Introduction to R

University of Bristol

Hans Henrik Sievertsen (h.h.sievertsen@bristol.ac.uk)

Version: January 24 - 2020

Plan

1. Getting started & R basics
2. Tidyverse
3. Matrix algebra
4. Functions, control structures and loops

1. Getting started & R basics

Getting started

1. Download this **slide deck** , **example datasets** and **exercises** from github.com/hhsievertsen/rintro
2. Download **R** from stats.bris.ac.uk/R/ and install it.
3. Download **RStudio** https://rstudio.com/ and install it.
4. Open R Studio

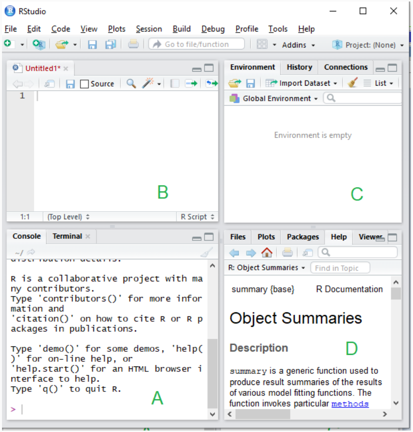
Organisation of RStudio

A. Console

B. Script editor

C. Overview of objects

D. Documentation/plots/file browser/packages



R as a calculator

- We can use R as a calculator. Try typing the following in console and press enter:

```r
5+3
```

```
## [1] 8
```

- You can also type 5+3 in the script editor, highlight what you just wrote and click `Ctrl+Enter`.
- Using the script editor, the keyboard combination `Ctrl+Enter` executes the current line or the selected area.

R basics II

The assignment operator: `<-`

```
value1<-5
```

- The number five is assigned to an object named "value1".
- We can also achieve this using `=` instead of `<-` , but I recommend getting used to using `<-` as it will become of advantage later on.
- We can also use named objects in the calculator approach:

```
value1<-5
value1+3
```

```
## [1] 8
```

R basics III

Printing

- We can ask to display the content of an object using `print()`

```
value1<-5
value2<-3
value3<-value1+value2
print(value3)
```

```
## [1] 8
```

- `R` returns the value of an expression automatically, this is called automatic printing.

```
value1<-5
value2<-3
value3<-value1+value2
value3
```

```
## [1] 8
```

- Automatic printing is disabled in loops, functions etc (more on that later).

R functions

- `print()` is an example of a R function.
- The name of this function is `print`
- The function options (called arguments) go inside the `()`.
- This is general R syntax:
- If you include `()` after a name, R knows it is a function. If you don't include `()` R knows it is not a function.
- Functions can accept many arguments inside the `()`.
- **Ordered** arguments

```
print(value3,TRUE)
```

- **Named** arguments

```
print(x=value3,quote=TRUE)
```

Object types

```r
var<-TRUE
typeof(var)
```

```
## [1] "logical"
```

```r
var<-4L
typeof(var)
```

```
## [1] "integer"
```

```r
var<-4141.2
typeof(var)
```

```
## [1] "double"
```

```r
var<-"Hello1"
typeof(var)
```

```
## [1] "character"
```

- Additional types: NULL,raw, complex, list, expression

Vectors

- We **combine** several objects in a vector using the `c()` function.

```
value1<-5
value2<-3
value3<-value1+value2
vector1<-c(value1,value2,value3)
print(vector1)
```

```
## [1] 5 3 8
```

- a list is **homogeneous**: all objects are **coerced** to be of the same type.

```
object1<-414.041
object2<-"hello!"
vector2<-c(object1,object2)
print(vector2)
```

```
## [1] "414.041" "hello!"
```

(all objects are strings)

R basics VII

Comments

- We should annotate our R script with comments about what we are doing.
- *Problem:* R will try to execute or comments as R code.
- *Solution:* Content after the # symbol is ignored by R.

```
# This is line is ignored by R
this is not ignored by R # but this is
```

R basics VIII

Working directory

- We specify the working directory with ‘setwd()‘.

  (the default location for saving and loading files.)

```
setwd("C:\\Users\\hs17922\\Documents")
```

- Note: use \\ instead of \.

R documention

- Most functions in R are well documented.
- We can access the documentation by typing `?nameoffunction`. For example:

```
?setwd
```

Getting started and R basics - summary

- We assign objects using the the assignment operator `<-`.
- We specify working directory with `setwd()`.
- We access R documentation for the function called `functionname` with `?functionname`.
- We use the `#` to add comments to our script.
- R has five atomic classes of objects: character, numeric, integer, integer, complex (not covered) and logical.
- We create vectors with `c()`
- Lists are homogeneous and only contain one object class.

For more details see chapter 4 in "R Programming for Data Science" (Peng, 2019)

2. Tidyverse

Installing and loading packages

- R is powerful.
- R with extra packages is very powerful.
- **tidyverse** is a collection of packages (ggplot2, tidyr,readr, dplyr and more) that are useful for working with data.
- to **install a package** (examplified by "tidyverse").

  (We need to do this only once on every system.)

```
install.packages("tidyverse")
```

- to **load a packages** (examplified with "tidyverse").

  (We have to do that once for every R session.)

```
library("tidyverse")
```

Data formats

- Datasets come in many formats depending on how they were created and saved (Excel, Stata, etc).
- R can load many types of datasets (but sometimes we have to load a special package to load a specific format).
- `read_csv()` from the `readr` package (included in tidyverse) is convenient for loading datasets ending on ".csv".
- Note that `read.csv()` is a slightly different function.

Loading data into R

Loading data with read_csv()

```
mydataset<-read_csv("example_data1.csv")
```

```
## Parsed with column specification:
## cols(
##   person_id = col_character(),
##   school_id = col_double(),
##   summercamp = col_double(),
##   female = col_double(),
##   parental_schooling = col_double(),
##   parental_lincome = col_double(),
##   test_year_1 = col_double(),
##   test_year_2 = col_double(),
##   test_year_3 = col_double(),
##   test_year_4 = col_double(),
##   test_year_5 = col_double(),
##   test_year_6 = col_double(),
##   test_year_7 = col_double(),
##   test_year_8 = col_double(),
##   test_year_9 = col_double()
## )
```

- The dataset named "example_data1.csv" in the current working directory is now loaded in R under the name `mydataset`.
- The dataset is loaded with 15 columns.
- The first variable is a character (i.e. text) type variable and all outhers are double precision floating point numbers.

