# Exercises for progamming with functions

Timothée Bonnet

March 18, 2019

## Contents

# 1 Simplest functions

### * Exercise 1 Build a first function

Write a function that return the product of three arbitrary numbers together (x*y*z) provided by the user.

**Answer of exercise 1**

```
myfunction <- function(x,y,z){
  return(x*y*z)
}
myfunction(100,2, -0.005)

## [1] -1
```

### ** Exercise 2 Multiple values in output

Write a function that return the product of three arbitrary numbers together (x*y*z) as well as their sum (x+y+z).

**Answer of exercise 2**

```
myfunction <- function(x,y,z){
  return(c(x*y*z, x+y+z))
}
myfunction(100,2, -0.005)

## [1]  -1 102
```

## ** Exercise 3 Turning a piece of code into a function

In a previous session we wrote the following piece of code to simulate p-values for a random effect in a mixed model:

```
RandomVariance <- 0
sampsize <- 500
nbblocks <- 30
pvals <- vector(length = 1000)
altpvals <- vector(length = 1000)
for (i in 1:1000)
{
x <- rnorm(sampsize,mean = 4, sd=0.25)
block <- sample(x = 1:nbblocks, size = sampsize, replace = TRUE)
blockvalues <- rnorm(n = nbblocks, mean = 0, sd = sqrt(RandomVariance))
y <- 8 - x + blockvalues[block] + rnorm(sampsize,0,1)
dat <- data.frame(response = y, predictor = x, block=block)
lm0 <- lm(response ~ 1 + predictor, data=dat)
lmm0 <- lmer(response ~ 1 + predictor + (1|block), data=dat )
(LRT0 <- anova(lmm0, lm0)) #mixed model must come first!
pvals[i] <- LRT0$`Pr(>Chisq)`[2] # the p-value
altpvals[i] <- 1-pchisq(LRT0$Chisq[2],0.5) # a better p-value
}
pvals
altpvals
```

Turn this into a function were the arguments will let you control:

- The value of the simulated variance (RandomVariance)
- The number of simulations (above it is fixed to 1000)
- (optionally) The sample size and number of random levels (blocks)

And output:

- A vector of standard p-values from the LRT
- (optionally) a vector of "alternative" p-values (from the hand-made Chi-square)
- (optionally) histograms of the distribution of p-values

**Answer of exercise 3**

```
Fpvalsimul <- function(RandomVariance = 0,
                       sampsize = 500,
                       nbblocks = 30,
                       nsimuls = 100){
  require(lme4)
pvals <- vector(length = nsimuls)
altpvals <- vector(length = nsimuls)
for (i in 1:nsimuls)
{
x <- rnorm(sampsize,mean = 4, sd=0.25)
block <- sample(x = 1:nbblocks, size = sampsize, replace = TRUE)
blockvalues <- rnorm(n = nbblocks, mean = 0, sd = sqrt(RandomVariance))
y <- 8 - x + blockvalues[block] + rnorm(sampsize,0,1)
dat <- data.frame(response = y, predictor = x, block=block)
lm0 <- lm(response ~ 1 + predictor, data=dat)
lmm0 <- lmer(response ~ 1 + predictor + (1|block), data=dat )
(LRT0 <- anova(lmm0, lm0)) #mixed model must come first!
pvals[i] <- LRT0$`Pr(>Chisq)`[2] # the p-value
altpvals[i] <- 1-pchisq(LRT0$Chisq[2],0.5) # a better p-value
}
output <- list(pvals=pvals, altpvals=altpvals)
par(mfrow=c(1,2))
hist(pvals); hist(altpvals)
return(output)
}
```

## ** Exercise 4 Recover the central-limit theorem

To simplify, the central-limit theorem states that the sum of independent random distributions tends towards a normal distribution even if the original distributions are non-normal. Let's write a function that shows that by randomly drawing distributions with random parameters, summing them, and comparing the sum to a Gaussian distribution of same mean and variance.

