R Workshop – Walkthrough

January 2026

This file contains a series of sections that introduce various aspects of R from basic program syntax to advanced data analysis and modeling using matrix algebra or optimization. The topics are tailored slightly in favor of the practitioners of econometrics than operations research. The topics assume basic knowledge of regression analysis and matrix algebra. Questions, corrections or comments can be sent to the instructor at tungakantarci@outlook.com.

1. Installing R

R is free. It can be downloaded from https://cran.r-project.org. The user interface of R is not user friendly, however. RStudio is developed to address this. RStudio is just an Integrated Development Environment (IDE) in which you are using the R language. RStudio can be downloaded from https://rstudio.com/products/rstudio/. This workshop makes use of RStudio. Download and install RStudio on your computer.

2. Program Window

The program window consists of four panels. 'Console' is where you type command syntax. Type, e.g., `a = 1` in the command prompt, which creates a scalar named `a` that takes a value of `1`. Type `a` in the command prompt, which will display the result in the Console.

R has added the scalar you just created to the 'Environment'. Environment shows the variables you create and held in memory during a R session. It is currently displaying the scalar variable `a`.

Type `file.edit('untitled.R')` in the command prompt. This will open the 'Editor'. Editor is a simple text editor that allows to type and keep program syntax.

'File Browser' is where you can browse, delete or rename R files. E.g., click on the 'Go to directory' button (···) located at the right hand side of the File Browser, search for the Desktop folder where the R program file `RWScript.R` is located. Click 'Open'. Note that you are opening a folder, not a file. Do not search for `RWScript.R`, but for the folder that contains `RWScript.R`. After you click 'Open', `RWScript.R` will appear in the File Browser. Click on it, to open it in the Editor. We will learn about `RWScript.R` in a future section.

```
a = 1
file.edit('untitled.R')
```

3. Interacting with R

You can interact with R in three ways. In the first way, you use the program menu, e.g., to create R program files, open R data files, etc. What you can do through the program menu is very limited, however.

In the second way, you type commands in the command prompt, in the Console. For this you need to know command syntax. In this workshop you will learn some standard command syntax. You can check the R documentation for special command syntax. You will learn about the R documentation in a future section.

R keeps a history of the commands you have typed in the current or a previous session in 'Command History'. To view the Command History, activate your cursor in the Console, and use the up- and down-arrow keys to browse the command history. In this way, you can adjust a previously executed command without retyping it all over again. You can also find a list of all the previously executed commands under the 'History' tab in the Environment pane.

In the third way, you interact through a 'script' file which is a simple text editor where you can type and execute commands.

How do we execute a command contained in a script file? Go to the Editor, and have the script file `RWScript.R` on display. Go to Section 3 in the script file, and find the command statement `a = 2`. There are two ways to execute this command. In the first way you use the

mouse. Place the mouse cursor to the left of the command, and click on the 'Run' button located at the right hand side of the Editor. The second, and for many users the more practical way is to use the keyboard. Use the up- and down-arrow keys on the keyboard to place the cursor to the left of the command `a = 2`. Hit together the 'control' and 'return' keys on a Windows or a OSX operating system.

```
a = 2
```

4. Creating a Script File

A script file is a simple text file that contains command syntax exactly as you would type them in the command prompt. It takes the file extension .R. Before creating a script file, you need to specify the directory where you will keep your script file. R has a default working directory but often we want to change it to one of our own preference. Type in the command prompt `setwd("M:/")` to change the current working directory to be the Desktop. If you type `getwd()` in the command prompt, it will show the current working directory.

Type `file.edit('scriptfile.R')` in the command prompt to create a script file named `scriptfile`. This will open the Editor where you can store program syntax. Alternatively, you can use the program menu to create a script file. On the program menu, click on 'File'. In the drop down menu go to 'New File', and choose 'R Script'. This will open the 'Editor' panel, and create a 'script file' with the name 'Untitled1'.

Close the script file you have just created. We will work with the script file created for this workshop. To open it, double click on `RWScript.R` located on the Desktop.

```
setwd("M:/")
getwd()
file.edit('scriptfile.R')
```

5. Program Syntax

R syntax takes two forms; command syntax and function syntax. In its simplest form, a command syntax is a statement. E.g., the statement `rm(list = ls(all = TRUE))` clears the memory from any preloaded program code or data. The statement `5+5` executes an arithmetic operation. If you type `5+5` in the command prompt, R will return `10`.

Often we would want to assign a statement to a variable, to refer to it easily throughout the code, or to use the statement as input in other statements. E.g., you can assign the statement `5+5` to a variable named `a` by typing in the command prompt `a = 5+5`. If you type `a` in the command prompt, R will return `10`. You can then use `a` in a new statement, say in `b = a+1`.

Note that before assigning a statement, or a value, to a variable, you do not have to declare the variable. Also note that if you do not assign a statement to a variable, R will automatically assign it to a variable named `.Last.value`. In fact, this is what happened when you have typed `5+5` above. If you type `.Last.value` in the command prompt, R will return `10`. However, R will overwrite `.Last.value` with every command that returns an output value that is not assigned to a variable. If you would like that `.Last.value` always appears in the Environment, this can be enabled in the program preferences.

R stores created variables in the Environment. To inspect variable `a`, you first need to assign it to a matrix using the syntax `as.matrix(a)`. This will create a matrix in the Environment.

To inspect it, click on the icon that symbolises a spreadsheet view located next to the variable `a` in the Environment.

Function syntax, as its name suggests, is used when using built-in R functions. Functions take input arguments and return output. You enclose input arguments in parentheses, and separate them with commas. E.g., you could use the `round` function to round the mathematical pi constant to the nearest 3 decimal digits using the function syntax `round(pi,3)` which will return `3.142` as output.

You can suppress output produced by a command and printed in the Console by using the `invisible` function. E.g., `invisible(round(pi,3))` will suppress the output `3.142` in the Console.

R is case sensitive meaning that lowercase and uppercase letters have different meanings. E.g., `Pi` is not an alternative for `pi`.

```
rm(list = ls(all = TRUE))
5+5
a = 5+5
a
b = a+1
b
a = as.matrix(a)
round(pi,3)
invisible(round(pi,3))
```

6. Creating, Indexing, and Searching Objects

In R environment, there are different types of data structures. The most common types of data structures are 'vector', 'matrix', 'list', and 'data frame'. The most basic data data structure is the vector. Other data structures are built on this atomic data structure. A data structure is often called an 'object'. From now on we will also use the term object to refer to a data structure. You will learn about the different types of objects throughout this workshop.

A vector object holds data. It can hold different types of data. The most common types of data are numeric (1), character (Tilburg), and logical (TRUE, FALSE).

You can determine the type of an object using the `class` function. You can determine the type of the data an object holds using the `typeof` function. However, if the object is a vector, the `class` function will not return the type of the object, but the type of the data. For example, type `d = 5` to create a vector object. The type of the data this vector holds is 'numeric'. If you type `class(d)`, R will not return the type of the object as 'vector', but it will return the type of the data this vector holds, which is 'numeric'.

Create a vector object by typing `d = c(2,7,5,1,8,9,0,1,1)` in the command prompt. `c()` is a generic function which combines its arguments to form a vector. In R, a vector object does not have a row or column dimension but only a length. To determine the length of the vector, type `length(d)` in the command prompt.