For more details on importing data see chapter 11 in "R for data science: import, tidy, transform, visualize, and model data." (Wickham & Grolemund, 2016)

Viewing data

What is in `mydataset`?

- `print()` will (attempt to) display the full dataset in the console. Not feasible for large datasets.
- `head()` displays the first six observations in the dataset.

```
head(mydataset)
```

```
## # A tibble: 6 x 15
##   person_id school_id summercamp female parental_schooling parental_lincome test_year_1 test_year_2 test_year_3 test_year_4
##   <chr>         <dbl>      <dbl>  <dbl>              <dbl>            <dbl>       <dbl>       <dbl>       <dbl>       <dbl>
## 1 p1                5          1      0                 14             15.3       3.25         2.99        2.58        2.16
## 2 p2               14          0      1                 11             14.0       0.993        1.59        1.16        0.817
## 3 p3                7          1      0                 13             15.1       1.82         1.02        2.38        1.88
## 4 p4                8          1      0                 14             15.3       2.15         2.99        2.01        2.23
## 5 p5                9          1      1                 14             15.7       3.03         3.17        2.66        3.02
## 6 p6               26          0      0                 12             14.0       1.52         1.55        1.10        1.53
```

- `tail()` displays the last six observations in the dataset.
- `View()` opens the dataset in a viewer.

Tidying data I

Tidy data

- The **tidy data principles** state that each *variable* must have its own *column* and each *observation* must have its own row.
- The example dataset is not tidy.
  - The variables `test_year_1` to `test_year_9` violate the tidy data principles.
  - The variables contain information about test scores.
  - The values 1,2,. . . ., 9, and 9 are information about the year of the test score, this should be stored in rows for a variable called year.
- The function `pivot_longer()` (from the *tidyr* package) gathers several columns in one column (makes the dataset longer).
- The function `pivot_wider()` (from the *tidyr* package) spreads one column to several columns (makes the dataset wider).

pivot_longer()

- `pivot_longer(data, cols, names_to, values_to=)`
- `data`: the name of the dataset.
- `cols`: the columns to convert.
- `names_to`: the new variable where the information that is currently in the column headers (for example the test year) should be stored.
- `values_to`: the variable where the values from the old rows are to be stored.

```
tidydata<-pivot_longer(mydataset,cols=7:15,
                       names_to="year",values_to="test_score")
options(dplyr.width = Inf)
head(tidydata)
```

```
## # A tibble: 6 x 8
##   person_id school_id summercamp female parental_schooling parental_lincome year        test_score
##   <chr>         <dbl>      <dbl>  <dbl>              <dbl>            <dbl> <chr>            <dbl>
## 1 p1                5          1      0                 14             15.3 test_year_1       3.25
## 2 p1                5          1      0                 14             15.3 test_year_2       2.99
## 3 p1                5          1      0                 14             15.3 test_year_3       2.58
## 4 p1                5          1      0                 14             15.3 test_year_4       2.16
## 5 p1                5          1      0                 14             15.3 test_year_5       2.61
## 6 p1                5          1      0                 14             15.3 test_year_6       3.10
```

(I used `options(dplyr.width = Inf)` to specify the number of columns to print.)

For more details on `pivot_longer()` data see section 12.3.1 in "R for data science: import, tidy, transform, visualize, and model data." (Wickham & Grolemund, 2016).

Tidying data III

pivot_wider()

- `pivot_wider(data, names_from, values_from=)`
- `data`: the name of the dataset.
- `names_from`: the new columns get their names from this variable.
- `values_from`: the new columns get their values from this variable.

```
dirtydata<-pivot_wider(tidydata,
                       names_from="year",values_from="test_score")
options(dplyr.width = Inf)
head(dirtydata)
```

```
## # A tibble: 6 x 15
##   person_id school_id summercamp female parental_schooling parental_lincome test_year_1 test_year_2 test_year_3 test_year_4
##   <chr>         <dbl>      <dbl>  <dbl>              <dbl>            <dbl>       <dbl>       <dbl>       <dbl>       <dbl>
## 1 p1                5          1      0                 14             15.3       3.25         2.99        2.58        2.16
## 2 p2               14          0      1                 11             14.0       0.993        1.59        1.16        0.817
## 3 p3                7          1      0                 13             15.1       1.82         1.02        2.38        1.88
## 4 p4                8          1      0                 14             15.3       2.15         2.99        2.01        2.23
## 5 p5                9          1      1                 14             15.7       3.03         3.17        2.66        3.02
## 6 p6               26          0      0                 12             14.0       1.52         1.55        1.10        1.53
```

(I used `options(dplyr.width = Inf)` to specify the number of columns to print.)

For more details on `pivot_wider()` data see section 12.3.2 in "R for data science: import, tidy, transform, visualize, and model data." (Wickham & Grolemund, 2016).

filter

- `filter(data, criteria)`
- We filter specific rows (observations) of a dataset using the `filter()` function (from the dplyr package).
- `data`: the name of the dataset.
- `...`: the filtering criteria.

```
filtered_data<-filter(tidydata,year=="test_year_2")
options(dplyr.width = Inf)
head(filtered_data)
```

```
## # A tibble: 6 x 8
##   person_id school_id summercamp female parental_schooling parental_lincome year        test_score
##   <chr>         <dbl>      <dbl>  <dbl>              <dbl>            <dbl> <chr>            <dbl>
## 1 p1                5          1      0                 14             15.3 test_year_2       2.99
## 2 p2               14          0      1                 11             14.0 test_year_2       1.59
## 3 p3                7          1      0                 13             15.1 test_year_2       1.02
## 4 p4                8          1      0                 14             15.3 test_year_2       2.99
## 5 p5                9          1      1                 14             15.7 test_year_2       3.17
## 6 p6               26          0      0                 12             14.0 test_year_2       1.55
```

(I used `options(dplyr.width = Inf)` to specify the number of columns to print.)

