R Programming for Demographers

Control Structures

Tim Riffe
riffe@demogr.mpg.de

September, 2016

Contents

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.

2 Flow-control

2.1 Conditional execution

Branching via `if`

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  0.1871585  0.1267636 FALSE
## 2 -1.5811115  0.8643044 FALSE
## 3  0.9548997 -0.6259445 FALSE
## 4  1.4041418  0.6913412 FALSE
## 5  0.1854475  3.0745918 FALSE
```

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.

2.2 Repetitive execution

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

`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
```

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.

`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

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

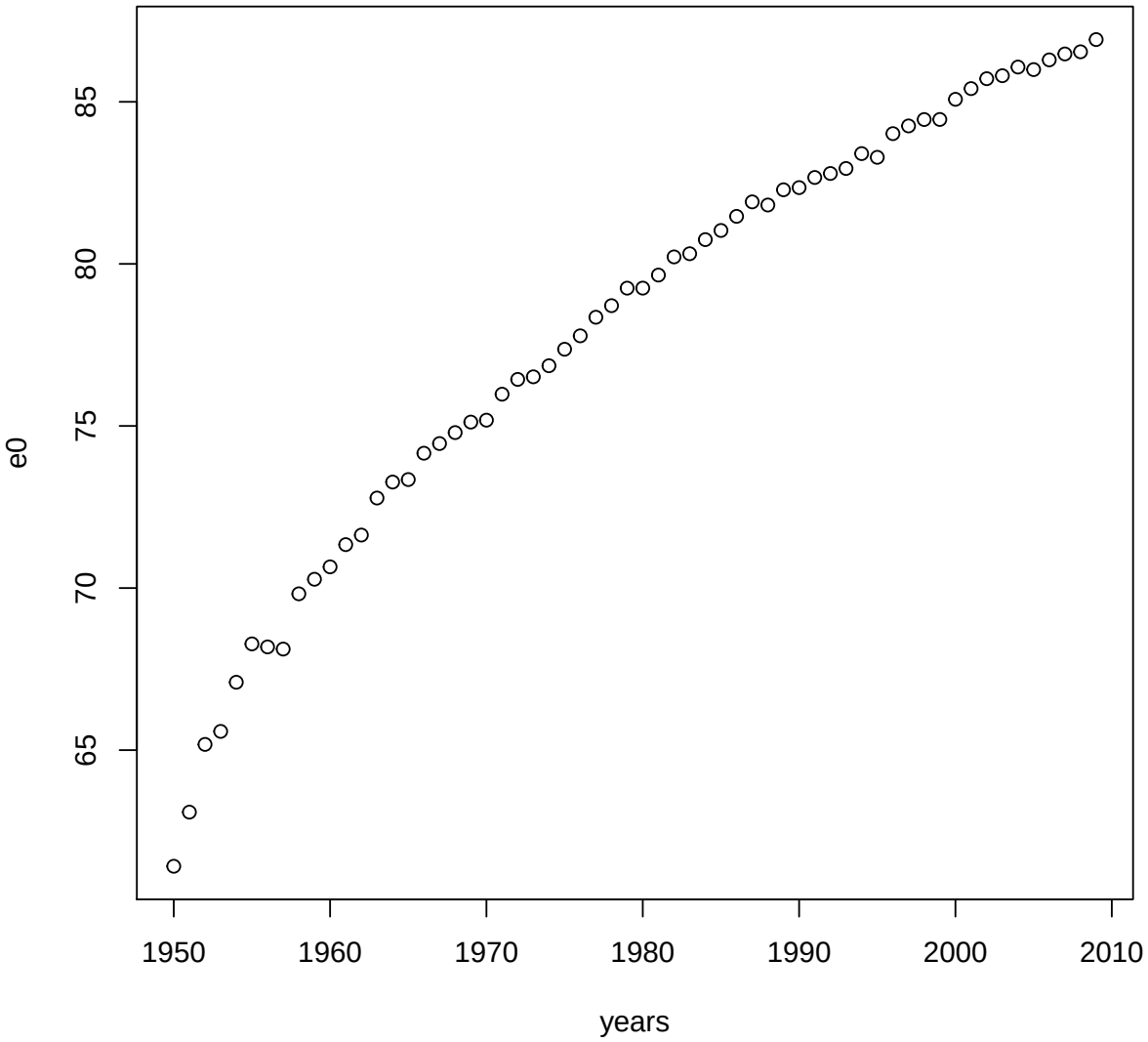

Figure 1: Life expectancy at birth from 1950 to 2009 for Japanese women

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.

```
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.

3 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]

3.1 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"
```

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

```
    }
}
```

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,] "128"       "OK"
##  [2,] "133"       "Critical"
##  [3,] "134"       "Critical"
##  [4,] "124"       "OK"
##  [5,] "132"       "Critical"
##  [6,] "127"       "OK"
##  [7,] "138"       "Critical"
##  [8,] "109"       "OK"
##  [9,] "137"       "Critical"
## [10,] "128"       "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] FALSE

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

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

3.2 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].