To take the transpose of a vector, use the transpose operator `t()`. Type `d = t(d)` in the command prompt to obtain the transpose of the vector `d`. If you type `d` in the command prompt, in the Console, R will print the vector as a row vector. You can take the transpose of `d` for a second time if you like `d` to be a column vector. However, in R, it usually makes little difference whether a vector is a row or column vector.

To access selected elements of an object, we use 'indexing'. To access the second element

of the vector object you just created, type `d[1,2]` in the command prompt. `1` is the 'row subscript', and `2` is the 'column subscript'. Since the vector object `d` is a row vector, we could suppress the row subscript. That is, `d[2]` is equivalent to `d[1,2]`. To access the first three elements of the vector, type `d[1:3]`. `:` is the 'colon operator' which allows you to specify a range of the form `start:end`. Hence, `1:3` selects all the elements from the first to the third element in the vector object.

To create an equally spaced vector of values, we use the `seq` function. E.g., `d = seq(0,10,2)` creates a vector array with values increasing from 0 to 10 by an increment of 2.

To search for certain elements in a vector, and for their position in the vector, we can use the `which` function. Execute once again the syntax `d = c(2,7,5,1,8,9,0,1,1)`. Let us investigate if `d` contains the value 1, and its position in `d`. We could use the function syntax `which(d == 1)`. The function accepts several input arguments. Here we specify only one argument. The argument specifies the vector where the search will take place, and the value to be searched for. We require that the search takes place in vector `d`, and that we search for the value 1. Execute the function syntax. The function returns three values as output. The values represent vector indices. Indices refer to the position of the elements we are searching for in the vector. The values we obtain indicate where the value 1 is located in the vector. To learn more about the `which` function and the input arguments it accepts, type `help(which)` in the command prompt.

```
d = 5
class(d)
d = c(2,7,5,1,8,9,0,1,1)
d
d = t(d)
d
length(d)
d[1,2]
d[1:3]
d = seq(0,10,2)
d
d = c(2,7,5,1,8,9,0,1,1)
d
which(d == 1)
```

7. Commenting, Breaking, and Masking Syntax in a Script File

There are two ways to include comments in a script file. First, you can begin a line with the hash symbol, `#`, and everything you type after the hash symbol to the end of the current line is considered a comment. Second, following a command syntax, you can start with the hash symbol, and everything you type after the hash symbol to the end of the current line is considered a comment. See below examples for each type of syntax commenting.

In your script file you may need to type a line of code that is wider than the width of the script file editor. You can simply type the long line of code. However, if you want the code to stay within the limits of the script file editor, you can split the code across multiple lines just by breaking the code over multiple lines, but the break should occur after, say, an open bracket or an operator so that as you execute a line of the code, R will expect the next line. Experiment with the example below.

You may want to mask code if you want R to ignore it. You can mask a single line of code

simply by placing a hash symbol in front of the code as if you were commenting syntax. You can mask several lines of code in the same manner but this can become tedious if the code occupies many lines. In that case you can comment out the whole block of code by first selecting it, and then by choosing 'Comment Lines' from the 'Code' tab in the program menu. See the examples below.

```
# You can add a line of comment.
a = 5+5 # You can annotate code.
# You can split a line of code across multiple lines.  E.g.:
a =
5+5
# You can mask multiple lines of code.  E.g.:
# a = 5+5
# b = a+1
# You can mask a single line of code.  E.g.:
# a = 5+5
```

8. Importing Raw Data Files

Data files hardly come in a format that can be read directly by the software of choice. Often they come in raw text format. One such raw data format is the 'comma separated value' (csv) format. Inspect the content of the csv file we will use. To open the csv file, right click on the csv file located on the Desktop, in the menu that appears choose Notepad++, which is a simple text editor. Observe that the variable names are stored in the first row, the values are stored beneath the variable names, both are separated from each other with a comma (,), and a period (.) acts as a decimal point.

This section explains how to import a csv file into R, and to save it in a data format native to R. Before you proceed, clear the memory by typing `rm(list = ls())` in the command prompt. We start by defining several objects which we will use as input arguments in the function we will consider to import data eventually. First, we define the directory where the raw data file is located, alongside the name of the raw data file. We can do this using the statement `M:/RW csv file.csv`. However, we need to enclose the statement in quotes, as `"M:/RW csv file.csv"`, because the statement is a string entry rather than a numeric entry. Next, assign the statement that defines the directory to a variable named `filename`. We can now define our first input argument as `filename = "M:/RW csv file.csv"`.

In the raw data file, the names of the variables are stored in the top row, and the values of the variables are stored in the rows beneath. In the rows of the data file, the names and the values of the variables are separated by a delimiter where the delimiter is a comma (`,`). We need to define the delimiter. We should do this by enclosing the delimiter in quotes, as `","`, because the delimiter is a string entry. Let us assign the delimiter to an object named `sep`. We can now define our second input argument as `sep = ","`.

Raw data files usually come with variable names stored in the first row of the data file. This is also the case with the csv file at hand. We need to declare that this is the case. This is done by assigning the logical object `TRUE` to an object named `header` by typing `header = TRUE` in the command prompt. This defines our third input argument.

In the raw data file, a period acts as a decimal point. We need to define the decimal separator. `dec = "."` assigns the period to an object named `dec`.

We are now ready to use the `read.csv` function that allows to import the csv file into

R. The function accepts the four objects created above as input arguments, and returns data in `RData` format native to R as output. In particular, consider the command `rwrdatafile = read.csv(filename,header = TRUE,sep = ",",dec = ".")`. `rwrdatafile` is a name we give to the output that the `read.csv` function will return. The command in whole instructs R to read the data from `filename`, with the delimiter being a comma, where it is true that the raw data file includes a header line, and with the decimal separator being a period. The command will return a 'data.frame' object named as `rwrdatafile` that contains the imported data that is now in `RData` format. We will learn about the data.frame in a future section.

We would want to save the imported data, currently held in the system memory, to our hard drive. Note, however, that the Environment contains several objects, and we want to save in a file only the data.frame that contains the imported data, and exclude all other objects. We can use the `save` function to achieve this. The function syntax is `save(rwrdatafile,file = "M:/RW rdata file.RData")`, which instructs R to select the object `rwrdatafile`, and save it in the file `RW rdata file`.

```
rm(list = ls())
filename = "M:/RW csv file.csv"
rwrdatafile = read.csv(filename,header = TRUE,sep = ",",dec = ".")
save(rwrdatafile,file = "M:/RW rdata file.RData")
```

9. Opening Data Files

Remove items from the Environment using the command `rm(list = ls())`. Suppose that you just started a new session and want to open a RData file, say the RData file you just created in the preceding section. You can use the `load` command to achieve this. That is, you can type `load("M:/RW rdata file.RData")` in the command prompt to open the RData file.

```
rm(list = ls())
load("M:/RW rdata file.RData")
```

10. Browsing the Data

A 'data frame' is an object type that groups data. It is used as the fundamental data structure by most functions in R. It can contain data of any type or size. We can refer to the data in a data frame using 'indexing', as we will study below. For more information on the data frame, type `help("data.frame")` in the command prompt.

`rwrdatafile` is a data frame contained in the RData file `RW rdata file.RData`, and loaded into the system memory in the preceding section. In the Environment, click on this data frame to access the data it contains. This will open the data browser where you can inspect the data. It shows a collection of vectors, and indicates a name for each vector.