For more details on `filter()` see section 5.2 in "R for data science: import, tidy, transform, visualize, and model data." (Wickham & Grolemund, 2016).

select

- `select(data, ...)`
- We select specific columns (variables) of a dataset using the `select()` function (from the dplyr package).
- `data`: the name of the dataset.
- `...`: the name (or number) of the columns to keep. To remove a variable, add a "-" in front of the variable.

```
selected_data<-select(filtered_data,
                      c(person_id,summercamp,test_score))
options(dplyr.width = Inf)
head(selected_data)
```

```
## # A tibble: 6 x 3
##   person_id summercamp test_score
##   <chr>          <dbl>      <dbl>
## 1 p1                 1       2.99
## 2 p2                 0       1.59
## 3 p3                 1       1.02
## 4 p4                 1       2.99
## 5 p5                 1       3.17
## 6 p6                 0       1.55
```

(I used `options(dplyr.width = Inf)` to specify the number of columns to print.)

For more details on `select()` see section 5.4 in "R for data science: import, tidy, transform, visualize, and model data." (Wickham & Grolemund, 2016).

Data processing III

rename

- `rename(data, newname1=oldname1,newname=oldname,...)`
- We rename columns using the rename function
- `data`: the name of the dataset.
- `newname1`: the new name of the first column to rename.
- `oldname1`: the old name of the first column to rename.
- `newname2`: the new name of the second column to rename.
- ...

```
renamed_data<-rename(selected_data,
                     score=test_score,camp=summercamp)
options(dplyr.width = Inf)
head(renamed_data)
```

```
## # A tibble: 6 x 3
##   person_id  camp score
##   <chr>     <dbl> <dbl>
## 1 p1            1  2.99
## 2 p2            0  1.59
## 3 p3            1  1.02
## 4 p4            1  2.99
## 5 p5            1  3.17
## 6 p6            0  1.55
```

(I used `options(dplyr.width = Inf)` to specify the number of columns to print.)

For more details on `rename()` see section 5.4 in "R for data science: import, tidy, transform, visualize, and model data." (Wickham & Grolemund, 2016).

Data processing IV

mutate

- `mutate(data, nameofnewvariable=expression, ...)`
- We create and modify columns using the mutate function
- `data`: the name of the dataset.
- `nameofnewvariable`: the name of the first column to rename.
- `expression`: the definition of the new variable
- ...

```
mutated_data<-mutate(renamed_data,
                     camptest=score*camp,constant=1)
options(dplyr.width = Inf)
head(mutated_data)
```

```
## # A tibble: 6 x 5
##   person_id  camp score camptest constant
##   <chr>     <dbl> <dbl>    <dbl>    <dbl>
## 1 p1            1  2.99     2.99        1
## 2 p2            0  1.59     0           1
## 3 p3            1  1.02     1.02        1
## 4 p4            1  2.99     2.99        1
## 5 p5            1  3.17     3.17        1
## 6 p6            0  1.55     0           1
```

(I used `options(dplyr.width = Inf)` to specify the number of columns to print.)

For more details on `mutate()` see section 5.5 in "R for data science: import, tidy, transform, visualize, and model data." (Wickham & Grolemund, 2016).

Data processing V

merge

- `merge(x,y,by="matchingvar")`
- We merge two datasets using the `merge()` function.
- `x`: the name of the first dataset.
- `y`: the name of the second dataset.
- `matchingvar`: rows with the same value of `matchingvar` in both `x` and `y` are matched.
- ...

```
myotherdataset<-read_csv("example_data2.csv")
merged_data<-merge(mutated_data,myotherdataset,by="person_id")
options(dplyr.width = Inf)
head(merged_data)
```

```
##   person_id camp     score camptest constant rct
## 1        p1    1 2.9860402 2.986040        1   1
## 2       p10    1 2.1003354 2.100335        1   0
## 3      p100    0 2.4815373 0.000000        1   0
## 4      p101    0 0.7744099 0.000000        1   1
## 5      p102    1 2.5056268 2.505627        1   1
## 6      p103    1 2.6484405 2.648441        1   0
```

(I used `options(dplyr.width = Inf)` to specify the number of columns to print. Some output is hidden.)

For more details on `merge()` see section 13.4.7 in "R for data science: import, tidy, transform, visualize, and model data." (Wickham & Grolemund, 2016).

Data processing VI

The pipe: %>%

- Throw the left-hand side value forward into the right-hand side expression.
- So $f(x)$ can be written as as x%>%$f()$.
- **Example**

```
renamed_data<-rename(mydataset,score=test_score)
mutated_data<-mutate(renamed_data,camptest=score*camp)
filtered_data<-filter(mutated_data,female==1)
selected_data<-select(filtered_data,person_id,camptest)
```

can be written as:

```
selected_data<-mydataset%>%
               rename(score=test_score)%>%
               mutate(camptest=score*camp)%>%
               filter(female==1)%>%
               select(person_id,camptest)
```

For more details on the pipe see chapter 18 in "R for data science: import, tidy, transform, visualize, and model data." (Wickham & Grolemund, 2016).

Data processing VII

group_by & summarise

- `group_by(data,...)`
  - `data`: the name of the dataset.
  - `...`: the names of the columns to group the dataset by.
- `summarise(data,varname=expression)`
  - summarises the dataset on the `group_by` (if defined) level.
  - `data`: the name of the dataset.
  - `varname`: the name of the new variable.
  - `expression`: the definition of the new variable.

```r
summarised_data<-mydataset%>%
                 group_by(summercamp)%>%
                 summarise(average_score=mean(test_year_2))
print(summarised_data)
```

```
## # A tibble: 2 x 2
##   summercamp average_score
##        <dbl>         <dbl>
## 1          0          1.98
## 2          1          2.51
```

For more details see "section 5.6 in R for data science: import, tidy, transform, visualize, and model data." (Wickham & Grolemund, 2016).