[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)

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

##  [1] -0.383088679 -0.121953755 -0.193809379  0.215445955  0.180605629
##  [6]  0.136753328  0.356822276  0.140952450 -0.004223347 -0.030127789
```

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] -0.383088679 -0.121953755 -0.193809379  0.215445955  0.180605629
##  [6]  0.136753328  0.356822276  0.140952450 -0.004223347 -0.030127789
```

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] -0.383088679 -0.121953755 -0.193809379  0.215445955  0.180605629
##  [6]  0.136753328  0.356822276  0.140952450 -0.004223347 -0.030127789
```

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]  0.17486948  0.36513584  0.12483647 -0.14523804  0.26393492
##  [6] -0.16599366  0.55824518 -0.58244011  0.05169412  0.20277309
```

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:

[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

```
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] 0.4356350 0.5459728

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

## [1] 0.6195308 0.6365067 0.3855781 0.3959438 0.4300772 0.6098404 0.6352871
## [8] 0.2136669

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

## [1] 0.5594103 0.4408845 0.4721168
```

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 2):

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

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

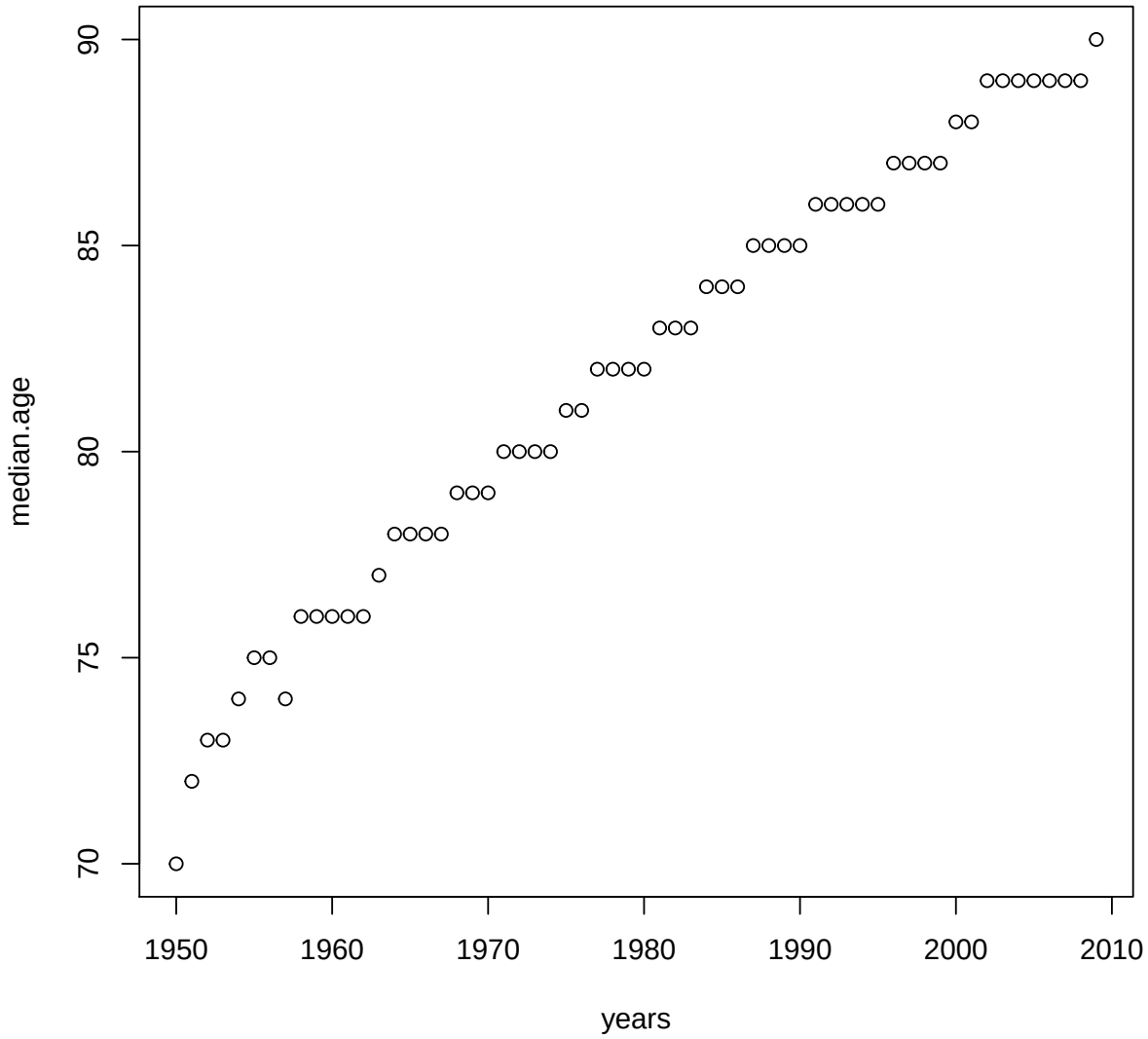

Figure 2: Median age at death from 1950 to 2009 for Japanese women

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%
##  3.302294  8.287191  9.489665 10.439150 11.625010 16.257452
##
## $b
##         0%        20%        40%        60%        80%       100%
## -0.5310945  5.8971966  8.0785464  9.5156446 12.1479269 19.1869923
##
## $c
##         0%        20%        40%        60%        80%       100%
## -17.760979   4.876961   8.470751  11.549395  15.107963  32.597800
```

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 α and β 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%    3.302294 -0.5310945 -17.760979
## 20%   8.287191  5.8971966   4.876961
## 40%   9.489665  8.0785464   8.470751
## 60%  10.439150  9.5156446  11.549395
## 80%  11.625010 12.1479269  15.107963
## 100% 16.257452 19.1869923  32.597800

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.

4 Exercises

4.1 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 $m(x,y) = \frac{D(x,y)}{N(x,y) - 0.5 \cdot D(x,y)}$

4.2 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, 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.

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

References

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).

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

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