Suppose that we want to browse the first 20 observations of the data frame `rwrdatafile`. For this we can use the function syntax `rwrdatafile[1:20,]`. `rwrdatafile` refers to the data frame we want to consider. We can access the observations of interest using indexing. The row subscript `1:20` selects all the rows from the first to the twentieth row of the data frame. Leaving the column subscript blank selects all the columns from the first to the last column of the data frame.

If we wanted to browse only the variable `wage` contained in the first column of the data

frame, we could use the syntax `rwrdatafile[,1,drop = FALSE]`.

We could browse the data using a logical condition. Suppose that we want to browse the data collected among women. Type `unique(rwrdatafile[,4])` to inspect the unique values the vector `rwrdatafile[,4]` contains. 1 represents women. We could consider the command syntax `rwrdatafile[rwrdatafile[,4] == 1,c(1,2,4)]` to select our sample. Pay attention to the row subscript: `rwrdatafile[,4] == 1`. Leaving the row subscript blank selects the observations contained in all rows of the data frame, and using a value of 4 for the column subscript selects the observations contained in the fourth column of the data frame. Hence `[,4]` selects all the observations in the fourth column of the data frame. `== 1` requires that these observations are for women. Consider the column subscript: `c(1 2 4)`. It selects the vectors `wage`, `educ`, and `female`. The two arguments together selects the sample.

Suppose that we want to browse the data collected among women, as we did above, but only in the first 20 observations of the vector `rwrdatafile[,4]`. It is tempting to use a syntax like `rwrdatafile[rwrdatafile[1:20,4] == 1,c(1,2,4)]` in analogy to the one used above. However, this will not work. The reason is that the subsetting mechanism for data frames in R needs a vector of the same length as the data frame has rows. Instead, we can use the syntax `rwrdatafile[which(rwrdatafile[1:20,4] == 1),c(1,2,4)]`. The syntax is similar to the one we used above except that the row subscript, `which(rwrdatafile[1:20,4] == 1)`, now searches for women in the first 20 rows of the vector `rwrdatafile[,4]`.

We might want to inspect a variable with its observations sorted in an ascending or descending order. To make it a little more complicated, suppose that we want to sort the observations of `wage` in a descending order 'within' the observations of `educ` sorted in an ascending order. The syntax `rwrdatafile[order(rwrdatafile[,2],-rwrdatafile[,1]),]` serves to this purpose. Pay attention in particular to the first input argument of the `order` function: `(rwrdatafile[,2],-rwrdatafile[,1])`. It requires the second column vector be sorted in an ascending order, and 'within' the sorted observations of the second column vector, it requires the first column be sorted in a descending order. Inspect the result stored in `.Last.value` in the Environment.

Note that the syntax `rwrdatafile[order(rwrdatafile[,2],-rwrdatafile[,1]),]` does not replace the original sorting of the data contained in the data frame `rwrdatafile`. That is, after sorting the data using `rwrdatafile[order(rwrdatafile[,2],-rwrdatafile[,1]),]`, if you inspect the data frame `rwrdatafile`, you will notice that the data is not sorted. If you want the sorting to apply to the data frame `rwrdatafile`, you need to 'assign' the sorting statement `rwrdatafile[order(rwrdatafile[,2]),]` to the data frame name `rwrdatafile` using the command `rwrdatafile = rwrdatafile[order(rwrdatafile[,2],-rwrdatafile[,1]),]`.

We could obtain the sizes of each dimension of the data frame `rwrdatafile` using the function syntax `N = nrow(rwrdatafile)` and `k = ncol(rwrdatafile)`. `nrow` and `ncol` are built-in functions that accept a data frame as input, and return, respectively, the sizes of the row and column dimensions of the data frame as numeric objects as output.

```
rwrdatafile[1:20,]
rwrdatafile[,1,drop = FALSE]
unique(rwrdatafile[,4])
rwrdatafile[rwrdatafile[1:20,4] == 1,c(1,2,4)]
rwrdatafile[which(rwrdatafile[1:20,4] == 1),c(1,2,4)]
rwrdatafile[order(rwrdatafile[,2],-rwrdatafile[,1]),]
rwrdatafile = rwrdatafile[order(rwrdatafile[,2],-rwrdatafile[,1]),]
N = nrow(rwrdatafile)
k = ncol(rwrdatafile)
```

11. Producing Descriptive Statistics

Built-in R functions can be used to produce descriptive statistics. To compute the mean of the elements in each column of the data frame `rwrdatafile`, we can use the function syntax `colMeans(rwrdatafile)`. `colMeans` is a function that accepts a data frame, and calculates the mean in each column of the data frame. Therefore the function returns a row vector containing the mean of the elements in each column of the data frame. If we have used the function `rowMeans`, we would have obtained a column vector containing the mean of the elements in each row of the data frame.

Relational operators can be used in statistical functions to obtain group statistics. E.g., `mean(rwrdatafile[rwrdatafile[,4] == 1,1])` produces the mean of wage among women, using the relational 'Equal to' operator (`==`). Convince yourself that this is true.

To produce a frequency table for `educ`, we need to use a number of built-in functions. Consider the syntax `cbind(table(rwrdatafile[,2]),prop.table(table(rwrdatafile[,2])))`. The `table` function produces a frequency count of the data specified. The `prop.table` function returns the fraction of unique values contained in a vector object. `cbind` is a function that combines objects by columns. In the example, it combines the output produced by the `table` and `prop.table` functions by columns.

In econometrics, we are hardly interested in the statistics of a single variable, but in the statistical relationships between variables. The `cor.test` function can be used for this. In particular, `cor.test(rwrdatafile[,1],rwrdatafile[,3])` accepts column vectors `wage` and `exper` as input, and returns, among other things, the correlation coefficient as well as a p-value for testing the hypothesis of no correlation as output.

Note that when you execute the function syntax `colMeans(rwrdatafile)`, it returns a row vector containing the value `NaN` as one of its elements. `NaN` stands for Not-a-Number. It results from operations which have undefined numerical results. The `coLmeans` function generated a `NaN` value because in its second column the data frame contains missing observations. Browse to the end of the data frame to notice the missing observations marked by `NaN` values. To ignore the `NaN` values and compute the mean of this vector, you can add `na.rm = TRUE` as a second input argument in the `colMeans` function. That is, you can consider the function syntax `colMeans(rwrdatafile,na.rm = TRUE)`. `na.rm` is a built-in argument of the `colMeans` function and takes a logical value. If the logical value is `TRUE`, it instructs the `colMeans` function to omit the missing values from the calculation.

To count the number of missing cases in the vector array `educ`, you can use the `summary` function. That is, the command `summary(rwrdatafile[,2])` returns the number of `NaN` values in the vector array `educ`. You can use the `is.nan` function within the `which` function to determine the row indices that contain the `NaN` values in the second column of the data frame. Consider the function syntax `which(is.nan(rwrdatafile[,2]))` and convince yourself that it returns what we want it to return.

```
colMeans(rwrdatafile)
mean(rwrdatafile[rwrdatafile[,4] == 1,1])
cbind(table(rwrdatafile[,2]),prop.table(table(rwrdatafile[,2])))
cor.test(rwrdatafile[,1],rwrdatafile[,3])
colMeans(rwrdatafile,na.rm = TRUE)
summary(rwrdatafile[,2])
which(is.nan(rwrdatafile[,2]))
```

12. Creating Variables

In the preceding sections, we have accessed data stored in a data frame using indexing. E.g., we have accessed wage data using the syntax `rwrdatafile[,1]`. This way of accessing vector objects is a little tedious unless we want to access certain rows or columns of the vector objects using indexing. Instead, we can assign vector objects to variables, and access vector objects using variable names. E.g., `wage = rwrdatafile[,1]` assigns the vector object `rwrdatafile[,1]` to a variable named `wage`.