Base graphics I

base refers to base R (without any functionality from loaded packages).

`plot(x,y,..)`

-`x` the variable to plot on the x-axis.

-`y` the variable to plot on the y-axis.

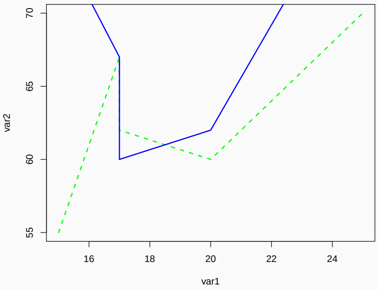
-`...` settings ()

- `type = "l"`: line chart type (try also "o", "p", "l", & "b").
- `lty = "dotted"`: dotted line type (try also "dashed").
- `col = "blue"`: blue line.
- `lwd=2`: line width.
- `lines()` or `points()` to add more lines

For more details see R Base Graphics: An Idiot's Guide.

Base graphics II

```
var1<-c(15, 17, 17, 20, 25)
var2<-c(55, 67, 62, 60, 70)
var3<-c(75, 67, 60, 62, 80)
plot(var1,var2,
     type="l",lty="dashed",col="green",lwd=2)
lines(var1,var3,
      type="l",lty="solid",col="blue",lwd=2)
```



ggplot I

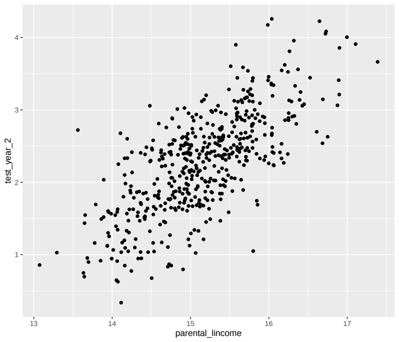
ggplot2 a grammar for graphics

- `ggplot(data,aes(x,y,...))`
- `data`: the name of the dataset.
- `...`: `aes()` the aesthetic mappings `-x` the variable to plot on the x-axis. `-y` the variable to plot on the y-axis.
- `+geom_line()`
- + add a layer to the ggplot object
- `geom_line()` add a line chart using the data and the aesthetic mappings specified in ggplot() (`geom_line` inherits the settings specified in `ggplot()`).
- `+geom_point(data,aes(x,y,...))`
- + add a layer to the ggplot object
- `geom_point()` add a scatter plot using the data and the aesthetic mappings specified specifeid within `geom_point()`.
- `+theme()`: specify theme settings (colors, position etc.)
- `+labs()`: specify axes titles, chart title, caption, legend titles, etc..

For more details see "chapter 3 in R for data science: import, tidy, transform, visualize, and model data." (Wickham & Grolemund, 2016).

ggplot2 a grammar for graphics

```
ggplot(mydataset,aes(x=parental_lincome,y=test_year_2))+
 geom_line()
```



Tidyverse summary

- Tidyverse package tools for working with data: load, tidy, process, visualize data.
- We install packages with `install.package()`.
- We load packages with `library()`.
- We use `pivot_wider()` and `pivot_longer()` to tidy the dataset.
- We use `mutate()` to create new/modify columns in our dataset.
- We use `select()` to specify which columns to keep/remove.
- We use `filter()` to specify which rows to keep.
- We use `group_by()` and `summarise()` to create aggregate statistics.
- We use `ggplot()` to create charts.
- We use `merge()` to merge datasets.

3. Matrix algebra

A vector

- We would like to enter the following vector A into R

$$A = \begin{bmatrix} 3 \\ 5 \\ 4 \end{bmatrix}$$

- We already know how to do this. We simply use `c()`:

```r
A=c(3,5,4)
A
```

```
## [1] 3 5 4
```

- Note that R prints it as row, but it is a column vector.

A vector transposed

- To verify that A is really a column vector, let's consider the transpose of A:

$$A^T = \begin{bmatrix} 3, 5, 4 \end{bmatrix}$$

- which we can achieve using `t()` in R:

```
t(A)
```

```
##      [,1] [,2] [,3]
## [1,]    3    5    4
```

- Where the `m` in `[m,n]` refers to the **row** and the `n` to the **column**.
- and let's consider the transpose of the transposed matrix to get back to the original A vector.:

```
t(t(A))
```

```
##      [,1]
## [1,]    3
## [2,]    5
## [3,]    4
```

The matrix function

- We can also explictly create vectors and matrices using the `matrix()` function.
- To create our A vector, we write:

```
A=matrix(c(3,5,4),ncol=1)
A
```

```
##      [,1]
## [1,]    3
## [2,]    5
## [3,]    4
```

- Note the difference between

```
A=matrix(c(3,5,4),ncol=1)
class(A)
```

```
## [1] "matrix"
```

and

```
A=c(3,5,4)
class(A)
```

```
## [1] "numeric"
```

Entering a matrix with the matrix function

Creating a 2x2 matrix

- Let us now consider a 2 by 2 matrix B:

$$B = \begin{bmatrix} 3, 5 \\ 11, 2 \end{bmatrix}$$

- which we can enter as:

```
B=matrix(c(3,11,5,2),ncol=2)
B
```

```
##      [,1] [,2]
## [1,]    3    5
## [2,]   11    2
```

- R first fills the first column, then the second etc...
- and the transpose of B:

```
t(B)
```

```
##      [,1] [,2]
## [1,]    3   11
## [2,]    5    2
```

Adding a number to a matrix

- Let α be a number (a scalar), then $\alpha + B$ is:

$$\alpha B = \begin{bmatrix} \alpha + 3, \alpha + 5 \\ \alpha + 11, \alpha + 2 \end{bmatrix}$$

- and in R:

```
B=matrix(c(3,11,5,2),ncol=2)
alpha=0.5
alpha+B
```

```
##      [,1] [,2]
## [1,]  3.5  5.5
## [2,] 11.5  2.5
```

Addition

Adding two matrices together

- Consider the following two matrices

$$C = \begin{bmatrix} 1, 2 \\ 3, 4 \end{bmatrix} \text{ and } D = \begin{bmatrix} 5, 6 \\ 7, 8 \end{bmatrix}$$

- The sum of these two matrices is then given by::

$$C + D = \begin{bmatrix} 1 + 5, 2 + 6 \\ 3 + 7, 4 + 8 \end{bmatrix}$$

- and in R:

```
C=matrix(c(1,3,2,4),ncol=2)
D=matrix(c(5,7,6,8),ncol=2)
C+D
```

```
##      [,1] [,2]
## [1,]    6    8
## [2,]   10   12
```

Addition

Warning: pay attention to dimensions of matrices

- Note: the dimensions have to align when adding two matrices together:

```
E=matrix(c(1,3,2,4),ncol=1)
F=matrix(c(5,7,6,8),ncol=2)
E+F
```

```
## Error in E + F: non-conformable arrays
```

- E is a 1×4 matrix.
- F is a 2×2 matrix.

