`, resp.

`colSums(A), rowSums(A),`

Fast calculation of column/row sum/mean

`colMeans(A), rowMeans(A)`

`apply(z, 2, mean)`

Apply a function (here: `mean`) along a dimension of a matrix or array

`cbind(A, B)`

Join matrices by column

`rbind(A, B)`

Join matrices by row

Common utility functions: `length()`, `dim()`, `numeric()` (create numeric vector), `sort()`, `rev()` (reverse vector), `rep()` (repeat elements)

Control structures 1: if

R has several types of control structures: *if statements*, *loops*, *switch statements*.

If statements:

```
if (a > 1) { print("hello") }
```

```
if (length(x) > 5) {  
  print("long")  
} else {  
  print("short")  
}
```

Control structures 2: switch

Switch/case statements:

```
switch(type,  
      sqrt = sqrt(x),  
      log = log(x),  
      square = x^2,  
      twice =,  
      double = 2*x,  
      "Error: unknown type"  
)
```

Control structures 3: loops

For loops:

```
for (i in 1:10) { print(i) }  
for (i in list("a", "b", TRUE)) { print(i) }
```

While loops:

```
num <- 1  
while (num < 10) { print(num); num <- num * 2 }
```

Repeat loops:

```
num <- 0  
repeat {  
  num <- num + 1  
  if (num %% 2 == 0) { next } # Why not "if (num %% 2)"?  
  print(num)  
  if (num > 10) { break }  
}
```

Note: *do* remember [break](#). :)

Logical expressions

The tests in `if` and `while` statements are *logical expressions*. The logical operators are:

<code>==</code>	equality
<code><</code> , <code><=</code> , <code>></code> , <code>>=</code> , <code>!=</code>	inequality
<code> </code>	or
<code>&&</code>	and
<code>!</code>	not

Note that 0 evaluates to `FALSE` and any non-zero numerical value to `TRUE`.

Functions

- ▶ Functional language => 'everything' is a function
- ▶ All functions return a value (`NULL`, if nothing else)
- ▶ Arguments can be specified by name
- ▶ Arguments can be skipped
- ▶ Arguments can have default values
- ▶ See argument list: `args(rnorm)`
- ▶ See function definition: Just type its name, e.g. `ls`

Example:

```
> args(rnorm)
> rnorm(10, sd = 2) # versus
> rnorm(10, 0, 2)
```

Function declaration

```
diff1 <- function(x) {  
  y <- numeric(length(x) - 1)  
  for (i in 1:length(y)) {  
    y[i] <- x[i+1] - x[i]  
  }  
  return(y) # or simply y  
}
```

```
orig <- 10:1  
diff1(orig)
```

Function arguments

Arguments can have default values and fixed choices, and there can be a variable number of arguments:

```
argtest <- function(arg1, arg2 = "default",  
                    arg3 = c("choice1", "choice2"), ...) {  
  if(missing(arg1)) { cat("arg1 is missing\n") }  
  if(missing(arg2)) { cat("arg2 is missing\n") }  
  cat("arg2 has value", arg2, "\n")  
  if(missing(arg3)) { cat("arg3 is missing\n") }  
  arg3 <- match.arg(arg3)  
  cat("arg3 has value", arg3, "\n")  
  cat("The optional arguments are\n")  
  list(...)  
}
```

Scope of variables

Functions see variables in the environment where the function was *declared*, but modifications are local:

```
x <- 1
fun <- function() { print(x); x <- x + 1; print(x) }
fun()
x
```

Note: Braces (`{ }`) themselves do *not* create a local environment, so i.e., assignments in `if` statements are global:

```
rm(y)
if (TRUE) { y <- 2 }
y
```

Extended example

Look at file *mypls.fit.R*...

```
source("mypls.fit.R")  
data(gasoline, package = "pls") # Import data set  
result <- mypls.fit(gasoline$NIR, gasoline$octane, ncomp = 5)  
str(result)
```

Vectorisation

R is most efficient with vectors and matrices

```
diff1 <- function(x) { # Warning: suboptimal code
  y <- numeric(length(x) - 1)
  for (i in 1:length(y)) {
    y[i] <- x[i+1] - x[i]
  }
  return(y) # or simply y
}

diff2 <- function(x) {
  x[2:length(x)] - x[1:(length(x)-1)]
}

diff1(1:10)
diff2(1:10)
system.time(x <- diff1(1:100000))
system.time(x <- diff2(1:100000))
```

Preallocation

If you know the size of a vector, matrix or array, preallocate it. Let's see what happens if you don't:

```
diff0 <- function(x) { # Warning: really bad code
  y <- 0
  for (i in 1:(length(x) - 1)) {
    y[i] <- x[i+1] - x[i]
  }
  return(y) # or simply y
}
diff0(1:10)
system.time(x <- diff0(1:50000))
```

Avoiding pitfalls

- ▶ Use `TRUE` and `FALSE` instead of `T` or `F`.
- ▶ Use `X[,1:ncomp, drop=FALSE]` if `ncomp` can be 1.
- ▶ Use `seq_along(v)` instead of `1:length(v)` if `v` can be empty.
- ▶ Name arguments in code, e.g., `lm(y ~ x, data = mydata)` instead of `lm(y ~ x, mydata)`.
- ▶ Use `diag(v, ncol = length(v))` if `v` can have length 1.
- ▶ Use `isTRUE(x)` instead of `x == TRUE`, especially if `x` is a function argument
- ▶ Note that `|` and `&` are *not* the same as `||` and `&&`

Optimisation

Optimisation rules:

- ▶ 0. rule: don't do it (yet)!
- ▶ 1. rule: make sure the program is correct first!
- ▶ 2. rule: simplify/optimize/choose the right algorithm first
- ▶ 3. rule: follow general best practices (vectorisation, pre-allocation, pre-calculate stuff; move tests, formula handling etc. outside computing function)
- ▶ 4. rule: use profiling and memory-profiling to find the hot spots
- ▶ 5. rule: try jit-compiling of innermost loops
- ▶ 6. rule: try compiling R with a faster BLAS/LAPACK library
- ▶ 7. rule: try re-writing the hot spots (create less general code)
- ▶ 8. rule: implement hottest spots in C/Fortran

Other topics

There are several things we haven't touched in this lecture:

- ▶ Parallel programming in R.
- ▶ Interfaces to other languages.
- ▶ Creating R packages.
- ▶ Objects, classes, generic methods.
- ▶ Formal object-oriented programming

See also...

- ▶ The help pages
- ▶ The manuals (`help.start()`) - especially *The R language definition*, *Writing R Extensions* and *An Introduction to R*
- ▶ The book *Parallel R*, McCallum & Weston, O'Reilly
- ▶ There are many R books covering fields as statistics, bioinformatics, linguistics, graphics/plotting, programming, etc.
- ▶ `www.r-project.org`
- ▶ `www.bioconductor.org`

Help!

This is probably the most important slide!

- ▶ `?mean` - help for a function
- ▶ `help.search("regression")` or simply `??regression` - search in your installed R
- ▶ `RSiteSearch("logistic")` - search the R web site
- ▶ `demo()` - list/run demos
- ▶ `vignette()` - list package vignettes
- ▶ `help.start()` - start help centre