*R Programming for Demographers*

# Flow control and Functions

Tim Riffe[1]

September 19, 2017

[1]First, a big thanks to Giancarlo Camarda, who has let me recycle his impeccable material for teaching this course. Higher order thanks are also due to the very same people that Giancarlo himself thanked, especially Sabine Zinn and Roland Rau. Sabine was my teacher in this very course in 2009 (cohort 5)! Part of these lectures has been freely inspired by courses that both Roland, Sabine, and Giancarlo taught in the past. Furthermore, in recent years, Adam Lenart, Fernando Colchero, and Silvia Rizzi have aided in teaching this course and improving handouts. On the other hand, it goes without saying that I am uniquely responsible for any deficiencies and mistakes you may find in handouts.

# Contents

# Chapter 1

# Flow control

## 1.1 Introduction

In this lecture we will learn some basic programming tools such as loops (as well as tips to avoid them). Though at a glance it will seem beyond practical application in demography, I assure you that you will use the skills covered today daily.

## 1.2 Conditional execution

Conditional execution is performed in `R` using the command/function `if`. It works like this: One first tests a condition. *Only* if this condition is true, the following code is executed. If it's not true, then the following code chunk gets skipped. The main syntax is:

```
if ( condition == TRUE ) do.this
```

(double equals is used when checking for equality) As actual example:

```
life.expectancy <- 60
if (life.expectancy > 80) print("Life Expectancy is relatively high")
# nothing happens!
life.expectancy <- 85
if (life.expectancy > 80) print("Life Expectancy is relatively high")

## [1] "Life Expectancy is relatively high"
```

As you can see, nothing happens after the first line starting with `if`. Only if the condition is true, is the result printed. Note that the function `print` is used to show an `R` -object in the console, like a character-string.

It is often the case that you do not only want to perform one action after the `if-` condition is `TRUE`. You can group several statements into one by putting curly braces around them. Have a look at the following simple example. Our idea is to assign the value "Japan" to the variable `country` *only if* life expectancy (`lifeexp`) is greater than 85.

```
country <- "no idea"
lifeexp <- 84
if(lifeexp > 85){ # open code chunk
  print("Life Expectancy is larger than 85")
  country <- "Japan"
}                 # close code chunk
country

## [1] "no idea"
```

Obviously the object `country` remains equal to “no idea” since `lifeexp` is not greater then 85. Like in previous cases, the indentation before `print...` and `country...` is not necessary, but it makes your code more elegant and decipherable in the future, i.e. with the indentation you visibly create a hierarchy in your program.

Like in other languages, it is also allowed in `R` to tell the interpreter what happens if the condition is not `TRUE` but `FALSE` via the `else` statement. See this example using 1 number randomly drawn from a Normal Distribution with mean equal to -10:

```
my.ran <- rnorm(1, mean=-10)
if(my.ran >= 0){
  print("My Random Number is 0 or Greater.")
  }else{
  print("My Random Number is Smaller than 0.")
}

## [1] "My Random Number is Smaller than 0."
```

As expected, `my.ran` is smaller than 0: the actual probability to get a different outcome is practically equal to 0.

When working with `if` constructions, one wants to use also other conditions. We have seen in Module 1, that we can specified both `AND` and `OR` constructions by the symbols `&` and `|`, respectively.

Let’s draw another number from the same distribution, but with mean equal to 10, and test whether both numbers are greater than zero, and whether at least one of them is greater than zero:

```
my.ran1 <- rnorm(1, mean=10)
my.ran > 0 & my.ran1 > 0

## [1] FALSE

my.ran > 0 | my.ran1 > 0

## [1] TRUE
```

We are pretty sure you get first `FALSE` and then `TRUE`, right?

We can carry out the same condition element-by-element using two vectors (with random numbers from a Standard Normal Distribution) and assign the logical values to a new vector `cond`:

```
my.vec1 <- rnorm(5)
my.vec2 <- rnorm(5)
cond    <- (my.vec1 < 0) & (my.vec2 < 0)
```

It works also without brackets, but using the brackets makes the structure clearer. Let’s check the results merging the three vectors in a dataframe:

```
data.frame(vec1 = my.vec1, vec2 = my.vec2, cond = cond)

##         vec1       vec2  cond
## 1 -0.6432636  1.6913237 FALSE
## 2  0.2369012  0.5682343 FALSE
## 3 -1.5826924 -0.4397737  TRUE
## 4  1.2762327  0.2149550 FALSE
## 5 -1.3085721 -0.7625239  TRUE
```

In this case we would not bet much on the absence of `TRUE` values in the variable `cond`: there is 25% chance to obtain `TRUE` at each condition.

## 1.3 Repetitive execution

Besides the branching command `if`, there are several commands available in `R` for repetitive execution.

### 1.3.1 `for`-loops

The most common command for repetitive execution is the `for` loop. The syntax is

```
for(iterator in set.of.values){
  do.something
}
```

The parameter `iterator` takes on the first value of `set.of.values` in the first loop and executes `do.something`. In the second run, the second value is taken from `set.of.values` and so on until the last value is taken from `set.of.values` and `do.something` is executed the last time.

`R` accepts numeric values as well as character vectors and other things. See the following examples for clarification:

```
countries <- c("AUT", "BEL", "DEU", "NLD", "GBR", "USA")
for (i in countries) {
    print(i)
}

## [1] "AUT"
## [1] "BEL"
## [1] "DEU"
## [1] "NLD"
## [1] "GBR"
## [1] "USA"

for (i in 1:4) {
    print(countries[i])
}

## [1] "AUT"
## [1] "BEL"
## [1] "DEU"
## [1] "NLD"
```

These were very trivial examples and we did not made any operation within the loop. The next slightly more complex example shows how to compute life expectancy over calendar years by looping over columns of a given matrix.

In a continuous framework we can compute life expectancy by

$$e^0 = \frac{\int_0^\infty l(a)da}{l(0)}$$

See (Preston et al., 2001, p. 61).

Assuming mortality can only happens on January 1 of each year, we can approximate the previous formula by:

$$e^0 = \frac{\sum_0^\omega l_x}{l_0}$$

where, in our case, $l_0$ is the radix of the life table equal to $10^5$.[1]

The file `lxJPNfem.txt` is a matrix of survivors from the life tables ($l_x$) taken from the Human Mortality Database (2015) for Japanese women from 1950 to 2009 over the complete age range (0-110). Rows and columns are indexed by ages and years, respectively.

First we read in the dataset, and it is always a good habit to set up the objects to be used for the problem in hand:

```
lx    <- read.table("lxJPNfem.txt")
ages  <- 0:110
years <- 1950:2009
l0    <- 10^5
```

[1]Will this overestimate or underestimate $e^0$?

We have all the elements for computing $e_0$ for each calendar year for Japanese women. First we create an object (`e0`) which is a vector of zeros of the required length; then for each column of the matrix `lx` we apply the previous equation by summing up its elements and dividing this summation by the scalar `l0`. The outcomes are then passed to the $i$-th position of the previously built object `e0`. In this way at the end of the `for`-loop, which should run from 1 to the length of the years, we have collected all life expectancy in the object `e0`. In `R` :

```
e0       <- rep(0, length(years))
for(i in 1:length(e0)){
  sum.lx <- sum(lx[, i])    # i moves over the columns
  e0[i]  <- sum.lx / l0
}
```

We show the development of the life expectancy for Japanese women with a simple scatter-plot of the elements of `e0` vs. `years`: See the outcome in Figure 1.1.

### 1.3.2 `while`-loops

Less often used, but very useful for mathematical and statistical computation is the `while`-loop. Specifically this loop is very valuable for iteration problems. The syntax is a bit simpler than in the case of the `for`-loop:

```
while (parameter is true) dosomething
```

See the following code-piece:

```
x <- 10
while(x>1){
  x <- x / 2
  print(x)
}

## [1] 5
## [1] 2.5
## [1] 1.25
## [1] 0.625
```

The first line sets $x = 10$. The condition that needs to be fulfilled is $x > 1$ and, as long as this condition is true, a command $x = \frac{x}{2}$ is repeatedly applied. In words the command within the loop says "the new value of $x$ is equal to the old value of $x$ divided by two". In other words this example will start with the number 10, keep on dividing it by two until a number less than one is obtained and then stop. In order to follow what the loop is doing, we additionally print the updated value of $x$.