  -We want to add the element in the first row and first column of E to the element in the first row and the first column of F That's okay.

  - ...
  - We want to add the element in the third row and first column of E to the element in the third row and the first column of F. That's not doable, because F only has two rows.

Multiplication

Multiplying a matrix with a number

- Let α be a number (a scalar), then αB is:

$$\alpha B = \begin{bmatrix} \alpha \times 3, \alpha \times 5 \\ \alpha \times 11, \alpha \times 2 \end{bmatrix}$$

- and in R:

```
B=matrix(c(3,11,5,2),ncol=2)
alpha=0.5
alpha*B
```

```
##      [,1] [,2]
## [1,]  1.5  2.5
## [2,]  5.5  1.0
```

Multiplication

Element-wise multiplication

- Consider the following two matrices

$$C = \begin{bmatrix} 1, 2 \\ 3, 4 \end{bmatrix} \text{ and } D = \begin{bmatrix} 5, 6 \\ 7, 8 \end{bmatrix}$$

- The Hadamard product (or element-wise multiplication) of matrices C and D is then given by:

$$E = C \circ D = \begin{bmatrix} 1 \times 5, 2 \times 6 \\ 3 \times 7, 4 \times 8 \end{bmatrix}$$

- and in R:

```
C=matrix(c(1,3,2,4),ncol=2)
D=matrix(c(5,7,6,8),ncol=2)
C*D
```

```
##      [,1] [,2]
## [1,]    5   12
## [2,]   21   32
```

Multiplication

Warning: pay attention to dimensions of matrices

- Note: element-wise multiplication of two matrices also requries that dimensions (i.e. number of rows and columns) match :

```
E=matrix(c(1,3,2,4),ncol=1)
F=matrix(c(5,7,6,8),ncol=2)
E*F
```

```
## Error in E * F: non-conformable arrays
```

- E is a 1×4 matrix.
- F is a 2×2 matrix.

-We want to multiply the element in the first row and first column of E with the element in the first row and the first column of F That's okay.

  - ...
  - We want to multiply the element in the third row and first column of E with the element in the third row and the first column of F. That's not doable, because F only has two rows.

Multiplication

Matrix multiplication

- Consider again the following two matrices

$$C = \begin{bmatrix} 1,2 \\ 3,4 \end{bmatrix} \text{ and } D = \begin{bmatrix} 5,6 \\ 7,8 \end{bmatrix}$$

- Let's now consider the product of matrices E and D:

$$CD = \begin{bmatrix} 1 \times 5 + 2 \times 7, 1 \times 6 + 2 \times 8 \\ 3 \times 5 + 4 \times 7, 3 \times 6 + 4 \times 8 \end{bmatrix}$$

- and in R:

```
C=matrix(c(1,3,2,4),ncol=2)
D=matrix(c(5,7,6,8),ncol=2)
C%*%D
```

```
##      [,1] [,2]
## [1,]   19   22
## [2,]   43   50
```

Warning: pay attention to dimensions of matrices

- Note: matrix multiplication of two matrices requries that the number of rows in the left hand side matrix correspond to the number of columns in the right hand side matrix.

```
E=matrix(c(1,3,2,4),ncol=1)
F=matrix(c(5,7,6,8),ncol=2)
E%*%F
```

```
## Error in E %*% F: non-conformable arguments
```

- E is a 1×4 matrix.
- F is a 2×2 matrix.

-We want to multiply the elements of the first row in matrix E to the elements of the first column of matrix F, but the former has one element and the latter has two elements!

Some special matrices

- A 0-matrix (all entries are zero):

```r
matrix(0, nrow = 2, ncol = 2)
```

```
##      [,1] [,2]
## [1,]    0    0
## [2,]    0    0
```

- A *J* matrix (all entries are 1s):

```r
matrix(1, nrow = 2, ncol = 2)
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    1    1
```

- A matrix where all entries outside the diagonal are zero:

```r
diag(c(1,2,3))
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    2    0
## [3,]    0    0    3
```

Some special matrices

- The identity matrix, I, is a diagonal matrix, where all elements in the diagonal are 1.

```
diag(c(1,1))
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

- An Identity matrix satisfies $IA = AI = A$, where A is a matrix.

```
A=matrix(c(1,3,4,5),ncol=2)
I=diag(c(1,1))
A%*%I
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    3    5
```

We can also appy `diag()` on matrix to extract the diaognal

```
A=matrix(c(1,3,4,5),ncol=2)
diag(A)
```

```
## [1] 1 5
```

- A J matrix (all entries are 1s):

The inverse of a matrix

- Let A be a $n \times n$ matrix (a square matrix, because the number of rows equals the number of columns).
- Let B be $n \times n$ matrix which multiplied by matrix A gives the identity matrix:

$$AB = BA = I$$

- The matrix B is called A's inverse, $B = A^{-1}$.
- Finding the inverse matrix is numerically complicated. But luckily we can ask R to do it for us by means of the `solve()` function:

```
A=matrix(c(1,3,4,5),ncol=2)
solve(A)
```

```
##            [,1]       [,2]
## [1,] -0.7142857  0.5714286
## [2,]  0.4285714 -0.1428571
```

- let's test it:

```
solve(A)%*%A
```

```
##      [,1]          [,2]
## [1,]    1 -4.440892e-16
## [2,]    0  1.000000e+00
```

The determinant of a matrix

- A matrix is not invertible if the determinant is zero (the matrix is then called singular).
- The determinant of matrix A is written as $det(A)$ or $|A|$.
- For a 2x2 matrix, the determinant is defined as

$$det(A) = |A| = \begin{vmatrix} a, b \\ c, d \end{vmatrix} = a \times d - b \times c.$$

- So for the A matrix defined earlier it is given by:

$$det(A) = \begin{vmatrix} 1, 3 \\ 4, 5 \end{vmatrix} = 1 \times 5 - 3 \times 4 = -7.$$

```
A=matrix(c(1,3,4,5),ncol=2)
det(A)
```

```
## [1] -7
```

Combining matrices

Column bind: `cbind(A,B,..)`

- combines matrices A, B, .. horizontally (binds the columns).