We can generate new variables using arithmetic, logical, and relational operators. Consider the following commands that generate new variables using arithmetic operators: `wage+1`, `-wage`, and `wage^2`.

The command `wage >= 20 & wage <= 40` creates a variable using the logical 'and' operator (`&`), but also the relational 'greater than or equal to', and the 'less than or equal to' operators (`>=`, `<=`). It creates in particular a logical variable that takes the logical value 'TRUE' if the condition we impose is true, and 'FALSE' otherwise. We would want to convert this logical variable to a numeric variable to obtain a dummy variable. For this we can use the `as.numeric` function. In particular, `as.numeric(wage >= 20 & wage <= 40)` creates a dummy variable taking a value of 1 when `wage` is between 20 and 40, and 0 otherwise. To browse the values of this variable, you can convert it to a data frame using the command `as.matrix(.Last.value)`.

We can use the built-in mathematical functions to transform variables. E.g., `log(wage)` takes the natural logarithm of `wage`, or `exp(wage)` performs the exponential transformation.

```
wage = rwrdatafile[,1]
wage+1
-wage
wage^2
wage >= 20 & wage <= 40
as.numeric(wage >= 20 & wage <= 40)
as.matrix(.Last.value)
log(wage)
exp(wage)
```

13. Manipulating Variables

To keep certain variables and remove all others in the Environment, use the `rm` function. E.g., `rm(wage)` removes `wage` but keeps other objects in the Environment.

To delete a column from a data frame, use the minus sign in data frame indexing. E.g., to delete column number five in the data frame `rwrdatafile`, use the command `rwrdatafile[,-5]`.

The vector `educ` in the matrix `rwrdatafile` contains observations with missing values marked by `NaN` values. We may want to delete observations with missing values, but also the observations with non-missing values contained in the same row but in other columns of the data frame, because observations with missing values cannot be used in combination with observations wth non-missing values to produce multivariate statistics. Consider the command `rwrdatafile = rwrdatafile[!is.nan(rwrdatafile[,2]),]`. Consider the row subscript of the data frame `rwrdatafile` which is `!is.nan(rwrdatafile[,2])`. `is.nan(rwrdatafile[,2])` selects the rows of `educ` that contain `NaN` values, and returns a logical value which is either TRUE

or FALSE. `!` is the logical 'not' operator. It negates the logical values returned by the `is.nan` function. Consider the column subscript of the data frame `rwrdatafile` which is left blank. It selects all the columns of the data frame. The command in whole deletes the rows that contain `NaN` values.

The `wage` data contains values larger than 30, and suppose we believe that these are outliers and want to replace them with missing values. Convince yourself that the command `rwrdatafile[rwrdatafile[,1] > 30,1] = NaN` replaces values of `wage` larger than 30 with NaN values.

Type `sapply(rwrdatafile,class)` in the command prompt. `sapply` is a function that applies a user specified function over the multiple vectors of a data frame. In the example, it applies the `class` function on each vector of the data frame `rwrdatafile`. `class` is a function that returns the type of the specified object. `sapply(rwrdatafile,class)` returns as output the data type of each vector in the data frame `rwrdatafile`. The output shows that the data in `rwrdatafile` is of numeric type. However, data can come in different formats. E.g., `female` in `rwrdatafile` could hold character data such as 'female' and 'male'. To work with such a character object, we would have needed to convert it to numeric array. Built-in functions can convert one data type into another. We do not pursue this here, however.

```
rm(wage)
rwrdatafile[,-5]
rwrdatafile = rwrdatafile[!is.nan(rwrdatafile[,2]),]
rwrdatafile[rwrdatafile[,1] > 30,1] = NaN
sapply(rwrdatafile,class)
```

14. Scalar vs Vector Oriented Coding

In R, or in comparable software like MATLAB, we can carry out matrix operations in two ways. In the first way we work with the elements of a vector and use loops to carry out matrix operations on each element in an iterative process. In the second way we work with all the elements of a vector at a time when carrying out matrix operations. Vectorised code is easier to understand, less prone to errors, and often runs faster than the corresponding code containing loops. Hence we might want to vectorise our code as much as possible.

Open the data frame `rwrdatafile` from the Environment to inspect its content. Note the sorted values of `educ`. Suppose that we want to delete the rows of the data frame if a unique value of `educ` is ordered first among its kind. E.g., there are six cases where `educ` takes the unique value of 4. We want to delete the rows if it is the first case of 4 among the ordered six cases. We will consider two different programs to delete the rows. The programs are presented at the end of the section.

First, consider the program containing a for loop. A for loop is a program that repeats an operation for a specified number of times. In the code, in first line of the for loop, `i` specifies the 'index' of the loop. `uniq` contains the 'values' the index takes. It contains the unique values of `educ`. A for loop iterates over elements of `uniq`. It does not matter if `uniq` is a column or a row vector. The next line contains the 'statement' of the loop that repeats itself for the index values. In particular, `which(rwrdatafile[,2] == uniq[i])[1]` conducts a search of the row, among the rows of the column vector `educ`, which contains the first case of the multiple cases of a unique element. The indexing operator `[1]` at the end of the `which` function returns the first sequential element of what the `which` function returns. Since we have specified the `which` function as a row subscript of the data frame object `rwrdatafile`, it acts as selecting the row

it conducts the search for. The column subscript which is left blank selects all the columns of the data frame. We use a minus sign for the row subscript to delete the selected rows of the data frame object. The right brace (}) ends the for loop.

Consider the second program containing only vector operations. The `match` function takes the unique elements of `educ` contained in `uniq`, and all elements of `educ` contained in the vector object `rwrdatafile[,2]` as input arguments. The function returns a numeric object as output. In particular, it returns a vector named `tag`, containing the lowest index in `rwrdatafile[,2]` for each row in `uniq` that is also a row in `rwrdatafile[,2]`. That is, `tag` contains the row numbers containing the first instances of the unique values of `educ` among the sorted values of `educ`. A note is the following. Browse the data frame `rwrdatafile`. You will realise that the row numbers are not sorted. This is because the data is sorted with respect to `educ`, in an earlier Section. However, `tag` contains the row numbers of the first instances of the unique values of `educ` after the data is sorted.

`unique` and `match` are functions to carry out set operations which prove useful when you need to access certain values in a data frame.

```
# Scalar-oriented coding
uniq = unique(rwrdatafile[,2])
length = length(uniq)
for (i in 1:length) {
  rwrdatafile = rwrdatafile[-(which(rwrdatafile[,2] == uniq[i])[1]),]
}
# Vector-oriented coding
uniq = unique(rwrdatafile[,2])
tag = match(uniq,rwrdatafile[,2])
rwrdatafile = rwrdatafile[-tag,]
```

15. Assign Names to Vector Arrays

We have learned working with vector objects. We can now put this knowledge in use to analyse economic phenomena. First, let us make our data ready. Clear the memory, reload the RData file containing our data, delete the rows where `educ` takes a value of `NaN` in the matrix array, and obtain the sizes of the row and column dimensions of the matrix array, using the first five lines of the syntax presented at the end of the section.