As a demographic example, we take the Swedish population in 2000 and assuming a constant geometric growth rate equal to 5.3 ‰, we test in which year this population will reach 10 million. First we set the starting population, the starting year and the known $r$ (geometric growth rate):

```
N  <- 8860859
yr <- 2000
r  <- 5.3 / 1000
```

Then we run a `while`-loop until the object `N` is no longe smaller than 10 million. Within the loop we update the population based on the following formula:

$$N_T = N_0(1 + r)$$

and we also update (increment) the year-object. The final line within the loop is use to concatenate and print the updated objects, `yr` and `N`. The argument `fill` breaks the output into successive lines.

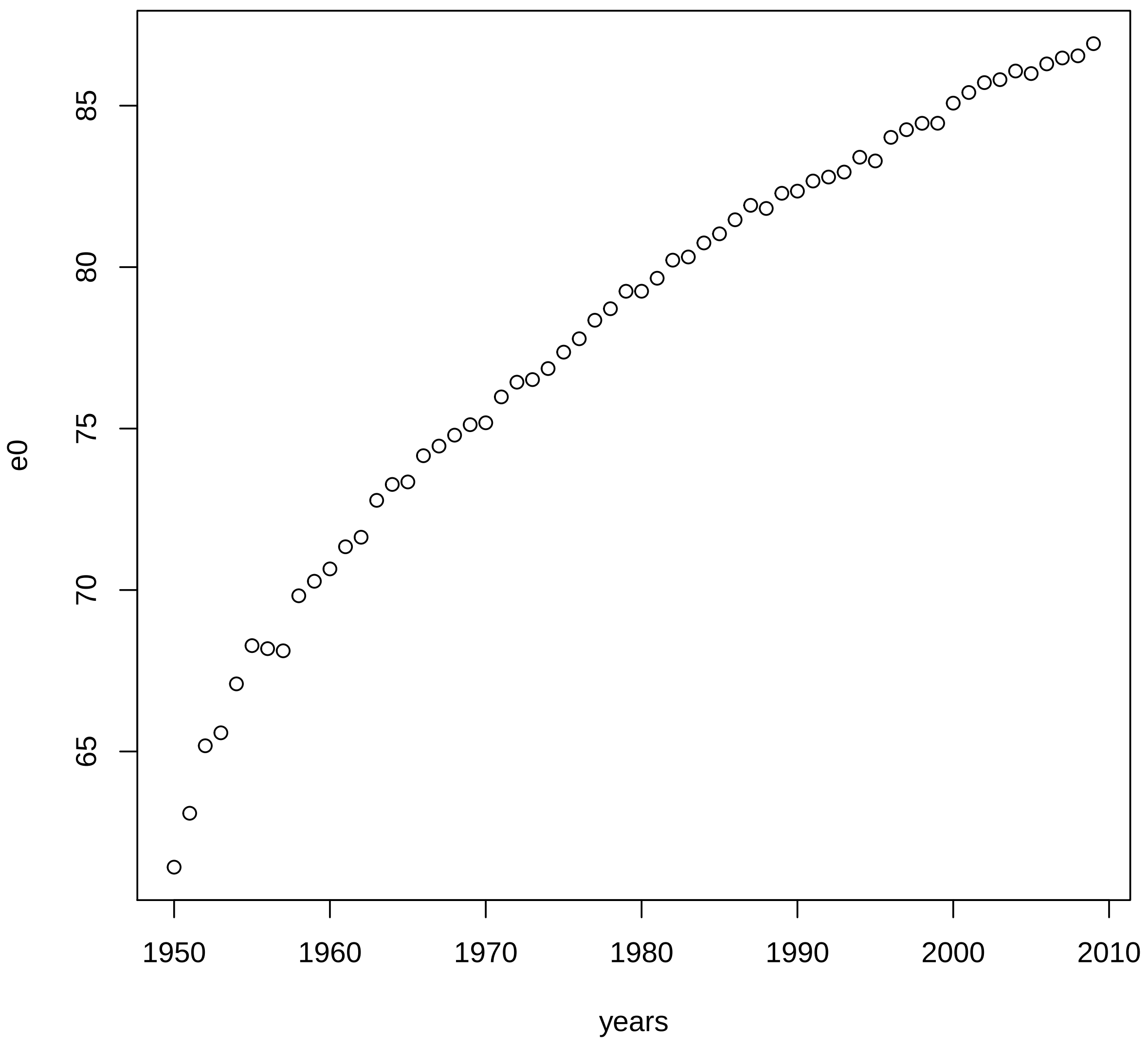


Figure 1.1: Life expectancy at birth from 1950 to 2009 for Japanese women

```
while(N < 10^7){
  N <- N * (1+r)
  yr <- yr+1
  cat(yr, N, fill=TRUE)
}

## 2001 8907822
## 2002 8955033
## 2003 9002495
## 2004 9050208
## 2005 9098174
## 2006 9146394
## 2007 9194870
## 2008 9243603
## 2009 9292594
## 2010 9341845
## 2011 9391357
## 2012 9441131
## 2013 9491169
## 2014 9541472
## 2015 9592042
## 2016 9642880
## 2017 9693987
## 2018 9745365
## 2019 9797015
## 2020 9848940
## 2021 9901139
## 2022 9953615
## 2023 10006369
```

Finally, given our specific assumptions, Swedish population will reach 10 millions in 2023.

Note that `while`-loop needs an increment/decrement in the `do.something` part (the body). Otherwise the program would end up in an infinite loop or would never start. When setting up a while-loop, just be certain that it will eventually stop!

If we run a `for`-loop until it reaches a given criterion, we can replace a `while`-loop. Specifically we need to inform the `for`-loop about a criterion on each new iteration and require it to stop when it is reached. We show this instance reproducing the last example with the Swedish population:

```
N <- 8860859
yr <- 2000
for(i in 1:100){
  N <- N * (1 + r)
  yr <- yr + 1
  cat(yr, N, fill = TRUE)
  if(N > 10 ^ 7) break
}

## 2001 8907822
## 2002 8955033
## 2003 9002495
## 2004 9050208
## 2005 9098174
## 2006 9146394
## 2007 9194870
## 2008 9243603
## 2009 9292594
## 2010 9341845
## 2011 9391357
## 2012 9441131
## 2013 9491169
## 2014 9541472
## 2015 9592042
## 2016 9642880
## 2017 9693987
## 2018 9745365
## 2019 9797015
## 2020 9848940
## 2021 9901139
## 2022 9953615
## 2023 10006369
```

The key command here is `break` which followed a conditional execution with the selected criterion. Note that in this setting we can specify the maximum number of iterations (here 100): a useful aspect in case of unknown computational times of the loop.

## 1.4 Vectorization

When changes or operations are to be made for values (some or all) in an array(matrix), a possible alternative to some of these control structures is to use vectorization methods. In other words we will learn how to use functions that operate on all the values in a matrix or an array "simultaneously".[2]

### 1.4.1 The logical commands

For applying particular condition to each element of a vector we create a vector of random systolic blood pressure for 10 observations from a Normal Distribution with mean 125 and standard deviation equal to 10. We round the outcomes to mimic an actual scenario. Then we use a `for`-loop which fills an empty vector called `assessment` using a conditional execution about the current systolic bloop pressure: if the value is lower than 130, then the observation could be `"OK"`, otherwise we have a `"Critical"` situation. In `R` :

```
systolic.bp <- round( rnorm(10, mean = 125, sd = 10) )
assessment  <- rep(0, length(systolic.bp))
for(i in 1:length(systolic.bp)){
    if(systolic.bp[i] < 130){
        assessment[i] <- "OK"
    }else{
        assessment[i] <- "Critical"
    }
}
```

Instead of this tedious double `if-else` into a `for`-loop, one can easily use the function `ifelse`. The syntax of is: `ifelse(condition, true, false)`.

```
assessment <- ifelse(systolic.bp < 130, "OK", "Critical")
```

And why not nicely frame the outcome?

```
cbind(systolic.bp, assessment)

##       systolic.bp assessment
##  [1,] "127"       "OK"
##  [2,] "132"       "Critical"
##  [3,] "145"       "Critical"
##  [4,] "113"       "OK"
```

[2]Or at least it appears so...

```
##  [5,] "136"       "Critical"
##  [6,] "127"       "OK"
##  [7,] "123"       "OK"
##  [8,] "123"       "OK"
##  [9,] "131"       "Critical"
## [10,] "121"       "OK"
```

We can generalize the logical operators `AND` and `OR` in order to apply them elementwise. We need just to double the symbols:

```
myrandom <- rnorm(10)
(myrandom < 0) && (myrandom < -1)

## [1] TRUE

# compare with:
(myrandom < 0) & (myrandom < -1)

##  [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE

# same behavior as &&:
all((myrandom < 0) & (myrandom < -1))

## [1] FALSE
```

The result is `FALSE` if any element is greaterthan -1. Vice versa, `TRUE` is obtained with just one element smaller than -1.