We can start from the following piece of code:

```
#how many numbers per distribution:
n=2000

# where we will save the sum of distributions:
output <- vector(length = n)

# three possible distributions to draw from:
rddistri <- list(rnorm,runif,rpois)

# drawing one number at random to tell which distribution we use:
choosedistri <- sample(1:length(rddistri), size = 1)
# a trouble is that the different distributions do not take the
#same number or type of arguments, so we need to have a special
#piece of code for each of them. For example, for the normal
#distribution (rnorm):
if(choosedistri==1)
{
    # Decide how you want to select mean and sd:
     output <- output + rddistri[[choosedistri]](n, mean=, sd=)
}
...
```

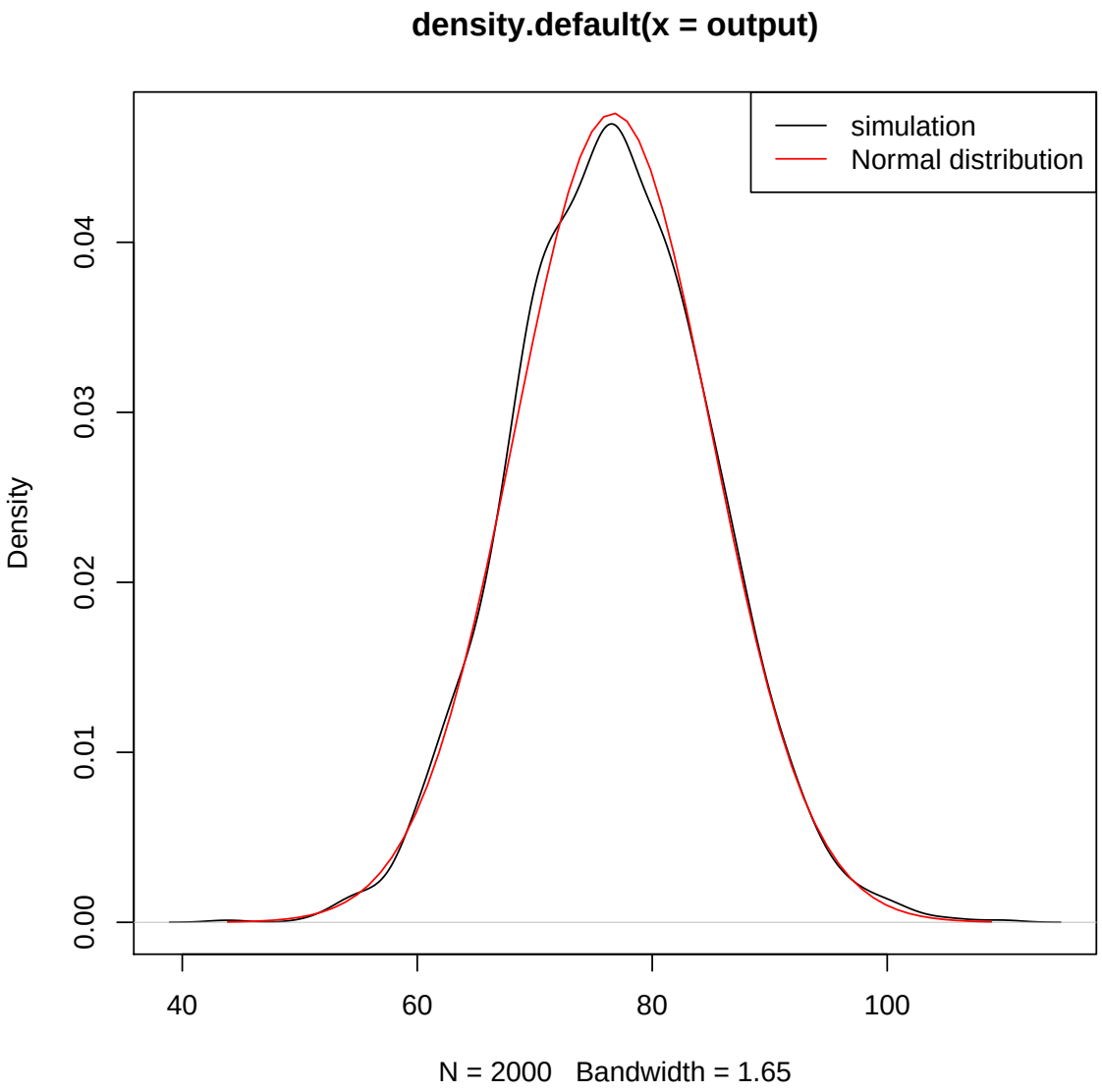
We may want to use a for loop. We could include also a plot or a statistical test (e.g., shapiro.test).