```
A=matrix(c(1,3,4,5),ncol=2)
B=matrix(c(1,3,4,5),ncol=2)
cbind(A,B)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    1    4
## [2,]    3    5    3    5
```

Row bind: `rbind()`

- combines matrices A, B, .. vertically (binds the columns).

```
rbind(A,B)
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    3    5
## [3,]    1    4
## [4,]    3    5
```

Application: ordinary least squares

- Let's return to our data and estimate the following model using ordinary least squares:

$$test\_year\_6_i = \beta_0 + \beta_1 parental\_lincome + \beta_2 summercamp_i + e_i$$

- We can achive this with `lm()` function in R:

```
mydataset<-read_csv("example_data1.csv")
my_lm<-lm(test_year_6~parental_lincome+summercamp,data=mydataset)
summary(my_lm)
```

- In `lm()` we first specify the model to estimate: `y~x1+x2+..` (R automatically adds a constant).
- We store the result of fitting the model using OLS in the object called `my_lm`
- We use the `summary()` function to summarize the results.

Application: ordinary least squares

- Let's check the output

```r
mydataset<-read_csv("example_data1.csv")
my_lm<-lm(test_year_6~parental_lincome+summercamp,data=mydataset)
summary(my_lm)
```

```
##
## Call:
## lm(formula = test_year_6 ~ parental_lincome + summercamp, data = mydataset)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.95406 -0.28911  0.00042  0.26314  1.34439
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)      -7.82804    0.42003  -18.64   <2e-16 ***
## parental_lincome  0.66349    0.02821   23.52   <2e-16 ***
## summercamp        0.55130    0.03941   13.99   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4022 on 488 degrees of freedom
## Multiple R-squared:  0.7059, Adjusted R-squared:  0.7047
## F-statistic: 585.6 on 2 and 488 DF,  p-value: < 2.2e-16
```

Manual OLS using R

- Let's try to manually reproduce these results from lm() using the matrix tools we just covered.
- We know that the OLS estimates in matrix form are given by:

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

- We know how to find the inverse, how to multiply matrices and how to transpose matrices in R. We are ready!

But - We need to get the data from the dataset into the matrix format first!

```
mydataset<-read_csv("example_data1.csv")
class(mydataset)
```

```
## [1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

- `mydataset` is a `tibble` (or a data frame), not a matrix.
- A matrix is homogeneous (just like vectors created with `c()`).
- Data frames allow for a mix of types (integer, double, character).

From data frame to matrix: method 1

- We can access a specific column of a dataframe using the $ symbol:

```
mydataset<-read_csv("example_data1.csv")
mydataset$test_year_6
```

```
##   [1] 3.0968914 1.7556519 2.5691076 2.9608262 3.5352431 1.9390220 1.4565211 0.8530146 1.9422049 2.3242188 4.0378935 2.941284
##  [60] 2.6405047 1.9769918 3.0799881 1.9301455 2.6791417 3.0750632 2.6380266 2.2997583 1.9301881 2.3110535 3.9045953 2.730110
## [119] 2.1362477 1.9975301 1.9805466 3.2193071 1.7284275 1.3998921 1.8137896 2.5268532 2.4869136 1.8184657 2.8640394 3.467747
## [178] 2.2438199 2.7951478 2.3820597 3.3607016 1.9551251 2.2914024 2.5629094 2.5516064 2.4793032 1.7563591 2.6096906 2.196540
## [237] 2.8029133 2.6756473 3.0413707 1.9748095 3.8084274 1.5147583 3.2044558 2.7468969 2.1079639 1.9867444 2.7115403 2.129352
## [296] 3.2880623 2.9644639 2.1247169 2.8076262 2.2059820 1.3455721 1.8300033 2.7072177 2.7233655 1.2591008 2.3801160 3.113431
## [355] 2.9584985 1.9224016 2.5383838 2.4929356 3.1961359 3.2343892 4.0994714 2.9938602 2.8084186 3.0216950 2.6169923 3.243022
## [414] 2.2764986 1.7132658 2.3578819 2.5405297 1.6079901 1.4885995 2.6985074 2.2916481 2.6920764 1.5655158 2.4127587 2.354595
## [473] 3.6784746 2.4112565 2.4748137 4.0057407 1.1091630 3.5154961 2.4298711 2.9244919 3.4194154 2.1559040 2.2093878 3.132777
```

- let's save data in a column vector called y:

```
mydataset<-read_csv("example_data1.csv")
y=matrix(mydataset$test_year_6,ncol=1)
```

From data frame to matrix: method 2

- We can convert a data frame to a matrix using the `as.matrix()` function:

```
X<-read_csv("example_data1.csv")%>%
            select(parental_lincome,summercamp)%>%
            mutate(constant=1)%>%
            as.matrix()
class(X)
```

```
## [1] "matrix"
```

Application: Ordinary Least Squares

Let's find the OLS estimates

- We now simply plugin the OLS formula in R

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

```
# load dataset
mydataset<-read_csv("example_data1.csv")
# extract the dependent variable and store as column vector y
y=matrix(mydataset$test_year_6,ncol=1)
# extract the covariates to include and create a constant and as matrix X
X<-mydataset%>%
   select(parental_lincome,summercamp)%>%
   mutate(constant=1)%>%
   as.matrix()
# implement OLS formula
betahat=solve(t(X)%*%X)%*%t(X)%*%y
# show betahat vector
betahat
```

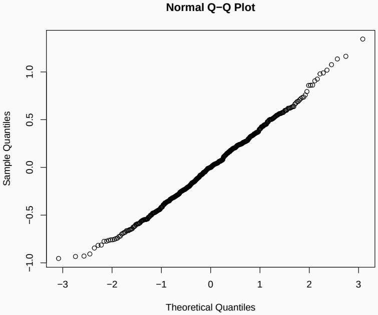
```
##                        [,1]
## parental_lincome  0.6634867
## summercamp        0.5513035
## constant         -7.8280434
```

- Estimated coefficients are identical to the results obtained with `lm()`!