Another example with the condition `||`

```
myrandom > -2.58 || myrandom < 2.58

## [1] TRUE
```

In this case we can be rather sure that the result would be `TRUE`: with 10 numbers from a standard normal distribution it would be really easy to get at least a number in [-2.58, 2.58]

### 1.4.2 Avoiding `for`-loops

Sometimes we overuse `for`-loops and any kind of flow-controls.

Often the procedure is intuitively iterative, but not from a programming point of view. Moreover using specific functions for avoiding loops can speed-up the code and makes it more elegant[3].

Good functions for considerable speed-up belong to the apply family such as `apply`, `lapply`, `tapply`, `sapply` which map a function to all elements of a given data structure. Let's see some examples of these constructs:

#### apply()

First we create a 10 by 10 matrix:

```
M <- matrix(rnorm(100), 10, 10)
```

We can naively compute the mean of each column with a `for`-loop:

```
m        <- rep(0, ncol(M))
for(i in 1:length(m)){
    m[i] <- mean(M[, i])
}
m
```

[3]As E.W. Dijkstra used to say: in programming "elegance is not a dispensable luxury, but a factor that often decides between success and failure." (Dijkstra, 1996)

```
##  [1] -0.42040596  0.10654636 -0.69037540  0.13925652 -0.22624439
##  [6] -0.16749009 -0.04967842 -0.32373646 -0.33699525 -0.11203101
```

or simply use the function `apply()`. Its syntax can be written such:

`apply(myobject, in_which_direction, which_function)`

where the directions are either rows (1) or columns (2)[4]. In the previous example one could just type:

```
apply(M, 2, mean)

##  [1] -0.42040596  0.10654636 -0.69037540  0.13925652 -0.22624439
##  [6] -0.16749009 -0.04967842 -0.32373646 -0.33699525 -0.11203101
```

obtaining the same outcome. Of course we can use any function `R` provides and also our own functions (see Module 4).

However, `R` occassionally has built in gold-nuggets for just such situations, and these always perform much faster still:

```
colMeans(M)

##  [1] -0.42040596  0.10654636 -0.69037540  0.13925652 -0.22624439
##  [6] -0.16749009 -0.04967842 -0.32373646 -0.33699525 -0.11203101
```

However, not every function has a special built-in vectorized version of itself, and thus one often needs to use `apply()`. A nice feature of `apply()` is also the possibility to give optional arguments to the "internal" function. As example:

```
apply(M, 1, weighted.mean, w = 1:10)

##  [1]  0.15805932 -0.14577570 -0.45948549  0.03488070 -0.23152273
##  [6]  0.07505364 -0.21131536 -0.75092229  0.13006575 -0.58967126
```

In this case the argument `w=` belongs to the function `weighted.mean`(see `?weighted.mean`), but since we are using such function within an `apply()` command we can use also its arguments. In this case we compute for each row a weighted mean of their elements where the weights increase from 1 to 10.

The function `apply()` can be applied also into an array. Here is a short example to note how to assign the dimensions:

```
myarray <- array(runif(48), dim=c(2,8,3))
```

In this case the possible directions for `apply()` are 3, namely row-column-layer. As example we may write:

```
mean.row <- apply(myarray,1,mean)
mean.row

## [1] 0.4862156 0.5969544

mean.col <- apply(myarray,2,mean)
mean.col

## [1] 0.7730462 0.4403505 0.6402805 0.3647764 0.5237053 0.7220901 0.4731452
## [8] 0.3952859

mean.lay <- apply(myarray,3,mean)
mean.lay

## [1] 0.5554041 0.5566353 0.5127156
```

[4]Note that one can use both directions just typing as second arguments `c(1,2)`. This feature turns out to be useful using this function on an array

We can use the `apply()` command to shorten and make more elegant the computation of life expectancy at birth for Japanese women from 1950 to 2009. Instead of the previous `for`-loop, we just need to type the command:

```
e0.new <- apply(lx, 2, sum)/l0
```

in which we apply to each column of `lx` the function `sum` obtaining a vector which is divide by the radix of the life table `l0`.

As you might expect, we can be even more efficient still:

```
e0.new.new <- colSums(lx) / l0
```

You can check that all three procedures lead to the exact same results.

The matrix of survivors in `lx` could be also used to compute the other demographic indicators.

The median age at death is defined as the age such that $l_x$ is equal to $l_0/2$. Obviously this would work in a continuous framework, with our data we need to assume a constant force of mortality over each age and search for the last age in which $l_x$ is just over $l_0/2$.

To compute the median age at death we first apply to each cell of `lx` the *bigger than* (>) command, specifying to which value we compare. This operation create a new matrix fills with `FALSE` and `TRUE` for all the ages in which the values of `lx` are below or above `l0/2`, respectively. [5]

```
under.over <- lx > l0 / 2
```

Have a look at the outcome, showing about every 10 rows and columns:

```
under.over[seq(1, 111, by = 10), seq(1, 51, by = 10)]
```

We have seen in Module 1 that logical values are efficiently stored as 0/1 values, therefore if we sum them up we basically count the number of `TRUE`. In our example the number of `TRUE` within each column are all the ages above $l_0/2$. Hence we can apply to each column of `under.over` the function `sum` obtaining the median age at death. Type the following commands to compute the median ages and plot them over years (Figure 1.2):

```
median.age <- apply(under.over, 2, sum)
#plot(years, median.age)
```

### lapply()

The function `lapply()` works analogously to the `apply()`-command. The only difference is that the function is not applied to rows and columns of an array, but to the elements of a list.

Before a demographic application we apply this command to random number from a Normal Distribution:

```
L <- list(a = rnorm(1000, mean=10, sd=2),
          b = rnorm(100, mean=10, sd=4),
          c = rnorm(10000, mean=10, sd=6))
lapply(L, quantile, prob = seq(from = 0, to = 1, by = 0.2))

## $a
##        0%       20%       40%       60%       80%      100%
##  3.372206  8.240400  9.471282 10.420886 11.656315 17.046469
##
## $b
##        0%       20%       40%       60%       80%      100%
##  3.691086  6.753027  9.247144 11.238039 13.194069 18.850768
##
## $c
##         0%        20%        40%        60%        80%       100%
## -17.020410   4.770218   8.336345  11.319422  14.896818  33.943621
```

[5]In the session 2 class-script, we did something very similar using a monotonic spline. Give it a try on `lx`!