In our data analysis we will work with vectors, rather than with a matrix. Therefore, we need to extract vectors from our matrix in `rwrdatafile`. In our dataset we have six vectors in total. We could assign each vector to a variable, and access the vectors using variable names. For this we could use, e.g., `wage = rwrdatafile[,1]`. However, it is tedious to assign each vector in the dataset to a variable for six times. Instead, we can write a for loop to automate creating variables. In particular, our aim is to assign vectors to variables stored in `rwrdatafile(,i)` where `i` is a vector. Consider the command `assign(names(rwrdatafile)[i],rwrdatafile[,i])`. `assign` is a function that assigns a value to a name. It accepts two input arguments. The first input argument is a character object, and contains the variable names. In particular, `names` is a function that extracts the name of an object, and `names(rwrdatafile)[i]` extracts the names of all vectors stored in the data frame `rwrdatafile`. The second input argument is a numeric object, and contains the vectors stored in the data frame `rwrdatafile`. Hence, the `assign` function assigns vector `rwrdatafile[,i]` to the name `names(rwrdatafile)[i]`. The for loop repeats the procedure for each `i` from `1` to `k` where `k` is the size of the column dimension of

the data frame we have defined above. Check `help("assign")` to learn more about naming variables.

```
rm(list = ls())
load("M:/RW rdata file.RData")
rwrdatafile = rwrdatafile[!is.nan(rwrdatafile[,2]),]
N = nrow(rwrdatafile)
k = ncol(rwrdatafile)
for (i in 1:k) {
  assign(names(rwrdatafile)[i],rwrdatafile[,i])
}
```

16. Regression Analysis Using Matrix Algebra

Consider the linear regression model with multiple regressors that take the form $y = \mathbf{X}\boldsymbol{\beta} + u$. $\mathbf{X}$ is a N×K matrix. N denotes the number of observations. K denotes the number of vectors in $\mathbf{X}$. There is one vector for the constant term, and k vectors for k regressors. Hence K = 1+k. Each vector is of size N×1. $\boldsymbol{\beta}$ is K×1. It represents the true coefficient vector. The error term is N×1, and follows a normal distribution. Our aim is to estimate the parameter vector $\boldsymbol{\beta}$. You may recall from your introductory econometrics course that solving the FOC of the least squares problem leads to the OLS parameter estimates, $\hat{\boldsymbol{\beta}}$, that take the form $(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$. This means that the matrix operation we need to carry out in R is `solve(t(X)%*%X)%*%t(X)%*%y`. `y` is the dependent variable. In our example it is the `wage`. For notational convenience let us rename `wage` as `y`. `X` is the systematic component of the regression equation. It includes a constant and a set of regressors. Let us denote the constant with `c`. To create the constant term consider the command `c = rep(1,N)`. The function `rep` creates a vector array of all ones where the size of the row dimension of the array is `N`. Let us consider `educ` and `exper` as two regressors. We can create `X` as `X = as.matrix(cbind(c,educ,exper))`. Consider the matrix operation `B_hat = solve(t(X)%*%X)%*%t(X)%*%y` that produces a vector array with OLS estimates of the true parameter vector $\boldsymbol{\beta}$. `solve` is the matrix inverse operator. See `help("solve")` for more information. `t` is the transpose operator. `%*%` is the matrix multiplication operator that multiply two matrices which have a common inner dimension.

```
c = rep(1,N)
y = as.matrix(wage)
X = as.matrix(cbind(c,educ,exper))
B_hat = solve(t(X)%*%X)%*%t(X)%*%y
```

17. Creating Graphs

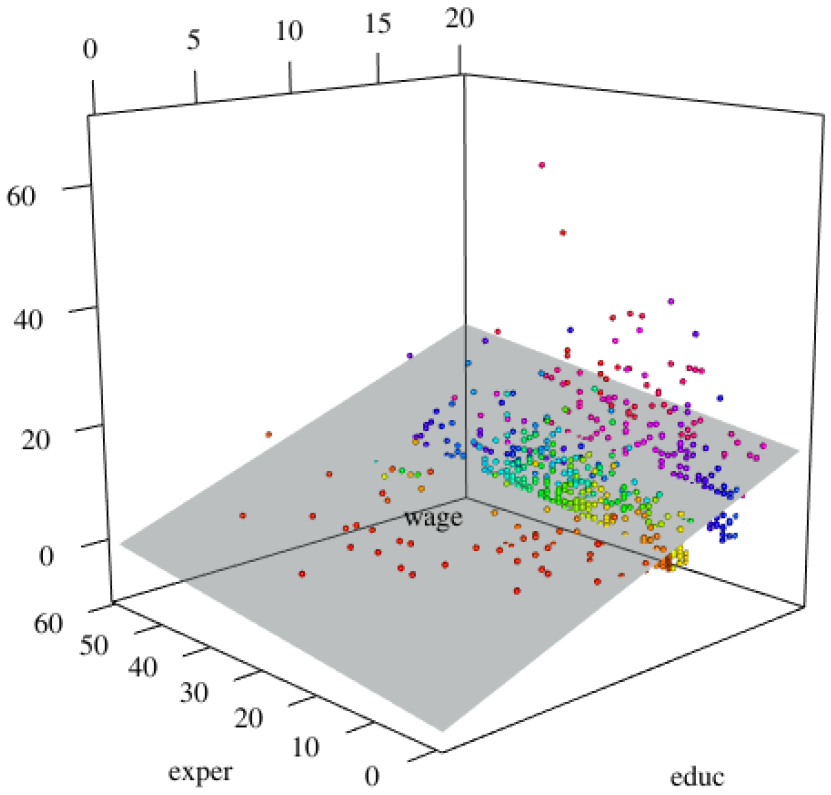
It is helpful to produce a scatter plot to visualise the relationships between the dependent variable and the independent variables to form expectations about the signs and significance of the effects of the independent variables. Here we learn how to produce a three dimensional scatter plot. We will then also combine it with a best-fit plane.

Consider the lines of syntax presented at the end of the section. The first line loads a third party package that allows to produce a three dimensional plot in R. R does not install this package by default, and therefore you need to install it before you can use it in R. In the second

line, we predict wage using the paired observations of educ and exper, and the estimated parameters of the regression model obtained in the preceding section. We make use of the predictions because we want to plot the predictions against the explanatory variables. In the third line, we use the `open3d` function to open a new rgl device, and set certain parameters. You can learn about the details of this function through the help system using the command `help("open3d")`. In the fourth line, the `rainbow` function creates a vector containing system colour codes. Note that we use the predictions of the regression model as an input argument for this function. In the plot, this allows the predictions to change colour with changing values of the explanatory variables. The fifth line makes use of the `plot3d` function to draw a three dimensional scatter plot. The function accepts several input arguments. The first three arguments specify three vectors representing, respectively, the x, y, and z dimensions of the Cartesian coordinate system. The fourth argument instructs the function to display spheres at the locations specified by the coordinates, and the fifth argument specifies the size of the spheres. You can learn about the other input arguments through the help system using the command `help("plot3d")`. In the last line, the `planes3d` function draws a plane using the parametrisation ax + by + cz + d = 0. Here 'a' and 'b' refer to the coefficient estimates of educ and exper, respectively. 'c' takes a value of –1 since 'z' represents the dependent variable, which is wage. 'd' represents the constant term of the regression equation.