Fitted values

- the fitted values are given by $X\hat{\beta}$.
- the residuals are given by $y - X\hat{\beta}$.
- we can create a q-q plot of the betas using base graphics:

```
#calculate residuals
residuals<-(y-X%*% betahat)
# create histogram
qqnorm(residuals)
```



Matrix algebra - summary

- Create column vector A: `A=c()`.
- Create matrix A: `A=matrix()`.
- Convert B to matrix A: `A=as.matrix(B)`.
- Transpose matrix A: `t(A)`.
- Inverse of matrix A: `solve()`.
- Element-wise multiplication of matrix `A` and `B`: `A*B`.
- Matrix multiplication of matrix `A` and `B`: `A%*%B`.
- Extract the column named `col1` from data frame `df`: `df$col1`.
- OLS estimator: `betahat=solve(t(X)%*%X)%*%t(X)%*%y`.

4. Functions, control structures and loops

Defining functions

What is a function?

- a set of R statements that perform a task given a set of provided arguments.
- Example: `lm(test_year_6~parental_lincome+summercamp,data=mydataset)`
  - The function `lm()` estimates coefficients of a linear model.
  - The model is provided as the first argument (`test_year_6~parental_lincome+summercamp`)
  - The dataset is provided as a second (named) argument (`mydataset`).

User defined functions

- We can easily create our own functions in R.
- The syntax is as follows:

```
function_name <- function(arg_1, arg_2, ...) {
   Function body
}
```

The function consists of four parts:

1. The function name.
2. The arguments (placeholders for settings, datasets etc).
3. The function body (a collection of statements to carry out using the arguments provided).
4. Return value (the last expression of the function)

Our first function

Let's define a function

1. name: Hansfunction
2. arguments: `x` and `y`
3. function body: `z=x*y`
4. return: return(z)

- Let's define that function

```
Hansfunction <- function(x, y) {
   z=x*y
   z
}
```

- Let's try the function

```
Hansfunction(3,6)
```

```
## [1] 18
```

- What if we forget to state an argument?

```
Hansfunction(3)
```

```
## Error in Hansfunction(3): argument "y" is missing, with no default
```

Default values

Let's define a function with default values

- To avoid such cases, we can specify **default** values:
- When defining the function, we set the arguments equal to their default values.
- Let's define that function

```
Hansfunction <- function(x=2, y=2) {
  z=x*y
  z
}
Hansfunction(3)
```

```
## [1] 6
```

- This works well, but if you accidentally forgot an argument!

Control structures

- We can use **control structures** to control the statements executed by our function.
- Here is an example of a control structure in plain English:

```
if logical test evaluates to true do the following
    statements to execute if TRUE
else do the following
    statements to execute if not TRUE
```

- and in terms of R syntax:

```
if (logical test){


  }
  else{


}
```

- A logical test is a statement that evaluates to TRUE or FALSE, for example:
  - “5 is greater than 3” The statement is TRUE
  - “3 is greater than 5” The statement is FALSE
- Control statements can also be used outside functions (in scripts, loops etc).

Control structures in Hansfunction()

```
Hansfunction <- function(x=2, y=3) {
  if (missing(x)|missing(y)){
     print("Warning: Not all arguments provided. Default values used.")
  }
  else{
     print("Well done, you specified all arguments!")
  }
     z=x*y
     z
}
Hansfunction(3)
```

```
## [1] "Warning: Not all arguments provided. Default values used."
```

```
## [1] 9
```

- here we use the function `missing()` to test whether the argument missing.
- the | corresponds to "or" (the logical expression evaluates to true if x OR y are missing).
- We can use the symbol "&" if we only want the expression to evaluate to true if both are x and y are missing.

Let's build our own lm() function

- We now use the tools from the matrix section and combine them with the function definitions to create our own lm function.

```
mylm <- function(y,x,data) {
  # specify dataset
  df<-data
  # extract the dependent variable and the covariates
  yvar<-df%>%select(y)%>%as.matrix()
  Xvar<-df%>%select(x)%>%mutate(constant=1)%>%as.matrix()
  # implement OLS formula and return betahat vector
  solve(t(Xvar)%*%Xvar)%*%t(Xvar)%*%yvar
}
# load data
mydataset<-read_csv("example_data1.csv")
# try our new function
mylm(x=c("parental_lincome","summercamp"),y="test_year_4",data=mydataset)
```

```
##                  test_year_4
## parental_lincome   0.6249679
## summercamp         0.1850357
## constant          -7.3108981
```

For loop

- a for loop repeats a list of statements
- the number of times the statements are repeated is stated in the loop *header*
- the statements are provided in the *body*
- example:

```
for (x in 1:3){
  print(x)
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

- the loop header `for (x in 1:3){` states that the loop should be repeated 3 times:
  1. Once where `x` has the value 1
  2. Once where `x` has the value 2
  3. Once where `x` has the value 3.
- the loop header states that in each loop *iteration* the statment `print(x)` should be executed.

While loop

- we can also create a loop that repats itself until a certain condition is violated.
- this is called a **while** loop.
- the loop body statements are repeated until the while condition is violated.
- example:

```
x=1
while (x<5){
  print(x)
  x=x+1
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

- the object `x` is initiated with a value of 1.
- the loop is repeated until `x<5` evaluates to false.
- in every iteration we:
  - print the value of x.
  - add the value of 1 to x.