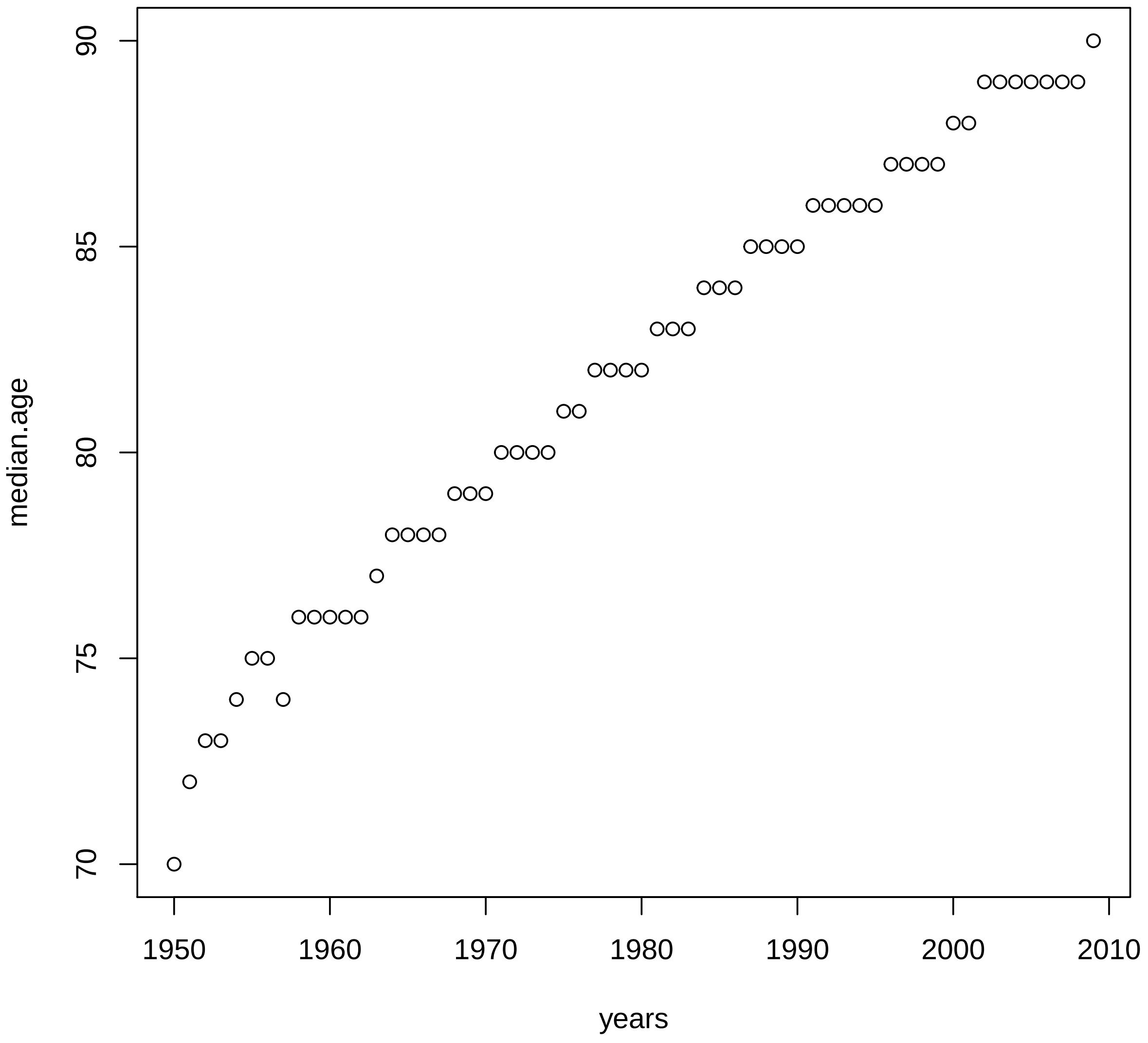


Figure 1.2: Median age at death from 1950 to 2009 for Japanese women

The outcome of `lapply()` is a list with the same number of elements like the input list. `prob` is an argument of `quantile()` (see: `?quantile`).

As a demographic example we will simultaneously compute life expectancy at birth and Total Fertility Rates (TFR) for Japan in 2000. From Preston et al. (2001, p. 95), we know that TFR is given by:

$$TFR = \sum_{x=\alpha}^{\beta} F_x$$

where $F_x$ are the age specific fertility rates and $\alpha$ and $\beta$ are the minimum and maximum age at childbearing.

The file `FertJPN.txt` contains fertility data for Japan in 2000. Specifically it has two columns: ages and age-specific fertility rates. Let's load the data and quickly look at them:

```
FertJPN <- read.table("FertJPN.txt")
head(FertJPN)

##   ages    asfr
## 1   13 0.00001
## 2   14 0.00006
## 3   15 0.00027
## 4   16 0.00132
## 5   17 0.00383
## 6   18 0.00732
```

Both life expectancy at birth and TFR are a product of a summation over vectors which have different length. We can thus merge both information into a list and simultaneously compute them. First we extract the survival function from the object `lx` in 2000 and the age specific fertility rates from `FertJPN`:

```
mort <- lx[, years==2000]/10
fert <- FertJPN$asfr
```

Note that instead of manually searching for the right column in `lx`, we used a relational operator for picking the survivors in 2000 calling the vector `years`. Then we divide the life-table survivor function by its radix (`10`) to obtain the survival probability function. For the fertility data the job is simpler: extract a variable from a dataframe.

Now we can merge this information in a list and compute $e_0$ for females and TFR for Japan in 2000:

```
listJPN <- list(mort=mort, fert=fert)
lapply(listJPN, sum)

## $mort
## [1] 85.07751
##
## $fert
## [1] 1.3591
```

### sapply()

The function `sapply()` is a user-friendly generalization of the function `lapply()`. It returns a vector or matrix if appropriate.

```
sapply(L, quantile, prob = seq(0, 1, by = 0.2))

##              a         b          c
## 0%    3.372206  3.691086 -17.020410
## 20%   8.240400  6.753027   4.770218
## 40%   9.471282  9.247144   8.336345
## 60%  10.420886 11.238039  11.319422
## 80%  11.656315 13.194069  14.896818
## 100% 17.046469 18.850768  33.943621
```

```
sapply(listJPN, sum)

##     mort     fert
## 85.07751  1.35910
```

The outcome of `sapply()` is a matrix.

**tapply()**

The `tapply()` function is useful when we need to break up a vector into groups defined by some classifying factor, compute a function on the subsets, and return the results in a convenient form.

Specifically the arguments of the function `tapply()` are the vector we aim to break and compute the function from, the classifying factor and the function we aim to apply.

To see an example of this useful function, we upload a dataset about life expectancy at birth (for both sexes) and Gross Domestic Product (GDP) based on purchasing-power-parity (PPP) per capita in US dollars. The data are given in the file `GDPe02009.txt`.

Both $e_0$ and GDP refer to estimates for 2009 and they are taken from the World Bank and World Health Organization web-sites, respectively. Additionally we include information about the coutry names as well as the geographical region and the World Bank classification according to the incomes.

First we load the data and look at them:

```
GDPe0 <- read.table("GDPe02012.txt", header = TRUE, sep = ",")
head(GDPe0)

##               Country GDPcapPPP e0                 Region
## 1         Afghanistan      1121 60 Eastern Mediterranean
## 2             Albania      9207 74                 Europe
## 3             Algeria      7305 72                 Africa
## 4              Angola      6092 51                 Africa
## 5 Antigua and Barbuda     18197 75               Americas
## 6           Argentina     17917 76               Americas
##        WorldBankGroup
## 1          Low-income
## 2 Lower-middle-income
## 3 Upper-middle-income
## 4 Lower-middle-income
## 5         High-income
## 6 Upper-middle-income
```

As an example we compute the mean GDP per capita by the geographical regions:

```
tapply(GDPe0$GDPcapPPP, GDPe0$Region, mean)

##                Africa              Americas Eastern Mediterranean
##              4279.152             14379.559             17469.850
##                Europe       South-East Asia       Western Pacific
##             23682.180              6455.900             15793.625
```

Within `tapply()`, we can specify multiple factors as the grouping variable, for example we can compute the mean of the life expectancy at birth by geographic regions and income groups:

```
tapply(GDPe0$e0,
       list(GDPe0$Region, GDPe0$WorldBankGroup),
       mean)

##                       High-income Lower-middle-income Low-income
## Africa                   55.00000            57.55556   58.44828
## Americas                 76.50000            71.88889   62.00000
## Eastern Mediterranean    77.00000            69.44444   59.66667
## Europe                     80.55556           70.85714   68.66667
## South-East Asia                  NA           71.14286   68.00000
## Western Pacific            81.66667           69.36364   71.33333
##                         Upper-middle-income
## Africa                             67.28571
## Americas                           76.11111
## Eastern Mediterranean              77.50000
## Europe                             73.92308
## South-East Asia                          NA
## Western Pacific                    71.00000
```

The `NA` outcomes shows that no country belongs simultaneouly to the `South-East Asia` region and to either the `High-income` or the `Upper-middle income` group.

## 1.5 Exercises

### 1.5.1 Exercise 1

In our data-folder you will find six files: `deaDENf.txt`, `popDENf.txt`, `deaFRAf.txt`, `popFRAf.txt`, `deaITAf.txt` and `popITAf.txt`. As you may guess from the names, they contain female population of Denmark, France and Italy by age (0-99) and year (1950-2000) as well as the number of people dead for the same age- and period-range.

(i) Read all files

(ii) Coerce each data.frame to be a matrix

(iii) Using a `for`-loop, compute the total population of Denmark in each year

(iv) Using `apply` compute the mean number of deaths for both France and Italy by single age

(v) Create two new matrices which sum up the three populations (`TOTpop`) and deaths (`TOTdea`)

(vi) Calculate a matrix with age- and year-specific death rates `m.xy` for the three countries combined following $m(x,y) = \frac{D(x,y)}{N(x,y)-0.5 \cdot D(x,y)}$

### 1.5.2 Exercise 2

This exercise presents a typical programming exercise: computing square root of a number by Newton's Method[6].

We can define the square-root function as:

$$\sqrt{x} = y \text{ such that } y \geq 0 \text{ and } y^2 = x$$

How does one compute square roots? For instance, Wikipedia lists 14 different ways, but we will exercise our programming skills with only one of them which uses Newton's method of successive approximations: whenever we have a guess $\tilde{y}$ for the value of the square root of a number $x$, we can modify it for getting a guess closer to the actual square root. This operation is the average between $\tilde{y}$ and $\frac{x}{\tilde{y}}$.

