Statistical Computing
Master Data Science
Winter Semester 2019/20

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Prof. Tim Downie

Workshop 7 — regression 2, control structures in R

In the Workshop

There are two subjects in today's workshop. The first is looking at quadratic regression. The second is a first look at programming using R. The programming abilities in R are very useful and is an advantage over most other statistical software.

Preliminaries

- Download the data file `Grashoppers50.Rda` from Moodle page into your `H:\\StatComp` directory.
- Start RStudio
- Strg+Shift+N opens a new R script.
- Type in the comments
  ```
  #Statistical Computing: Workshop 7
  #Regression and control structures
  ```
- Save the file in your `H:\\StatComp` folder with the name `Workshop6.R`.
- Set your *working directory* to be `H:\\StatComp`. The code to do this is
  ```
  > setwd("H://StatComp")
  ```
- Clear your workspace using of objects from a previous session
  Session > *clear workspace*.
- Open a Word document or similar to answer the exercises in this workshop.

1 Regression

Exercise 1 Quadratic regression

The grasshoppers example for linear regression in Lecture 6 contained 15 observations. The original data has 50 observations, the version last week was shortened for teaching purposes. You will use the full data to investigate whether a linear or a quadratic regression fits the data best.

Load in the data

```
> load("grasshoppers50.Rda")
```

Fit the linear model to the full data:

```
> lm.obj1<-lm(chirp~temp,data=grasshoppers50)
> summary(lm.obj1)
```

Check that the estimates are similar to those obtained in the lecture.

Obtain a quadratic regression and look at the summary output:

```
> lm.obj2<-lm(???~???+I(???^2),data=grasshoppers50)
> summary(???)
```

Look at the p-value for the quadratic term. Is this "significant at the 5% level?" Ie. is the p-value less than or equal to 0.05?

Notice that the linear term for the quadratic model is greater than 0.05. This is irrelevant. If the quadratic term is significant you **keep both** the linear and quadratic term in the model! If the quadratic term is not significant you look at the linear model to see if the linear term is significant.

Obtain a residual plot for the two models.

```
> par(mfrow=c(1,2))
> plot(lm.obj1,which=1)
> plot(lm.obj2,which=1)
```

Lets plot the data and add both regression functions. The linear regression line is easy, just use the function `abline()`. For the quadratic regression we need to use `curve()` which takes a user defined function[1] as the first argument.

```
> par(mfrow=c(1,1))
> plot(grasshoppers50$temp,grasshoppers50$chirp,pch=16)
> abline(lm.obj1,col=3)
> myfun<-function(x) 15.75556-???*x+???*x^2
> curve(myfun,from=6,to=20,add=TRUE,col=4)
```

[1] You will learn how to write functions in R next week

Multiple regression

So that you have enough time on control structures this week, you will be set an exercise on multiple regression next week.

2 Control Structures

Control structures are the commands which make decisions or execute loops. Such structures are fundamental building blocks when writing R programs. We will look at

- Conditional execution of statements: `if` and
- Loops:`for` and `while`.

Many of the examples below are given as teaching examples. Usually there are much shorter and elegant methods of achieving the same result.

Conditional Execution of Statements

Conditional execution can be done using the `if` statement. If statements are usually used inside a loop or a function (see late sections). They allow commands to be optionally executed, depending on whether a condition is satisfied. The general structure is

```
if(logical.condition) statement1 else statement2
```

- First, R evaluates `logical.condition`, a condition that evaluates to `TRUE` or `FALSE`. It must be a single value, not a vector of logical values.
- `statement1` is either one command, or a group of commands that are delimited by `{ }`. It is only executed if `logical.condition` is `TRUE`.
- The `else` statement is optional, but when used `statement2` is evaluated when `logical.condition` is `FALSE`.

The `if` statement can be extended over several lines, and `statement1` (or `statement2` or both) may be a compound of simple statements, separated by semi-colons and enclosed within braces `{ }` .

Examples

```
> x<-3
> if(x>0)
+ sqrt(x)
[1] 1.732051
> x<- -4
> if(x>0)
+  sqrt(x)
>
> x <- 2
> y <- 5
> if (x >= y)
+ { absdiff <- x - y ;
+ cat("\n","Absolute difference is ", absdiff,"\n")} else
+ { absdiff <- y - x ;
+ cat("\n","The absolute difference is ",absdiff,"\n")}

The absolute difference is  3
>
```

Points to note:

- The R command `cat()` prints out its arguments.
- The term `"\n"` causes R to insert a carriage return (i.e. to continue output at the beginning of a new line).
- The continuation prompt + appears when in the middle of the `if` statement. If you type the commands in directly into the console the statement is not executed until the command is complete. Note that `if(logical.condition) statement1` is a complete command so if you want an `else statement 2` the linebreak should not come immediately before `else`, as in the above example.