In the plot, note how the slope of the plane confirms the magnitudes of the coefficient estimates of educ and exper relative to each other. That is, the slope of the plot is steeper at larger values of educ than at larger values of exper, confirming the larger magnitude of the coefficient estimate of educ than that of exper.

```
install.packages("rgl")
library(rgl)
y_hat = X%*%B_hat
open3d(windowRect = c(100,100,900,900),family = "serif")
color = rainbow(length(y_hat))[rank(y_hat)]
plot3d(educ,exper,wage,col = color,type = "s",size = 0.5,xlim = c(0,20),
       ylim = c(0,60),zlim = c(-10,70),box = FALSE,axes = TRUE)
planes3d(B_hat[2],B_hat[3],-1,B_hat[1],alpha = 0.5,col = "azure")
```



18. Working with a Custom-built Function

So far we have been working with a script file to keep and execute program code. A script file is the most basic program file type of R. A second type of program file is the function file. Like the script file, the function file takes the file extension .R. A function file contains only a function that accepts input arguments and returns output. We call the function from a script file and therefore the function acts as input for the command we execute in a script file. In our script file we have already been calling built-in R functions. In this section we will learn working with a custom-built function.

Double click on the function file `RWFunctionLSS.R` located on the Desktop to open it the Editor. Inspect the content of the function file. The first line always declares a function name. The function name is followed by the built-in syntax `function(argument list)`. We name our function as `RWFunctionLSS`, and the input arguments are `y` and `X`. This is followed by a left brace. The lines beneath the left brace contain executable statements which compute OLS estimates and related regression statistics. Statements should be assigned custom names. E.g., we assign `nrow(y)` to the custom name `N`. A typical function file includes the built-in `return` function as its last statement. This function accepts as input arguments the names of the statements, and it returns as output a list of the output produced by the statements of the function when the function is called. The right brace marks the end of the executable

statements of the function.

`RWFunctionLSS.R` contains new command syntax. E.g., `length` is a built-in function that simply returns the number of elements contained in a vector object.

To access the function defined in `RWFunctionLSS.R` in a script file, we need to make the content of the function file known to R. Go to the function file, select the content of the file, and click on the 'Run' button. Alternatively, click on the tab holder of the function file, and then on the 'Source' button. It is also possible to load the content of the function file using command syntax of the form `source("M:/RWFunctionLSS.R")`. We can now call the custom-built function in our script file. Consider the function syntax `RWFunctionLSS.Output = RWFunctionLSS(y,X)`. We have already defined the input arguments `y` and `X`. Executing the function syntax returns a list object named `RWFunctionLSS.Output`. See the created list object in the Environment. It contains standard regression statistics. E.g., `t` produces the t-statistic that shows if estimated coefficients are statistically significant at a given significance level. You can also access individual statistics produced by the `RWFunctionLSS` function from your script file. E.g., to obtain the OLS coefficient estimates, use object indexing of the form `RWFunctionLSS.Output[3]`.

You can compare the output of the custom-built function `RWFunctionLSS` with that of the built-in function `lm` which contains estimates of the parameters of a linear regression model and various other regression statistics similar to those produced by the `RWFunctionLSS` function. Consider the function syntax `lrm = lm(y ~ educ + exper)`. The input arguments of the function take a special form. First, the outcome variable is specified. After this the sim symbol (~) is specified which is followed by the explanatory variables added together with a sum operator. The `lm` function includes a constant term by default so that we do not need to specify it. Inspect the output the function has just returned in the Environment. Alternatively, use the function syntax `summary(lrm)` to obtain a summary of the output produced by the `lm` function. This demonstrates that it is not difficult to build your own function similar to a built-in R function.

```
source("M:/RWFunctionLSS.R")
RWFunctionLSS.Output = RWFunctionLSS(y,X)
RWFunctionLSS.Output[3]
lrm = lm(y ~ educ + exper)
summary(lrm)
```

Content of RWFunctionLSS.R:

```
# R Workshop - Function File - Least Squares Statistics

RWFunctionLSS = function(y,X)
{
# Number of observations and column dimension of X
N         = nrow(y)
K         = ncol(X)
# OLS estimates, predictions, residuals
B_hat     = solve(t(X)%*%X)%*%(t(X)%*%y)
y_hat     = X%*%B_hat
u_hat     = y-y_hat
```

```
# Analysis of variance
ess       = t(y_hat)%*%y_hat
tss       = t(y)%*%y
rss       = t(u_hat)%*%u_hat
R2_uc     = ess/tss
Mi        = diag(N)-rep(1,N)/N
tss_c     = t(y)%*%Mi%*%y
R2_c      = 1-rss*solve(tss_c)
# Inference
B_hat_var = 1/(N-K)*(solve(t(X)%*%X)%*%t(X))%*%(drop(t(u_hat)%*%u_hat)*X%*%
            solve(t(X)%*%X))
B_hat_se  = sqrt(diag(B_hat_var))
t         = B_hat/B_hat_se
p         = (1-pnorm(abs(t)))*2
return(list(N,K,B_hat,y_hat,u_hat,ess,tss,rss,R2_uc,Mi,tss_c,R2_c,B_hat_var,
            B_hat_se,t,p))
}
```

19. The Optimisation Problem of an Econometrician

Consider once again the linear regression model with multiple regressors that take the form $y = \mathbf{X}\boldsymbol{\beta} + u$. In Section 16, we have used the F.O.C. of the least squares problem to obtain estimates of the true parameter vector $\boldsymbol{\beta}$. In this section we will minimise the least squares objective function using a built-in optimisation algorithm to obtain estimates of $\boldsymbol{\beta}$.

Consider the function `optim` in second line of the list of commands presented at the end of the section. `optim` stands for optimisation. It is an iterative algorithm that attempts to find the local minimiser of an objective function in a given interval. `optim` is a search algorithm and does not guarantee a global minimum. It turns out that any derivative-based optimisation algorithm does not guarantee a global minimum.

`optim` accepts a number of input arguments. Here we do not consider all possible input arguments, and leave them out of the function, and R will assume default values for these input arguments. See `help("optim")` to learn about the possible input arguments that allow you to control the way the search algorithm works. The first input argument specifies an initial guess for `par`. `par` is a vector containing the true coefficients of the regression equation. It is a K×1 vector because we have K parameters to estimate. Let us denote this vector as `B_true`. `optim` uses an iterative algorithm that tries to find the `B_true` that minimize the objective function. The algorithm starts its search of the minimum with an initial guess which we specify as `c(1,1,1)`. `par` can be a scalar, vector, or a matrix.

The second input argument specifies the objective function to be minimised. Our objective function is the sum of squared errors. This function is contained in the function file `RWFunctionSSE.R`.

The third input argument specifies the first derivative of the objective function. This is the gradient, it is defined by the user, and it has a closed form solution. It is contained in the function file `RWFunctionGRD.R`. When supplied with a user-defined gradient, the minimisation of the objective function is most precise. If we do not specify a gradient, `optim` will need to approximate the gradient using numerical methods. In this case, the price we pay is that the algorithm can get stuck in some regions of the parameter space and therefore fail to converge.

The fourth input argument specifies the data which is defined in the first line of the list of

commands presented at the end of the section.

`optim` will return several arguments as output. This output is stored in `result` which is a list object. You can access the objects in this list, using indexing. For instance, to access the first object in the list, execute the command `result[1]`. The following are the output `optim` returns. `par` is where the objective function attains its minimum. Hence, `par` is the solution of the objective function, and gives the `B_hat`. `value` is the value of the objective function at the solution `par`. `counts` counts the number of objective function evaluations before a solution is found. `convergence` is an integer code which is either 0 indicating successful convergence, or 1 indicating that the iteration limit had been reached before the function attains an optimum. Finally, `message` indicates a reason if optimisation was not successful.