For example, we can compute the square root of $x = 3$ as shown in Table 1.1, supposing the our initial guess is $\tilde{y} = 1$. So, one has basically 3 steps which have to repeat changing the so-called "guess" and the subsequential outcomes.

The exercise asks you to create a `for`-loop for computing square roots of a given number, let's say $x = 123$.

Hints: set the object $x$ and an initial values such as 5. Then run the loop for (eventually) 100 times and stop it whenever the absolute difference between the current guess and the previous one is smaller than $10^{-4}$.

Given the suggested starting value, you should need only a handful of iterations to satisfy the mentioned tolerance level.

[6]For more information see `en.wikipedia.org/wiki/Newton's_method`.

| Guess: $\tilde{y}$ | Quotient: $\frac{x}{\tilde{y}}$ | Average: $\frac{\text{Quotient}+\tilde{y}}{2} = \text{new.guess}$ |
|---|---|---|
| 1 | $\frac{3}{1} = 3$ | $\frac{(3+1)}{2} = 2$ |
| 2 | $\frac{3}{2} = 1.5$ | $\frac{(1.5+2)}{2} = 1.75$ |
| 1.75 | $\frac{3}{1.75} = 1.7143$ | $\frac{(1.7143+1.75)}{2} = 1.7322$ |
| 1.7322 | $\frac{3}{1.7322} = 1.7319$ | $\frac{(1.7319+1.7322)}{2} = 1.7321$ |
| 1.7321 | ... | ... |

Table 1.1: Simple scheme for explaining the computation of the square root of 3 by Newton's method of successive approximations.

# Chapter 2

# Functions

## 2.1 Introduction

One of the biggest advantages of R is its immense flexibility: if you are unhappy with a way something is implemented or if you think something is missing, you can simply define your own function. Many functions in R have also been programmed that way. For example, the standard deviation, sd:

```
sd

## function (x, na.rm = FALSE)
## sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
##     na.rm = na.rm))
## <bytecode: 0x3a13d38>
## <environment: namespace:stats>
```

Although you can write programs in R without defining your own functions, we would recommend that you do it anyway because it allows you to be lazy (you don't have to write the same thing over and over again), your programs become more readable, and, most importantly: it is easier to maintain your programs. If you have made a mistake (which will happen for sure, we promise!), it is easier to correct the function definition than going through the whole program and try to correct every occurrence of a specific calculation.

## 2.2 The basic structure (with simple examples)

Functions are always built in the same way. The syntax is:

```
nameoffunction <- function(argument1, argument2, ...) {
  function-body using argument1, argument2, ...   and producing myresult
  return(myresult)
}
```

A very simple example is to write a function that calculates the square of a given input (which could be a scalar or a vector).

```
mysquare <- function(x) {
  myresult <- x * x
  return(myresult)
}
mysquare(4)

## [1] 16

mysquare(1:12)

##  [1]   1   4   9  16  25  36  49  64  81 100 121 144
```

## 2.3 The individual parts of a function definition

`nameoffunction` : In `R` , you can give functions you define yourself any kind of name. There is no restriction on the length of the name. As far as we know, any kind of object in `R` (data, functions, ...) has to start with an alphabetic letter and contains afterward also numbers. We would suggest that you use a name which makes it clear what is being done. Better make sure that you do not overwrite an internally defined function with your newly defined function. Examples for legal function names are:

```
myfunction
my.function
GC.function
DoomsdayFunction
...
```

`<- function` : then you tell `R` that the object you create is a function.

`argument1, argument2, ...` : after the `function`-keyword, you can give `arguments` that make up the *input* for your function. The minimum number of `arguments` is zero:

```
hello <- function(){
  print("Hello my friends")
}
```

But this is rather the exception. Usually, the function should perform different things according to the input. Let's define, for example, a power function which takes as input a scalar or a vector $x$ and an exponent $y$. The function should return $x^y$.

```
mypower <- function(x, y) {
  result <- x^y
  return(result)
}
mypower(2, 3)

## [1] 8
```

To be more precise, we recommend to give the names of the arguments:

```
mypower(y = 3, x = 2)

## [1] 8
```

If you want to give some default values for some arguments you can do this as in the following example where we assume that the most often used power-function is to square the base.

```
mypower <- function(x, y = 2) {
  result <- x^y
  return(result)
}
mypower(2,3)

## [1] 8

mypower(2,2)

## [1] 4

mypower(2)

## [1] 4
```

`function-body` : In the `function-body` you write all the statements and operations that you want to perform on the given arguments. This is the area where typically the data transformation/manipulation takes place with all the `if`, `for`, ... statements. Although this is the shortest part of the function description here, this is the most difficult to write. Here you can put in your creativity, effort, mistakes, ....

`return` : With the `return` statement you specify what the output of the function is. You can only return one object - but this object can be anything: a number, a matrix, a data.frame, nothing at all, a list, .... The return statement is actually not mandatory. If you omit it, the last evaluated term will be returned. Experience shows however, that it can get very difficult to spot this term in moderately complex programs. That is why we suggest using it explicitly.

## 2.4 A statistical example: computing the Pearson correlation coefficient

To demonstrate how you can write a function, let's provide here a slightly larger example to "walk through" line by line. Specifically, we aim to write a function to compute the Pearson correlation coefficient, $r$. Given two vectors $x$ and $y$, we want to apply the following formula (Agresti and Finlay, 1997, p. 354):

$$r = \frac{\sum (X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum (X_i - \bar{X})^2 \sum (Y_i - \bar{Y})^2}}$$

Here what we can write in `R` :

```r
Pea.corr <- function(x,y) {
   # "Handmade" Pearson correlation, normally denoted by "r"
   # see Agresti, 1997, p. 354

   # calculate the means
   xmean <- mean(x)
   ymean <- mean(y)

   # numerator & denominator
   numerator   <- sum((x - xmean) * (y - ymean))
   denominator <- sqrt(sum((x - xmean)^2) * sum((y-ymean)^2))

   # compute the actual correlation
   r <- numerator / denominator

   return(r)
}
```

**Line 1** : Here you see the beginning of the function definition. It tells us that the new function has the name `Pea.corr` and that it takes two arguments, `x` and `y`.

**Line 2-3** : It has become good practice in many programming languages that you write a short documentation string in the second line of a function definition. With a short phrase you say what the function should do.

**Lines 5–7** : following the formula above, first calculate the means of `x` and `y`.

**Lines 09–11** : In these lines we calculate the numerator and the denominator separately.

**Line 13-14** : To obtain $r$, divide the numerator by the denominator.

**Line 16** : And finally we state that `r` is the object which should be returned from our "blackbox" `Pea.corr`.

Empty lines in between the operations and indentation are not necessary, but useful to increase code legibility. Let's use our function `Pea.corr` on random numbers:

```
myx <- rnorm(100)
myy <- rnorm(100)
Pea.corr(x=myx, y=myy)

## [1] 0.1187643
```

Of course in this case, we expect a rather low correlation: both `myx` and `myy` are independently randomized. Note that the object-names within the function can be completely different from the actual object we aim to use. Indeed we suggest using names for your in-script objects that are different from the argument-names in the function. This helps keep things clean and separate.