Application: maximum likelihood

Maximum likelihood: find the parameters that maximize the likelihood that we observe, given a distribution.

- application: estimate the probability that a child probability participates in the summer school using a probit model.
- **benchmark**: R's built-in function

```
mydataset<-read_csv("example_data1.csv")
# Estimate the probit model
probit<-glm(summercamp~parental_lincome,
            family = binomial(link = "probit"), data = mydataset)
# Show parameter estimates
summary(probit)
```

```
##
## Call:
## glm(formula = summercamp ~ parental_lincome, family = binomial(link = "probit"),
##     data = mydataset)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.2385  -1.0561   0.5512   1.0291   1.8498
##
## Coefficients:
##                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)      -11.93926    1.44268  -8.276   <2e-16 ***
## parental_lincome   0.79602    0.09538   8.346   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
```

Application: maximum likelihood

- The likelihood for a single observation:

$$L(\beta; y_i, x_i) = [\Phi(x_i'\beta)]^{y_i}[1 - \Phi(x_i'\beta)]^{1-y_i}$$

- Observations are assumed to be *iid*, we can therefore write the likelihood of the entire sample as the product of the individual likelihoods:

$$L(\beta; y, X) = \prod_{i=1}^{N}[\Phi(x_t'\beta)]^{y_i}[1 - \Phi(x_t'\beta)]^{1-y_i}$$

- The **log**-likelihood is then given by:

$$l(\beta; y, X) = \sum_t \Big(y_t \ln \Phi(x_t'\beta) + (1 - y_t) \ln \Big(1 - \Phi(x_t'\beta)\Big)\Big)$$

- **R implementation**
  - $\Phi()$ is the cumulative distribution function of the standard normal distribution, which we implement in R with `pnorm()`.
  - `sum()` computs the sum
  - We can therefore implement the above in R as:
    `l<-sum(y*log(pnorm(xb))+(1-y)*log(1-pnorm(xb)))`

Application: maximum likelihood

- The log-likelihood for $\beta = [-11.9, 0.79]$ (the values R found for us).

```
# load data
df<-read_csv("example_data1.csv")
# y variable
y<-df%>%select(summercamp)%>%as.matrix()
# x variable
X<-df%>%select(parental_lincome)%>%mutate(constant=1)%>%as.matrix()
# xb (note constant is given last)
xb<-X%*%c(0.79,-11.9)
# log likelihood
l<-sum(y*log(pnorm(xb))+(1-y)*log(1-pnorm(xb)))
# return value
l
```

```
## [1] -300.0516
```

- Okay, but how do we know this is maximized? Let's evaluate the log likelihood value for various values of beta. To do this we:
  1. Wrap the likelihood expression in a function.
  2. Loop over the function and use different values.

1. Our likelihood function

```
my_loglikelihood <- function(y,x,beta,data) {
  # specify dataset
  df<-data
  # extract the dependent variable and the covariates
  yvar<-df%>%select(y)%>%as.matrix()
  Xvar<-df%>%select(x)%>%mutate(constant=1)%>%as.matrix()
  # beta
  betavec=beta
  # xb (note constant is given last)
  xb<-Xvar%*%betavec
  # log likelihood
  l<-sum(yvar*log(pnorm(xb))+(1-yvar)*log(1-pnorm(xb)))
  # return
  l
}
# load data
mydataset<-read_csv("example_data1.csv")
# try our new function
my_loglikelihood(x="parental_lincome",y="summercamp",
                 beta=c(0.79,-11.9) ,data=mydataset)
```

```
## [1] -300.0516
```

2. Loop over values

- Simplification: keep beta 0 constant at -11.9 and vary beta1

```
# loop over values of beta1
for (i in 0:1){
  print(my_loglikelihood(x="parental_lincome",y="summercamp",
                    beta=c(i,-11.9),data=mydataset))
}
```

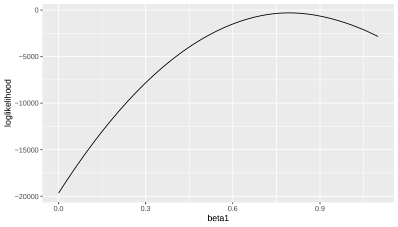
```
## [1] -19664.97
## [1] -1503.169
```

- Okay, that works, but 2 values are a bit boring. Let's try more!

Application: maximum likelihood

- Let's loop over more values, store all results in a data frame and show the likelihood in a chart as a function of beta 1:

```
# create empty matrix to store values
df<-data.frame(beta1=seq(0,1.1,by=0.01),loglikelihood=NA)
# loop over values of beta1
for (i in 1:nrow(df)){
  beta1<-df[i,1]
  df[i,2]<-my_loglikelihood(x="parental_lincome",y="summercamp",
                beta=c(beta1,-11.9),data=mydataset)
}
# let's plot the log likelihood as a function of beta 1
ggplot(df,aes(x=beta1,y=loglikelihood))+geom_line()
```



Application: maximum likelihood

- The chart indicates that the maximum value of the log-likelihood function could be around 0.79 (as the R built-in function suggest).
- But how do we find the exact values? And how about β_0?
- We use a built-in optimizer. An function to maximize or minimize to mimize an expression.
- One such function is `optim()`:

```
# use the R function optim to optimize ll
optim(c(0,0), my_loglikelihood, control=list(fnscale = -1),
      x="parental_lincome",y="summercamp",data=mydataset)
```

```
## $par
## [1]   0.7960093 -11.9392406
##
## $value
## [1] -299.6738
##
## $counts
## function gradient
##       89       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Functions, control structures, loops - summary

- We **define functions** using the following syntax:

```
function_name <- function(arg_1, arg_2, ...) {
   function body
}
```

- We control the flow of our function using control structure:

```
if (logical test){
  action to do if logical test evaluates to true
}
else {
  action to do if logical test evaluates to false
}
```

- We repeat statements using loops

```
for (x in range){
  action to repeat for all values in range
}
```

- We can combine these tools and implement a maximum likelihood estimator