`if` statements: details

- Usually the `logical.condition` contains an equality or inequality: ==, <, >, <=, or >=. Note the logical equality symbol uses a "double equals" symbol, ==. This is to differentiate it from object assignment and from argument names in functions. There is also a "not equals" symbol `!=`. For any other *negation* use `!logical.condition` or `!(logical.condition)`. In particular, there is no direct symbol for *not greater than*; `if(x!>y)` will cause an error, use `if(!(x>y))` or better `if(x<=y)` instead.

- The `if` statement checks one logical condition. If you have more than one condition to test, then you need to consider which of the following you really want.
  - All of the multiple conditions are `TRUE`
    E.g. `if(all(y==z))` will execute the following statement only if all elements of `y` are equal to the corresponding element of `z` (elementwise comparison).
  - At least one of the multiple conditions is `TRUE`
    E.g. `if(any(y==z))` will execute the following statement when at least one element of `y` is equal to the corresponding element of `z` (elementwise comparison).
  - You have a vector and want to test a condition for each element: use the function `ifelse`

```
> x<-1:10
> ch<-ifelse(x>=7,"big","small")
> print(ch)
[1]"small" "small" "small" "small" "small" "small" "big" "big" "big" "big"
```

- If the `logical.condition` is a numerical value then it will be *coerced* to a logical value:
  0 evaluates to `FALSE` and any other number evaluates to `TRUE`

```
> x<-3
> if(x) cat("x is non zero\n")
x is non zero
```

- You can use `T` and `F` as the logical values `TRUE` and `FALSE`, but you should be aware of a potential problem. You are not allowed to define objects with the name `TRUE` or `FALSE`, in the same way that you cannot define an object called `2`. You *are* however allowed to call an object `T` or `F`. Doing this can lead to bugs that are very difficult to find. Here is an example.
  Example

```
> x<-15
> if(x>10) y<-TRUE else y<-FALSE
> if(y==T) cat(x,"is greater than 10\n") else
+ cat(x,"is not greater than 10\n")
15 is not greater than 10
> T<-"tea"
> x<-25
> if(y==T) cat(x,"is greater than 10\n") else
+ cat(x,"is not greater than 10\n")
25 is not greater than 10
```

  Although this example looks artificial, students do sometimes define an object as `T` or `F` leading to the above problem at some point in the future. Now please remove the object `T`!

```
> rm("T")
```

Another `if` example with comments below

```
> x <- c(1,3,-2)
> if (is.numeric(x) && min(x) > 0 )
+ sx <- sqrt(x) else
+ stop("x must be numeric and positive")
Error: x must be numeric and positive
>
```

- The logical operator `&&` is simply the standard "AND" for logical expressions. Similarly `||` is the standard logical "OR". Use `eor(y,x)` if you want to use the "exclusive or" "EOR" expression.
- The command `stop()` ends the execution of whatever R is doing, reports an error, and prints message supplied as an argument.
- The function `is.numeric()` returns `TRUE` only if all elements of its argument are numeric (as opposed to logical, character etc.).

Loops: `for` and `while`

A loop repeats a statement (or group of statements) a number of times. Each iteration (repetition) will execute the same syntax, but will do something slightly different as one or more variables are different.

There two main types of loop. "`for` loops" and "`while` loops"

`for` loops

A `for` loop allows a statement to be iterated as a counting variable proceeds through a specified sequence. The statement has the form `for(variable in vector) statement`

Example

```
> for(i in 1:5) print(i)
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

The variable `i` takes the values 1 to 5 sequentially, and `for` each value of `i` the command `print(i)` is executed. This is different from

```
> print(1:5)
[1] 1 2 3 4 5
```

where the `print` command is only called once.

In the above example, only one command is executed within the loop, `print(i)`. If the `statement` consists of more than one command, then you must enclose the commands in curly braces `{ }`, as with the `if` statements
Example:

```
> sum1<-0
> sum2<-0
> for(i in 1:10){
+ sum1<-sum1+i
+ sum2<-sum2+i^2
+ cat("i=",i," sum =",sum1," sum of squares = ",sum2,"\n")
+ }
i= 1  sum = 1,  sum of squares =  1
i= 2  sum = 3,  sum of squares =  5
i= 3  sum = 6,  sum of squares =  14
i= 4  sum = 10  sum of squares =  30
i= 5  sum = 15  sum of squares =  55
i= 6  sum = 21  sum of squares =  91
i= 7  sum = 28  sum of squares =  140
i= 8  sum = 36  sum of squares =  204
i= 9  sum = 45  sum of squares =  285
i= 10  sum = 55  sum of squares =  385
```

Syntax of a `for` loop

- In these above examples, the *looping variable* was called `i` but it could be called anything:

  ```
  > for(this.would.be.a.tedious.variable.to.have.to.type.everytime in 1:10)
  + print(this.would.be.a.tedious.variable.to.have.to.type.everytime)
  ```

  gives exactly the same result as the first example.

- The word `in` is required verbatim.

- Aa vector follows the word `in`, this is different from most other programming languages. This vector contains the sequence of values that we want our looping variable to take. This vector can be numeric, character or logical.

  For example:

  ```
  > for(theta in c(0,pi,2*pi)) print(sin(theta))
  [1] 0
  [1] 1.224606e-16
  [1] -2.449213e-16
  > for(let in letters[c(20,9,13)]) cat(let); cat("\n")
  tim
  ```

  Note in the last example there are no braces, so only the command `cat(let)` is evaluated in each iteration. Only once the `for` loop has finished is the command `cat("\n")` evaluated. The semicolon allows two commands to be specified on one line.

`while` loops

The `while` loop is very similar to the `for` loop. Instead of a statement like `for(i in 1:10)`, there is a logical condition in the brackets: `while(x<y)`. The general form is: `while(logical.condition) statement`

Example

```
> x <- 1
> while(x<11) {
+ cat(3*x,"\n")
+ x <- x+2
+ }
3
9
15
21
27
>
```

Again the statement that is to be repeated can either be one command, or several statements bracketed by `{ }`. A `while` loop tests the logical statement *before* each iteration. It executes that iteration if the logical statement is `TRUE` and exits the loop if it is `FALSE`.

Example: what is the first factorial number that is greater than 10,000?

```
> n<-0
> prod.sofar<-1
> while(prod.sofar<10000){
+ n<-n+1
+ prod.sofar<-prod.sofar*n
+ }
> prod.sofar
[1] 40320
```

Each time the loop is iterated, n increases by one, and `prod.sofar` is multiplied by n. Therefore `prod.sofar` is equal to $n!$ (n factorial). At the end of each iteration, the condition `prod.sofar<10000` is checked. Once it reaches 40320 the `while` loop terminates.

Note: it is up to you to make sure the `while` loop is specified sensibly.
Don't type in the example below, but consider what would happen if you did.

```
> x<-2
> y<-5
> while(x<y) print(x)
```

Use of control structures

Loops in R are computationally inefficient There are many functions in R which allow you to perform the same task without needing a loop. For example: Suppose you want to calculate $\sum_{r=1}^{20} r^4$. You could use a for loop

```
> total <- 0
> for (r in 1:20) { total <- total + (r^4) }
> total
[1] 722666
```

But the function `sum` allows you to to do this directly and more efficiently.

```
> total <- sum((1:20)^4)
> total
[1] 722666
```

Question: `sum(1:20^4)` gives a different answer. Why?

Another way to avoid loops is using the apply family of functions `apply(), tapply(), lapply()` or `sapply()`.

Example: in most compiled languages to calculate the row sum of an array, you need a double for loop.

```
> M<-matrix(1:25,nrow=5,byrow=TRUE)
> rowsum<-rep(0,5)
> for(i in 1:5)
+ for(j in 1:5)
+ rowsum[i]<-rowsum[i]+M[i,j]
> rowsum
[1]  15  40  65  90 115
```

In R, you can eliminate one loop using the function `sum` on a vector.

```
> rowsum<-rep(0,5)
> for(i in 1:5) rowsum[i]<-sum(M[i,])
> rowsum
[1]  15  40  65  90 115
```

You can eliminate both loops using `apply` and `sum` together

```
> rowsum<-apply(M,1,sum)
> rowsum
[1]  15  40  65  90 115
```

The second argument `1` indicates the function `sum` is applied to each row of `M`.
`apply(M,2,sum)` applies the function `sum` to each column of `M`.

Exercise 2 in Workshop

1. Write a for loop to compute
$$\sum_{i=1}^{10} 0.5^i.$$
2. Suppose S_n is defined as
$$S_n = \sum_{i=1}^{n} 0.5^i.$$
Use a `while` loop to find S_n, where n is the smallest integer with
$$|S_n - S_{n-1}| < 10^{-6}.$$
Hints : `abs(x)` is the R function to find the absolute value $|x|$. 10^{-6} is `1e-6`

3 Tidying up

► Tidy up your script file including sensible comments.

► Save the script file (source file) again: Strg + S or *File* > *Save as*.

► Leave RStudio by typing the command:

```
> q()
```

When R asks you *Save workspace image ...?*, click on **Don't save**!

► Feierabend!