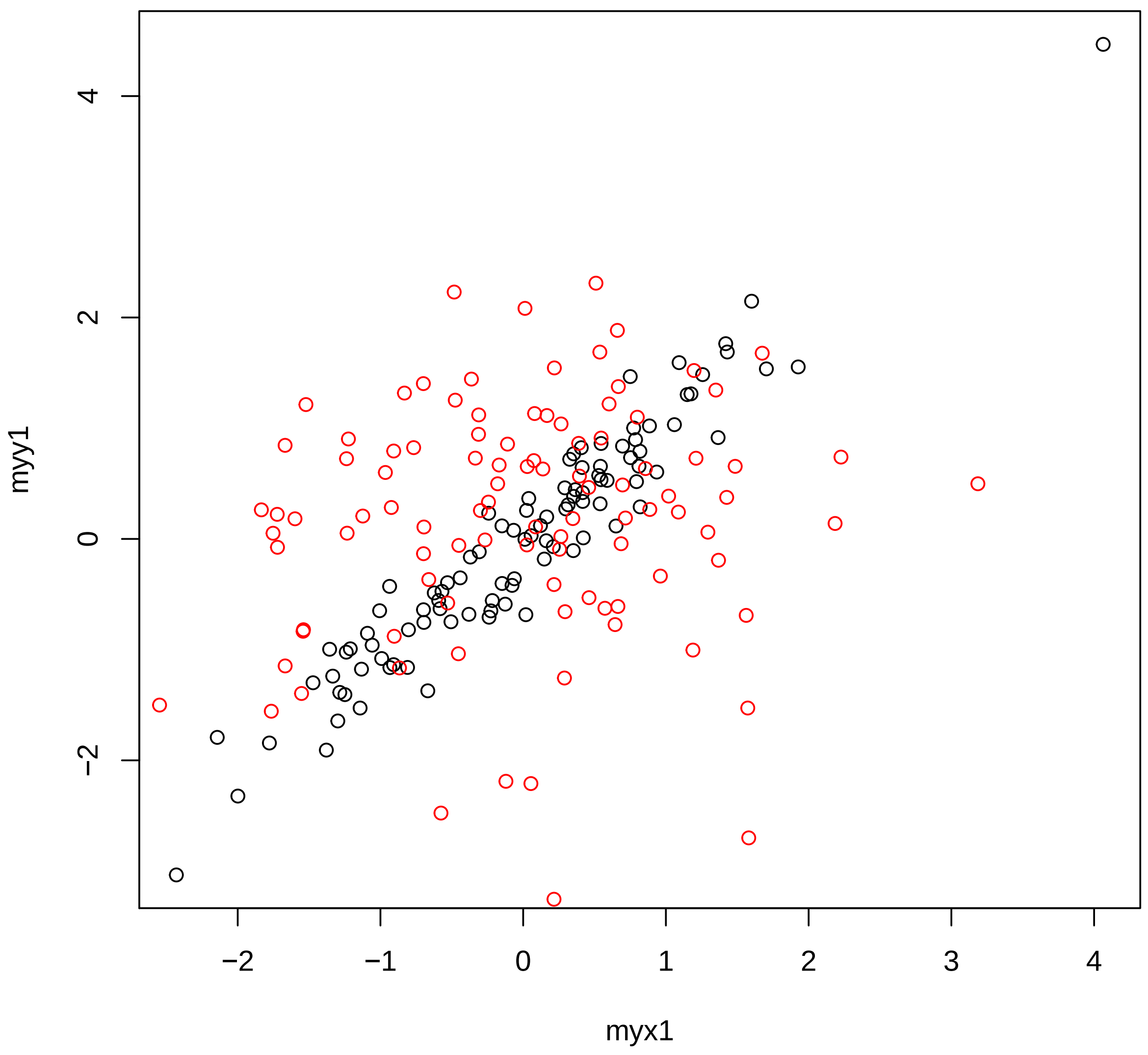
Let's simulate some data in which we ensure high correlation:

```
myx1 <- rnorm(100)
myy1 <- myx1 + rnorm(100, sd = 0.3)
Pea.corr(x = myx1, y = myy1)

## [1] 0.9617872
```

To check the correlation graphically you can plot `x` against `y` for both cases:

```
plot(myx1, myy1)
points(myx, myy, col = 2)
```



More information about plotting features in R will be given in a later module.

## 2.5 A demographic example: Age-standardization and decomposition

As those of your with a background in demography may know, sometimes the difference in age structures between two populations has a major influence on the crude death rates that distorts comparisons. Age-standardization is a tool for overcoming this issue(**?**, pp. 24–28).

Moreover it would be nice to quantify the difference between deaths rates in two population attributable to differences in their age distribution. In this case a decomposition of differences between rates helps (**?**, pp. 28–30).

In formulas the age-standardization:

$$\begin{aligned} ASCDR^A &= \sum_{i=1}^{\infty} M_i^A \cdot C_i^{ave} \\ ASCDR^B &= \sum_{i=1}^{\infty} M_i^B \cdot C_i^{ave} \end{aligned}$$

where $M_i^A = \frac{D_i^A}{N_i^A}$ and $M_i^B = \frac{D_i^B}{N_i^B}$ are the death rate in the $i$th interval for population $A$ and $B$, respectively.

The average age distribution is given by

$$C_i^{ave} = \left( \frac{C_i^A + C_i^B}{2} \right) .$$

We can summarize in formulas the decomposition of differences between rates such as (Kitagawa, 1955):

$$\begin{aligned}
\Delta &= CDR^A - CDR^B = \\
&= \sum_i (C_i^A - C_i^B) \cdot \left[ \frac{M_i^A + M_i^B}{2} \right] + \\
&\quad + \sum_i (M_i^A - M_i^B) \cdot \left[ \frac{C_i^A + C_i^B}{2} \right] = \\
&= \text{contribution of differences in age} + \\
&\quad \text{contribution of differences in rates}
\end{aligned}$$

The inputs for a function which computes those values are:

- Population by age in country $A$: `P.A`
- Deaths by age in country $A$: `D.A`
- Population by age in country $B$: `P.B`
- Deaths by age in country $B$: `D.B`

A desirable output should contain:

- Age standardized crude death rate for country $A$: `ASCDR.A`
- Age standardized crude death rate for country $B$: `ASCDR.B`
- Contribution of age compositional differences: `comp.diff`
- Contribution of age-specific rate differences: `ASR.diff`

With some comments, using the procedures in **?** and following what we have written in Section 2.4, we write our function:

```r
stand.decom <- function(P.A, D.A, P.B, D.B){
    # CRUDE RATES
    A.crude     <- sum(D.A) / sum(P.A)
    B.crude     <- sum(D.B) / sum(P.B)
    Diff.crude  <- A.crude - B.crude

    # age distribution of country A and B
    C.A         <- P.A / sum(P.A)
    C.B         <- P.B / sum(P.B)

    # age-specific death rate in country A and B
    M.A         <- D.A / P.A
    M.B         <- D.B / P.B

    #### AGE-STANDARDIZATION
    # average age distribution
    C.ave       <- (C.A + C.B)/2

    # age-standardized crude death rate for country A and B
    ASCDR.A     <- sum(M.A * C.ave)
    ASCDR.B     <- sum(M.B * C.ave)
    #### DECOMPOSITION OF DIFFERENCES BETWEEN RATES
    comp.diff   <- sum((C.A - C.B) * ((M.A + M.B)/2))
    ASR.diff    <- sum((M.A - M.B) * C.ave)

    # preparing the outcomes
    outcome     <- c(Astand = ASCDR.A,
                     Bstand = ASCDR.B,
                     A.crude = A.crude,
                     B.crude = B.crude,
                     Diff.crude = Diff.crude,
                     Diff.comp = comp.diff,
                     Diff.rates = ASR.diff)
    # giving the outcomes
    return(outcome)
}
```

Let's use this function with the raw data which are used in Box 2.1 (**?**) on Sweden ($A$) and Kazakhstan ($B$):

```
ourdata <- read.table("swed_kaza.txt", header=TRUE)
head(ourdata)

##   pop.swe dea.swe pop.kaz dea.kaz
## 1   59727     279  174078    3720
## 2  229775      42  754758    1220
## 3  245172      31  879129     396
## 4  240110      33  808510     298
## 5  264957      61  720161     561
## 6  287176      87  622988     673

stand.decom(P.A = ourdata$pop.swe,
            D.A = ourdata$dea.swe,
            P.B = ourdata$pop.kaz,
            D.B = ourdata$dea.kaz)

##       Astand       Bstand      A.crude      B.crude   Diff.crude
## 0.007374094  0.011881997  0.010547561  0.007423042  0.003124519
##    Diff.comp   Diff.rates
## 0.007632422 -0.004507903
```

## 2.6 A second demographic example: Identifying the Intrinsic Growth Rate

In this section we show how to write a function for estimating the intrinsic growth rate in a stable population. Instead of following the approach proposed by Coale (1957) and described in Box 7.1 of **?**, we estimate this demographic value using Newton's method for root-finding. In this way we also introduce a popular numerical method, which might be useful in other situations.

Without going into details, we present the framework before carrying out the analysis in R . If age specific mortality and fertility rates are constant over time and net migration rates are zero at all ages, then Lotka (1939) showed that a stable population will emerge: A population with an invariable age structure and a fixed rate of natural increase.

Specifically he showed that, in such situation, a continuous model of population dynamics is described by the following equation:

$$1 = \int_{\alpha}^{\beta} e^{-ra}\, p(a)\, m(a)\, da\,,$$

where the survival and maternity (female-female) schedules over ages $a$ are represented by $p(a)$ and $m(a)$, respectively. The integral is computed between the minimum and the maximum age at childbearing: $\alpha$ and $\beta$. The intrinsic growth rate in the stable population will growth is denoted by $r$. Among many other consequences, this equation allows for an estimation of how a population igrows once stability is reached. In order to be operative we will first present its version in a discrete time for single-age intervals:

$$1 = \sum_{a=\alpha}^{\beta} e^{-ra} \, L_a \, m_a \; = y(r)$$

which we write also as a function that depends only on $r$: age, mortality and fertility schedules are given. Specifically, $L_a$ is the vector of lifetable exposure from a female period life table with $l_0 = 1$, and $m_a$ is the vector with the rate of bearing female children.