Consider the syntax contained in `RWFunctionSSE.R`. The first line declares a function name, and the input arguments of the function. The function accepts as input arguments the data frame containing the vector `y` and matrix `X`, and the parameter vector `B_true` which we denote as `par`. The statement of the function contains the built-in `with` function. The `with` function evaluates the objective function using the data for `y` and `X`. The idea is that, using the observed data, we choose as estimates those values of `B_true` that minimise the square of the error `y-X*B_true`. The minimised square error is a scalar.

Consider the syntax contained in `RWFunctionGRD.R`. The file contains the gradient of the objective function. It returns the derivative of the objective function at the optimising values of `B_true`. Hence the gradient is a vector with dimensions equal to the dimensions of `B_true`. In particular, it is a 3×1 vector.

To execute the `optim` function, you first need to load the functions `RWFunctionSSE.R` and `RWFunctionGRD.R` to the system memory. Otherwise `optim` cannot be executed.

```
source("M:/RWFunctionSSE.R")
source("M:/RWFunctionGRD.R")
data = data.frame(y,X)
result = optim(par = c(1,1,1),fn = RWFunctionSSE,gr = RWFunctionGRD,data = data)
result[1]
```

Content of RWFunctionSSE.R:

```
# R Workshop - Function File - Sum of Squared Errors

RWFunctionSSE = function(data,par)
{
with(data,t((y-X%*%par))%*%(y-X%*%par))
}
```

Content of RWFunctionGRD.R:

```
# R Workshop - Function File - Gradient

RWFunctionGRD = function(data,par)
{
```

```
with(data,2%*%(t(X)%*%X)%*%par-2%*%t(X)%*%y)
}
```

20. The Optimisation Problem of an Agent

In the preceding section we have dealt with an econometric problem, which was estimating the parameters of a regression model using a built-in optimisation algorithm. In this section we will deal with an economic problem, and in particular, with maximising consumer utility using a built-in optimisation algorithm.

Consider a consumer choosing a bundle of goods to maximise her utility. Assume that her utility is represented by a function of the form $u(x_1, x_2) = x_1^{1/2} x_2^{1/2}$. This is a simple Cobb Douglas utility function. Note that it is non-linear. $x1$ and $x2$ represent the quantities of two goods, and u represents utility. Assume also that the agent is subject to a budget constraint represented by the inequality $I \geq p_1 x_1 + p_2 x_2$. p_1 and p_2 represent the prices of the two goods, and I the total income available for consumption.

We will use the built-in `constrOptim` function to find the bundle of goods maximising the utility of the consumer. `constrOptim` is a function that finds the minimum of a function subject to linear inequality constraints. Inspect at `help("constrOptim")` the formal representation of the optimisation problems it solves.

Consider the function syntax presented in the fourth line of the list of commands at the end of the section. The `constrOptim` function accepts a number of input arguments. In the example it accepts five arguments. Additional input arguments can be specified depending on the details pursued. Let `X` denote the vector of choice variables which are quantities of each good the agent wants to consume ($x1$ and $x2$). `constrOptim` uses an iterative algorithm that tries to find the `X` that minimise the objective function subject to a constraint. The algorithm starts its search of the minimum with an initial guess. `X_ig` is the initial guess for `X`. This is the first input argument.

The second input argument specifies that the objective function is `RWFunctionCDU` declared in `RWFunctionCDU.R`.

The other input arguments, following the first two input arguments, specify the components of the constraint function. `constrOptim` accepts linear inequality constraints of the form $Ax \geq b$. The input arguments A and b are made known to `constrOptim` by specifying them in correct order in `constrOptim`. That is, first A is specified and then b is specified. Our linear inequality constraint is $p_1 x_1 + p_2 x_2 \leq I$. Let us define `P` to be a vector object containing good prices p_1 and p_2, and assume that they are 4 and 7, respectively. Let us define `I` as the vector object representing the total income available for consumption, and assume that it is 100 euros. We can now specify `P` and `I` as input arguments in `constrOptim`. We can also specify a lower bound for the choice variables. That is, assume that the smallest quantities of each good to be consumed are both 0. This is also an inequality constraint, and hence we can incorporate it in the vector objects `P` and `I`. Note that `P` creates a matrix object where it places each vector object in a row using the `rbind` function, and `I` creates a simple vector object. When we multiply the `P` matrix, which is a 3×2 matrix, with the `X` vector, which is a 2×1 column vector, and set it to greater than or equal to the `I` matrix which, is a 3×1 column vector, we obtain all the described inequality constraints. Convince yourself that this is true.

The output the `constrOptim` function returns is stored in the `result` list object. The list contains several vectors. `par` is where the objective function attains its minimum. It represents the optimal consumption bundle. `value` is the value of the objective function at the solution `par`. It represents the utility from consuming the optimal consumption bundle. For brevity we

do not discuss here the other output arguments.

Consider the content of `RWFunctionCDU.R`. The first line declares a function name, and the input argument of the function. The input argument of the function is the vector of choice variables `X`. In fact, our utility function takes two input arguments (x_1 and x_2). However, `constrOptim` is designed to work with a single vector of variables. Therefore, the third and fourth lines of the function define the vector array `X` that contains the quantities of two goods to be consumed. `X` is a 2×1 vector because the agent consumes two goods. In the next line we define the objective function. We take the negative of the objective function because `constrOptim` uses a minimisation, and not a maximisation algorithm. The last line ends the function.

This example is taken from Adams, A., Clarke, d., and Quinn, S., 2015. Microeconometrics and MATLAB: An Introduction. Oxford University Press. The code is modified, but also adjusted for R, however.

```
source("M:/RWFunctionCDU.R")
X_ig = c(15,5)
P = rbind(c(-4,-7),c(1,0),c(0,1))
I = c(-100,0,0)
result = constrOptim(X_ig,f = RWFunctionCDU,grad = NULL,P,I)
result[1]
```

Content of RWFunctionCDU.R:

```
# R Workshop - Function File - Cobb Douglas Utility

RWFunctionCDU = function(X)
{
x1 = X[1]
x2 = X[2]
-(x1^0.5)*(x2^0.5)
}
```

21. Debugging Program Code

At the end of this section you will find a for loop. This is the same for loop used in Section 14 with the exception that it now contains an error, or as it is usually called, a bug. In this section you will learn how to debug your code.

Have the script file `RWScript.R` on display. Highlight all the code in the script file. Click on the 'Run' button located at the right hand side of the Editor. R will start executing the code in the script file, but will break the execution due to an error. In the Console, R will tell you about where the error is, and even what the error is about. We will take a number of simple steps to inspect the error and fix it.

In the Console, R indicated the part of the code where the error is. The error occurs on line 206 in the script file, which is part of Section 21. Go to this section, and click just left of line 201 which is the first executable line of Section 21. A breakpoint (•) will appear. This asks R to pause the execution of the entire code at this point. If we do not receive an error until this pause, we know that the error is not in part of the code before the pause. It makes sense to

set multiple breakpoints (pauses) so that we can carry out our search of the error progressively from one breakpoint to the other, along the script file. Let us set another breakpoint at line 203 since we suspect that the error is not related to loading the data.