**Answer of exercise 4**

```r
simclt <- function(n=2000, nbdistri=100)
{
  output <- vector(length = n)
  rddistri <- list(rnorm, runif, rpois)

    for (i in 1:nbdistri)
  {
    choosedistri <- sample(1:length(rddistri), size = 1)
    if(choosedistri %in% 1)
    {
      output <- output + rddistri[[choosedistri]](n, rnorm(1), runif(1,0,1))
    }
    if(choosedistri %in% 2)
    {
      output <- output + rddistri[[choosedistri]](n, 0, runif(1,0.1,3))
    }
    if(choosedistri %in% 3)
    {
      output <- output + rddistri[[choosedistri]](n, exp(runif(1)))
    }
  }
  plot(density(output))
  lines(x=seq(min(output),max(output),by = 1), y=dnorm(seq(min(output),max(output),by
  legend(x="topright", legend = c("simulation", "Normal distribution"), col=c("black"
  return(output)
}

res <- simclt()
```




```
shapiro.test(res)

##
##  Shapiro-Wilk normality test
##
## data:  res
## W = 1, p-value = 0.6
```

# 2 Funny things

## 2.1 The dot-dot-dot

### * Exercise 5 Understand the ...

Consider these two functions:

```
halfmean1 <- function(x)
{
  mean(x)/2
}
halfmean2 <- function(x,...)
{
  mean(x,...)/2
}
```

They behave the same way in the first case, but differently in the second:

```
somedata <- c(10, 25, NA)

halfmean1(somedata)

## [1] NA

halfmean2(somedata)

## [1] NA

halfmean1(somedata, na.rm=TRUE)

## Error in halfmean1(somedata, na.rm = TRUE): unused argument (na.rm = TRUE)

halfmean2(somedata, na.rm=TRUE)

## [1] 8.75
```

What does ... do?

**Answer of exercise 5**

...let you pass extra arguments to the function inside your function (but the extra arguments are passed to ALL functions inside, so be careful!)

## 2.2 The <<- operator

### * Exercise 6 Understand the <<-

We create to almost identical functions, f0 and f1 and add their output to an object `x`. Compare the output and explain what happens. What does `<<-` mean?

```
f0 <- function(x=2){
  x <- x
  y <- x+2
  return(y)
}

f1 <- function(x=2){
  x <<- x
  y <- x+2
  return(y)
}

x <- rnorm(1000)
f0()+x

x <- rnorm(1000)
f1()+x
```

**Answer of exercise 6**

The `<<-` operator breaks the borders of the function environment to assign a value to an object outside that function.

## 2.3 Recursive functions

### ** Exercise 7 What does this function do?

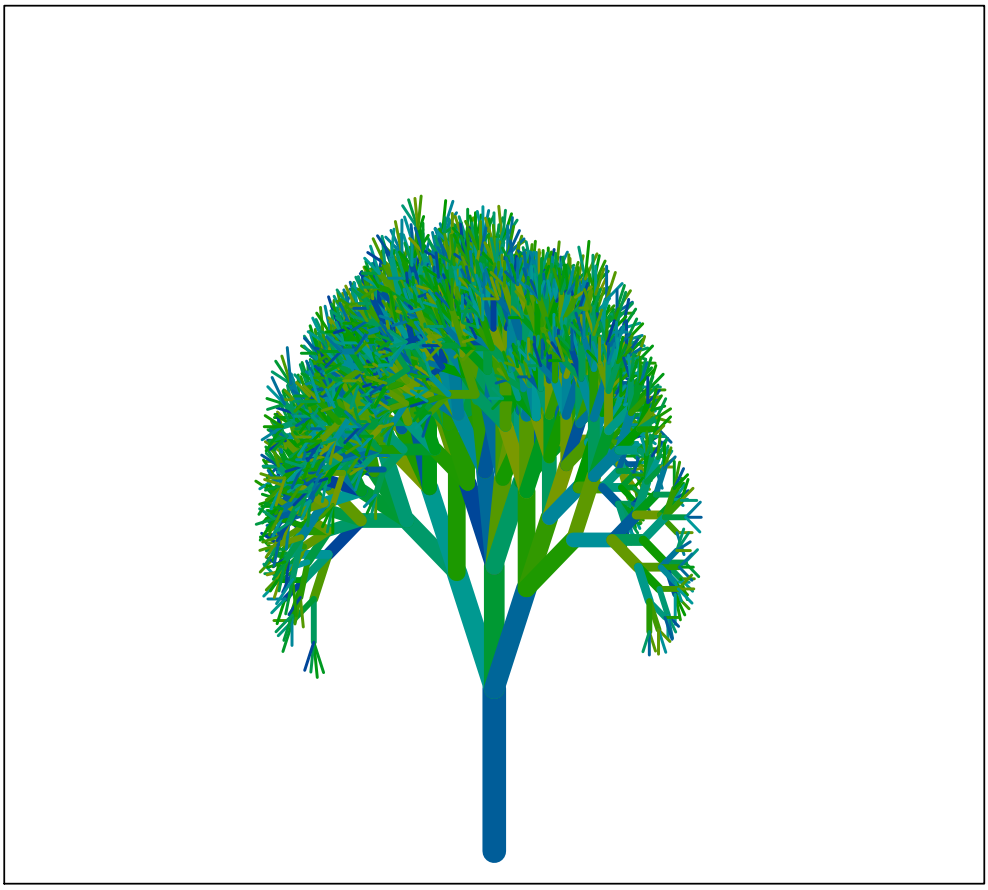
Consider the following function and try to understand what it does. (code by Dr. Koen van Benthem)

```
tree<-function(x,y,l,dir,n,nmax){
if(n==0){
return()
}
pos<-round(runif(1,1,199))
colour=rainbow(200,start=0.2,end=0.6,v=0.6)[pos]
lines(c(x,x+l*sin(dir)),c(y,y+l*cos(dir)),
      lwd=20*(n/nmax),col=colour)
branches<-round(runif(1,2,4))
for(i in 1:branches){
angle<-dir+(-pi/6)+(pi/3)*(i-1)/(branches-1)
l2<-runif(1,0.7,0.85)*l
tree(x+l*sin(dir),y+l*cos(dir),l2,angle,n-1,nmax)
}
}
```