In order to search for the intrinsic growth rate that satisfies the previous equation, we have to find where the following function is equal to zero, i.e. find its root:

$$r : f(r) = y(r) - 1 = 0 \, .$$

A method for finding the roots (or zeroes) of a real-valued function is given by Newton's method. In general if we aim to

$$x : f(x) = 0 \, .$$

Starting from the Taylor-expansion of $f(x)$, this approach finds $x$ iteratively by computing the following equation:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \, ,$$

starting from an $x_0$ which is sufficiently close to the solution.

More details can be found under `en.wikipedia.org/wiki/Newton's_method`.

In the case of the intrinsic growth rate, the unknown is $r$ and the first derivative with respect to $r$ of $f(r)$ is given by:

$$f'(r) = y'(r) = -\sum_{a=\alpha}^{\beta} a \, e^{-ra} \, L_a \, m_a \, .$$

By dividing and multiplying $f'(r)$ by $y(r)$ we obtain:

$$f'(r) = y'(r) \cdot \frac{y(r)}{y(r)} = -\sum_{a=\alpha}^{\beta} a \, e^{-ra} \, L_a \, m_a \cdot \frac{\sum_{a=\alpha}^{\beta} e^{-ra} \, L_a \, m_a}{\sum_{a=\alpha}^{\beta} e^{-ra} \, L_a \, m_a} = -\, y(r) \, A_B$$

where $A_B = \frac{\sum_{a=\alpha}^{\beta} a \, e^{-ra} \, L_a \, m_a}{\sum_{a=\alpha}^{\beta} e^{-ra} \, L_a \, m_a}$ is the mean age at childbearing in the stable population.

We now have all the ingredients for estimating $r$ using Newton's method. This can be expressed in the following the steps:

1. set as starting value $r_0 = \frac{\ln(NRR)}{A_{B,0}}$ where the Net Reproduction Rate is computed as $NRR = \sum L_a m_a$ and $A_{B,0}$ is a starting value for the mean age at childbearing. Commonly it is assumed equal to 27, though one could also take the stationary $A_{B,0}$ (the weighted average of $a$, where $L_a * m_a$ are the weights). Reasons for such choices can be found in **?**, pp. 150-155.
2. compute $y(r_0)$ and $A_B$
3. compute $f(r)$ and $f'(r)$
4. update $r_1 = r_0 - \frac{f(r)}{f'(r)}$
5. iterate 2., 3. and 4. until convergence: $y(r)$ close to one or equivalently $f(r)$ close to zero

The inputs for our function are:

- a: the vector of ages (or mid-point of age-intervals)
- L: survival schedule. Practically, the female lifetable $L_a$ divided by its radix, $l_0$

- `m`: maternity schedule, rate of bearing female children.

These inputs need to be vectors of the same length. Moreover we can include complementary arguments such as

- `st.Ab=27`: a user-defined starting value for the mean age at childbearing. By default we set it equal to 27
- `MON=FALSE`: a logical value that tells the function whether to trace information on the progress of estimation. By default we could decide to avoid this trace.

Aside from the estimated intrinsic growth rate, we could ask the function to produce the following outcomes:

- `data`: the input information in a dataframe
- `yr`: the solution of Lotka's integral at the last iteration
- `Ab`: the internally computed mean age at childbearing, which depends on *r*
- `NRR`: the internally computed Net Reproduction Rate

Let's write down the function `IntGroRat`:

```
IntGroRat <- function(a,
                      L,
                      m,
                      st.Ab = 27,
                      MON = FALSE){
  ## function estimating intrinsic growth rate
  ## and other demographic values
  ## by Newton's method

  ## Net Reproduction Rates
  NRR    <- sum(L * m)
  ## starting r
  r      <- log(NRR) / st.Ab
  ## iteration
  for(i in 1:20){
    ## y(r)
    yr   <- sum(exp(-r * a) * L * m)
    ## Ab
    Ab   <- sum(a * exp(-r * a) * L * m) / yr
    ## f(r)
    fr   <- yr -1
    ## f'(r)
    f1r  <- - yr * Ab
    ## updating r
    r    <- r - fr / f1r
    ## printing iteration, Lokta's equation and r
    if(MON) {
      cat(i, yr, r, fill = TRUE)
    }
    ## check convergence: fr close to zero which
    ## is equivalent to yr close to 1
    if(abs(fr) < 10^-6) {
      break
    }
  }
  # one final update to Ab (which depends on r)
  Ab   <- sum(a * exp(-r * a) * L * m) / yr
  ## outputs
  out  <- list(data = data.frame(ages = a, mort = L, fert = m),
               yr = yr,
               Ab = Ab,
               NRR = NRR,
               r = r)
  return(out)
}
```

A difference with respect to the previously coded functions is that we set up a `for`-loop within it, and we thus incorporate a conditional execution for stopping the iterations at the convergence level. Moreover we include a user-defined conditional execution for eventually tracing the estimation procedure, and the printing is done by the command `cat()` (see Module 3). Though Net Reproduction Rate (NRR) and mean age of childbearing ($A_B$) are only by-products of the main procedure, they can also be included in the final output.

Finally we compose all the results into a list: This is because they have different structures. Note that the function does not depend on the length of the age, mortality, and maternity schedules, as long they have same length. We could thus apply our function to one-year-wide age interval, grouped ages as well as to data with different number of available ages.

In order to test `IntGroRat()` we apply it to two datasets: Egypt in 1997 and USA in 1991. Both datasets are taken from **?**, pp. 149-150.

First we load the datasets and quickly look at them:

```
egy <- read.table("FunEgypt1997.txt", header=TRUE)
egy

##      a       L       m
## 1 17.5 4.66740 0.00567
## 2 22.5 4.63097 0.06627
## 3 27.5 4.58518 0.11204
## 4 32.5 4.53206 0.07889
## 5 37.5 4.46912 0.05075
## 6 42.5 4.39135 0.01590
## 7 47.5 4.28969 0.00610

usa <- read.table("FunUSA1991.txt", header=TRUE)
usa

##      a       L      m
## 1 12.5 4.94603 0.0007
## 2 17.5 4.93804 0.0303
## 3 22.5 4.92552 0.0566
## 4 27.5 4.91138 0.0578
## 5 32.5 4.89356 0.0388
## 6 37.5 4.86941 0.0157
## 7 42.5 4.83577 0.0027
## 8 47.5 4.78475 0.0001
```

They have the same column structure:

- `a`: mid-point in the 5-year age intervals
- `L`: person-years from female life-table divided by the radix
- `m`: rate of bearing female children

While Egyptian data start from age 17.5 (interval 15-19), USA data present also information about age 12.5 (interval 10-14).

Now we can apply our function to both cases:

```
fitEgy <- IntGroRat(a = egy$a, L = egy$L, m = egy$m)
fitEgy

## $data
##   ages    mort    fert
## 1 17.5 4.66740 0.00567
## 2 22.5 4.63097 0.06627
## 3 27.5 4.58518 0.11204
## 4 32.5 4.53206 0.07889
## 5 37.5 4.46912 0.05075
## 6 42.5 4.39135 0.01590
## 7 47.5 4.28969 0.00610
##
## $yr
## [1] 1
##
## $Ab
## [1] 29.47247
##
## $NRR
## [1] 1.527414
##
## $r
## [1] 0.01424406

fitUsa <- IntGroRat(a = usa$a, L = usa$L, m = usa$m,
                    st.Ab = 28, MON = TRUE)

## 1 0.9997601 -0.0001664836
## 2 1 -0.0001664824

fitUsa$r

## [1] -0.0001664824
```

In the first case we use the default values for `st.Ab` and `MON`, and then we print the whole output. For the USA data, we change the starting value for the mean age at childbearing, we trace the iterations, and we extract from the list-object `fitUSA` only the value for the intrinsic growth rate, *r*.[1]

## 2.7 Exercises

### 2.7.1 Exercise 1

Create a function for computing square roots of any number using successive approximations. Hint: have a look and solve first exercise 2 presented in Module 3, where the approach is introduced.

Your function could start as follows:

```
forsqrt <- function(x, init.guess=1, max.it=10){
#[...]
}
```

where `x` is the number you aim to compute the square root, `init.guess` is the starting value from where you begin the iteration and by default you could set up to 1, `max.it` is the maximum number of iteration of your internal `for`-loop which is set to 10 by default.