Once again, highlight all the code in the script file, and at the right hand side of the Editor, click this time on the 'Source' button. Since we have set breakpoints, pressing the Source button will now put R in the 'Debug' mode', and produce the following results. The Console will display the 'Debug' toolbar, and the Console itself will enter in the Debug mode. R pauses at the first breakpoint. Indeed, go to line 201, and notice the green right-arrow just to the left of line 201, which indicates the pause. R does not execute the line where the pause occurs until it resumes running. Notice that we did not receive an error, and hence realise that the error should be in the remaining part of the script file. Go back to the Console, and press the Continue button to continue execution of the code until the next breakpoint. Notice the green right-arrow at line 203 where R pauses for a second time. Again, we did not receive an error, and hence realise that the error should be ahead in the code. Meanwhile, notice that the content of the Environment is updated with respect to the code executed between the breakpoints. Go back to the Console, but now press the 'Next' button, to continue execution of only the next line of the code. The Next button allows you to slow down with your check because you suspect that the error is about to occur. R executes line 203 and returns no error, and moves the green right-arrow to line 204. Press the Next button. R executes line 204 and returns no error. Notice how the debugging feature of R allows you step in the execution of a for loop, making code debugging easy. Press the Next button. Again, R returns no error. Press the Next button. R returns an error and automatically quits the Debug mode, inviting you to fix the error.

The error is that the column dimension of the data frame `rwrdatafile` is 6, but on line 206, the data frame indexing is not correct. In particular, in the syntax `rwrdatafile[:,7]`, we specify the column subscript as 7 but the data frame has 6 columns. Hence, R complains that 'undefined columns selected'. Therefore, the fix to the error is to replace 7 with a smaller number. You have just debugged your code!

```
rm(list = ls())
load("M:/RW rdata file.RData")
uniq = unique(rwrdatafile[,2])
length = length(uniq)
for (i in 1:length) {
  rwrdatafile = rwrdatafile[-(which(rwrdatafile[,7] == uniq[i])[1]),]
}
```

22. Beyond Base R

In this workshop we began with base R to build an understanding of how the language works at its core. In this section we shift to a set of more modern, high-level tools that streamline common tasks such as data wrangling, visualization, and working with external packages. These additions do not replace base R, but they make some, if not many, workflows faster, clearer, and more consistent.

`iris` is a built-in R dataset containing measurements of iris flowers. It includes five columns: four numeric variables representing sepal and petal dimensions, and one categorical variable indicating the species (setosa, versicolor, or virginica). Our aim is to create a new data frame, named `IrisMutated`, that includes all original columns, along with a new column called `Sepal.Area`, defined as the product of `Sepal.Length` and `Sepal.Width`.

For this exercise we will make use of the `dplyr` package, the pipe operator `%>%`, and the `mutate` function. This package is needed in this exercise because it provides the `mutate` function and the pipe operator `%>%`, both of which are important for transforming data in an efficient way. While base R can also create new columns, `dplyr` offers a streamlined syntax that supports clear, step-by-step workflows.

Install the `dplyr` package, and load it into memory. Load the built-in `iris` dataset and view it to understand its structure. The key operation uses the pipe operator to pass the `iris` data frame into the `mutate` function, which adds a new column named `Sepal.Area`. This column is calculated by multiplying `Sepal.Length` and `Sepal.Width` for each observation. The result is stored in a new data frame called `IrisMutated`, which contains both the original data and the newly computed variable.

`mtcars` is a built-in R dataset containing specifications and performance metrics for 32 car models from the 1974 Motor Trend US magazine. It includes 11 columns: numeric variables such as miles per gallon (`mpg`), horsepower (`hp`), and weight (`wt`), as well as categorical-like variables such as the number of cylinders (`cyl`) and transmission type (`am`). Each row represents a different car model. Using the built-in `mtcars` dataset in R, our aim in this exercise is write code that returns the top two cars with the highest horsepower (`hp`) within each number of cylinders (`cyl`) group. Our output should include the car name (rownames), number of cylinders, and horsepower.

For this exercise we make use of the `tibble` and `dplyr` packages. The `tibble` package is required because it provides the `rownames_to_column` function, which converts the row names of the `mtcars` dataset into a proper variable that can be used in subsequent operations. The `dplyr` package is needed for the pipe operator `%>%` and for data manipulation verbs such as `group_by`, `slice_max`, and `select`, which allow us to efficiently identify the two cars with the highest horsepower within each cylinder group.

We need to convert rownames to a column using the `rownames_to_column` function. We also need to use `group_by`, `slice_max`, and `select` functions. After the line of code viewing the data, the first line converts rownames to a column so car names are preserved. The second line groups by number of cylinders and selects the top 2 by horsepower.

For a third exercise, we once again use the `iris` dataset. Our aim is to create a scatter plot of `Sepal.Length` versus `Sepal.Width`, colored by `Species`.

For this exercise we make use of the `ggplot2` package, which provides the functions needed to construct layered and customizable graphics.

Load the `iris` dataset into R. Install the `ggplot2` package and load it into memory. In the code, `geom_point` creates a scatter plot. The aesthetic mapping `aes(x = Sepal.Length, y = Sepal.Width, color = Species)` assigns `Sepal.Length` to the x-axis, `Sepal.Width` to the y-axis, and uses `Species` to color the points. Clear labeling is added through `labs()`, which specifies the plot title and axis labels. This is the only option that fully satisfies the requirement: a scatter plot of Sepal Length vs. Sepal Width, colored by Species.

```
rm(list = ls(all = TRUE))
install.packages("dplyr")
library(dplyr)
data(iris)
View(iris)
IrisMutated = iris %>% mutate(Sepal.Area = Sepal.Length * Sepal.Width)

rm(list = ls(all = TRUE))
install.packages("tibble")
```

```
library(tibble)
library(dplyr)
data(mtcars)
View(mtcars)
mtcarsNamed = mtcars %>% rownames_to_column(var = "car")
TopTwo = mtcarsNamed %>% group_by(cyl) %>% slice_max(hp,n = 2) %>%
  select(car,cyl,hp)

rm(list = ls(all = TRUE))
install.packages("ggplot2")
library(ggplot2)
data(iris)
View(iris)
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species)) +
  geom_point() +
  labs(title = "Iris Sepal Dimensions",
       x = "Sepal Length (cm)", y = "Sepal Width (cm)")
```

23. Help System

R's help system consists of two main parts: the R Documentation, and the R Online forums. Consider first the R Documentation. On the program menu, click on 'Help', and select 'R Help'. This will bring the 'R Documentation' window. Find and click on the 'Search Engine & Keywords' link. Type, e.g., 'mean'. R will present all the documentation available on mean.

Alternatively, you can access the documentation through the Console. Execute the command `help.search("mean")` to access all the documentation on mean. To access the documentation of the `mean` function, type `help("mean")`. `?mean` is shorthand for the `help("mean")` command.

R documentation is also available online at the address http://stat.ethz.ch/R-manual/R-devel/library/base/html/. However, searching for keywords at this address is difficult. Instead, a simple search of the keyword coupled with the letter r in an online search engine will usually provide access to the contents of the R documentation.

For your specific questions, you can get help at two online forums. First is the R-Help mailing list, the discussion forum of R users. The web address of the forum is https://stat.ethz.ch/mailman/listinfo/r-help. The second is Stack Overflow. The forum hosts questions about many different programming languages, and therefore you need to carry out your search using the R tag. The web address of the forum is http://stackoverflow.com/questions/tagged/r. To access the forums, you first need to create an account.

```
help.search("mean")
help(mean)
?mean
```