You can run the function and see if you had guessed right:

```
plot(0,0,type="n",xlim=c(-10,10),ylim=c(0,10),
xaxt="n",yaxt="n",ylab="",xlab="")

tree(0,0,2,0,8,12)
```

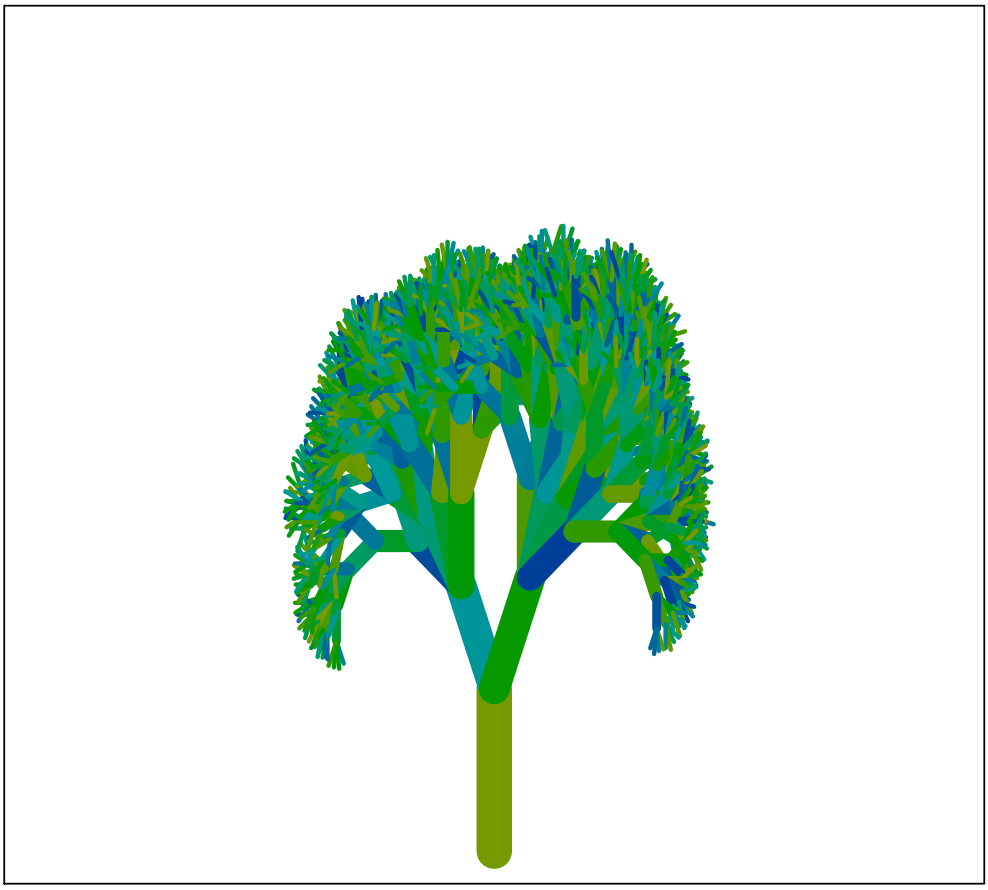


What is special about this function? Why should you be careful not to give large values to the parameters n?