You could also add into the function a warning that inform the user when the maximum number of iteration is reached, and therefore recommend to manually increase `max.it` to reach a good approximation of the square root (given the starting value).

Finally test your function for different values of $x$, including big numbers such as $10^7$.

[1]Note that our estimated *r* for USA in 1991 is different from the one presented in **?**, p. 150 likely due to a typo in the book.

### 2.7.2 Exercise 2

Write a function that, given a set of inputs, computes several fertility indicators following the ideas presented by **?** in Boxes 5.1 and 5.5.

Your function could start as follows:

```
FertFun <- function(W, B, Bf = rep(NA, length(B)), L,
                    srb = 1.05, n = 5, l0 = 10^5){
#[...]
}
```

Here you can see the inputs, which are:

- mid-year population of women: `W`
- number of total births during the year: `B`
- number of female births during the year: `Bf`. In this case, the default is a series of `NA` values of suitable length, assuming the absence of this information.
- lifetable exposure ($L_x$) for females: `L`
- sex ratio at birth: `srb`. By default equal to 1.05 for all ages. This would be useful to internally compute the number of female births when this information is not available
- the width of age-intervals: `n`. By default 5-year-wide age intervals
- the radix of the life table used: `l0`. By default equal to $10^5$ value.

Internally this function should compute:

- age-specific fertility rates: $F_x = \frac{B_x}{W_x}$
- total fertility rate: $TFR = n \cdot \sum F_x$, where $n$ is the length of the interval
- age-specific maternity rate: $F_x^F = \frac{B_x^F}{W_x}$.
- gross reproduction rate: $GRR = n \cdot \sum F_x^F$
- net reproduction rate: $GRR = \frac{1}{l_0} \cdot \sum F_x^F \cdot L_x$

Hint: you could test the availability of information on the fraction/number of female births with the following conditional execution: `if(is.na(Bf[1]))`

Remember that sex ratio at birth for mothers of a given age, $x$, is computed as:

$$srb_x = \frac{B_x^M}{B_x},$$

where $B_x^M$ are the number of male births to females age $x$. And that, in this exercise, you are only asked to assume the same *srb* for all ages.

Test your function with the following datasets: `USAfert1991.txt`, `USAfert1992.txt` and `USAfert2000.txt`. These data contain ages, mid-year female population for USA in 1991, 1992, and 2000, as well as the number of total births during the year. Moreover information about $L_x$ from the life table for females is given. Only for 1991 is number of female births provided.

Read your data as follows:

```
usa91 <- read.table("USAfert1991.txt", header = TRUE)
usa92 <- read.table("USAfert1992.txt", header = TRUE)
usa00 <- read.table("USAfert2000.txt", header = TRUE)
```

Assign the outcomes of the function to objects `fert91`, `fert92` and `fert00` for the 3 datasets, respectively. Check that your function produces the following results:

```
fert91

## $data
##          W       B     Bf         asfr          FxF
## 1  8620000   12014   5816 0.0013937355 0.0006747100
## 2  8371000  519577 253979 0.0620686895 0.0303403417
## 3  9419000 1089692 532712 0.1156908377 0.0565571717
## 4 10325000 1219965 596823 0.1181564165 0.0578036804
## 5 11125000  884862 431694 0.0795381573 0.0388039551
## 6 10344000  330993 162005 0.0319985499 0.0156617363
## 7  9496000   52095  25531 0.0054859941 0.0026886057
## 8  8188000    1709    829 0.0002087201 0.0001012457
##
## $TFR
## [1] 2.072706
##
## $GRR
## [1] 1.013157
##
## $NRR
## [1] 0.9952671

fert92

## $data
##          W       B          Bf         asfr          FxF
## 1  8831206   12220   5960.9756 0.0013837295 0.0006749900
## 2  8324273  505415 246543.9024 0.0607158127 0.0296174696
## 3  9344413 1070490 522190.2439 0.1145593629 0.0558826161
## 4 10047198 1179264 575250.7317 0.1173724256 0.0572548418
## 5 11165144  895271 436717.5610 0.0801844562 0.0391143689
## 6 10619275  344644 168119.0244 0.0324545696 0.0158314974
## 7  9519450   55702  27171.7073 0.0058513885 0.0028543358
## 8  7820172    2008    979.5122 0.0002567718 0.0001252546
##
## $TFR
## [1] 2.063893
##
## $GRR
## [1] 1.006777
##
## $NRR
## [1] 0.989555

fert00

## $data
##          W      B           Bf         asfr          FxF
## 1  1990582    242    118.04878 1.215725e-04 5.930366e-05
## 2  1967944   1427    696.09756 7.251222e-04 3.537182e-04
## 3  1975749   6850   3341.46341 3.467039e-03 1.691239e-03
## 4  1966356  21845  10656.09756 1.110938e-02 5.419210e-03
## 5  1937862  48581  23698.04878 2.506938e-02 1.222896e-02
## 6  1967052  86783  42333.17073 4.411830e-02 2.152112e-02
## 7  1962228 132786  64773.65854 6.767105e-02 3.301027e-02
## 8  2010232 178995  87314.63415 8.904195e-02 4.343510e-02
## 9  1989571 199510  97321.95122 1.002779e-01 4.891604e-02
## 10 1920574 202208  98638.04878 1.052852e-01 5.135862e-02
## 11 1853481 206182 100576.58537 1.112404e-01 5.426363e-02
## 12 1822935 206781 100868.78049 1.134330e-01 5.533317e-02
## 13 1784270 203125  99085.36585 1.138421e-01 5.553272e-02
## 14 1819779 208150 101536.58537 1.143820e-01 5.579609e-02
## 15 1809080 207956 101441.95122 1.149513e-01 5.607378e-02
## 16 1860643 214286 104529.75610 1.151677e-01 5.617937e-02
## 17 1947274 222130 108356.09756 1.140723e-01 5.564500e-02
## 18 2090972 235025 114646.34146 1.123999e-01 5.482921e-02
## 19 2074953 225938 110213.65854 1.088882e-01 5.311622e-02
## 20 2050623 206367 100666.82927 1.006362e-01 4.909084e-02
## 21 1979918 184174  89840.97561 9.302102e-02 4.537611e-02
## 22 2019283 165473  80718.53659 8.194642e-02 3.997386e-02
## 23 2074019 147326  71866.34146 7.103406e-02 3.465076e-02
## 24 2193370 131256  64027.31707 5.984216e-02 2.919130e-02
## 25 2267229 111446  54363.90244 4.915515e-02 2.397812e-02
## 26 2288339  88672  43254.63415 3.874950e-02 1.890220e-02
## 27 2271275  68721  33522.43902 3.025657e-02 1.475930e-02
## 28 2337368  51962  25347.31707 2.223098e-02 1.084438e-02
## 29 2311220  37255  18173.17073 1.611920e-02 7.863022e-03
## 30 2313002  24465  11934.14634 1.057716e-02 5.159593e-03
## 31 2271413  14852   7244.87805 6.538662e-03 3.189591e-03
## 32 2251570   8745   4265.85366 3.883956e-03 1.894613e-03
## 33 2198987   4696   2290.73171 2.135529e-03 1.041721e-03
## 34 2162568   2281   1112.68293 1.054764e-03 5.145192e-04
## 35 2118296   1135    553.65854 5.358080e-04 2.613697e-04
## 36 2054474    527    257.07317 2.565133e-04 1.251284e-04
## 37 1965555    250    121.95122 1.271906e-04 6.204418e-05
## 38 1950309    156     76.09756 7.998733e-05 3.901821e-05
##
## $TFR
## [1] 2.053445
##
## $GRR
## [1] 1.00168
##
## $NRR
## [1] 0.986658
```

# Bibliography

Agresti, A. and B. Finlay (1997). *Statistical methods for the social sciences* (3 ed.). Prentice Hall.

Coale, A. J. (1957). A New Method for Calculating Lotka's r–the Intrinsic Rate of Growth in a Stable Population. *Population studies 11*(1), 92–94.

Dijkstra, E. (1996). Elegance and effective reasoning. course syllabus, Fall 1996.

Human Mortality Database (2015). University of California, Berkeley (USA) and Max Planck Institute for Demographic Research (Germany). Available at www.mortality.org or www.humanmortality.de (data downloaded on July 10, 2014).

Kitagawa, E. M. (1955). Components of a difference between two rates*. *Journal of the American Statistical Association 50*(272), 1168–1194.

Lotka, A. J. (1939). *Théorie analytique des associations biologiques*. Hermann.

Preston, S. H., P. Heuveline, and M. Guillot (2001). *Demography: Measuring and modeling population processes*. Oxford: Blackwell.