**Answer of exercise 7**

This is a self-referencing function, it calls itself! Each call to the function produces several new calls to the function, and so the number of times the function is called grows exponentially with the value of n. I do not recommend values above 12. That is a general problem to consider with recursive functions.

```
tree<-function(x,y,l,dir,n,nmax){ # We define a function 'tree()'
# x,y: start of tree, l: length of first branch, dir: direction
# n and nmax: should be the same number: number of levels in the
# tree.
# An important escape argument, leave this out and the function
# will run forever:
if(n==0){
return()
}
# Picking a colour at random using the rainbow function
# (without this the code would also work, there would just be
# no colors)
pos<-round(runif(1,1,199))
colour=rainbow(200,start=0.2,end=0.6,v=0.6)[pos]
# Draw a line starting a (x,y) with length 'l' and in direction
# (in radials) dir. On top of that, we make the width of the
# line depend on how far the branch is from the stem.
lines(c(x,x+l*sin(dir)),c(y,y+l*cos(dir)),
      lwd=20*(n/nmax),col=colour)
# Generate a random number that defines how many branches the
# tree has at this point in the structure
branches<-round(runif(1,2,4))
# Now we go over the separate branches
for(i in 1:branches){
# to make sure not all branches point in the same direction,
# we calculate a direction for the branch
angle<-dir+(-pi/6)+(pi/3)*(i-1)/(branches-1)
# Also, we would like the later branches to be smaller than
# the first one:
l2<-runif(1,0.7,0.85)*l
# And finally the magic of recursion, we draw the new branch
# simply by using theexact same function: the function 'tree'.
tree(x+l*sin(dir),y+l*cos(dir),l2,angle,n-1,nmax)
}
}
# Now, to actually draw the tree, we first make an empty plot
plot(0,0,type="n",xlim=c(-10,10),ylim=c(0,10),
xaxt="n",yaxt="n",ylab="",xlab="")
# And then call the function tree() with the parameters we like
tree(0,0,2,0,8,8)
```



## *** Exercise 8 Find a mistake in a family tree

The file `wrongpedigree.csv` contains a pedigree (a family tree) containing a mistake: a function we tried to apply on this dataset informed us that the family tree was impossible without more information. We suspect that we have assigned an individual as its own ancestor (this can happen with genetic reconstruction of parentage). Use a self-referencing function to scan the tree and find where the problem is.

### Answer of exercise 8

I found a solution using two functions: a recursive one, and one going using a for loop to go across incidivuals. Clearly that is not efficient. Feel free to think of a better way!

```
checkp <- function(iniind, focalind, ped, level=1, maxlevel=3,
                   idname = "animal")
{
  parents <- ped[as.character(ped[,idname])==
                   as.character(focalind),c(2,3)]
  circular <- 0
  for(j in 1:2)
  {
    if(!is.na(parents[j]))
    {
      if(as.character(iniind)==as.character(parents[j]))
      {
        circular <- circular+1
        warning("Individual", iniind, " is its own ancestor at level ",
                level,  " (pair ", parents[1],";", parents[2], ")")
      }else{
        if(level<maxlevel)
        {level <- level+1
        circular <- circular + checkp(iniind=iniind, focalind=parents[j],
                                      ped = ped, level=level)
        }
      }
    }
  }
  return(circular)
}
```

```
checkpedwrap <- function(ped, idname="animal", maxlevel=3)
{
  ped$circularity <- NA
  for (i in 1:nrow(ped))
  {
    ped$circularity[i] <- checkp(iniind=ped[i, idname],
                                 focalind = ped[i, idname],
                                 ped=ped, level =1,
                                 maxlevel=maxlevel, idname=idname)
  }
  return(ped)
}
```

```
ped <- read.csv("wrongpedigree.csv", stringsAsFactors = FALSE)
pedchecked <- checkpedwrap(ped=ped, idname = "animal", maxlevel = 3)

## Warning in checkp(iniind = iniind, focalind = parents[j], ped = ped, level
= level):  IndividualCN15-053 is its own ancestor at level 3 (pair CN15-F7;CN15-053)
```