# Introduction to R

Intensive Statistics Course

Wihan Marais

wihanmarais@sun.ac.za | github.com/WihanZA

25-26 January 2022

## Why R?

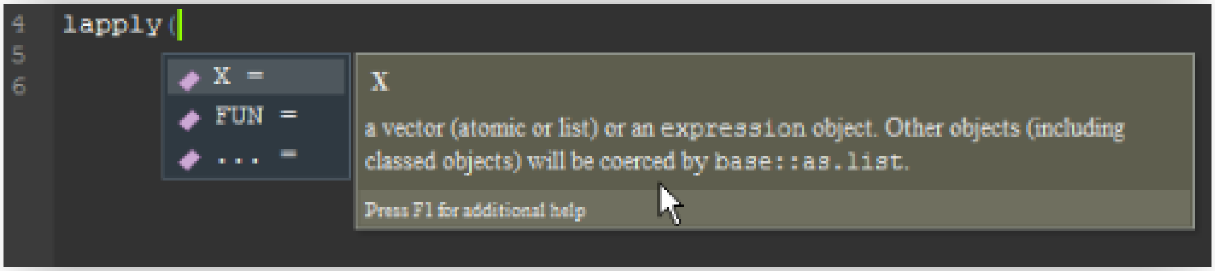
1. R is free and open-source.

    - At the time of writing, a new Stata 17 annual license is priced between R11,670 and R21,280 excluding VAT.
    - Free upgrades, updates and dissemination.
    - Availability of helpful resources like **stackoverflow**.

2. R uses **packages**

    - R consists of Base-R coupled with third-party libraries of pre-written code, or packages.
    - **CRAN**, or The Comprehensive R Archive Network, is a network of ftp (file transfer protocol) and web servers around the world that store identical, up-to-date, versions of code and documentation for R.
    - More on this later.

3. R uses predictive coding (Ctrl/Cmd + Space is very useful).



```
lapply(
```

X

a vector (atomic or list) or an expression object. Other objects (including classed objects) will be coerced by base::as.list.

Press F1 for additional help

4. R is compatible with Markdown.

    - These lecture notes were created as a '.Rmd' file using **R Markdown**, RStudio's native authoring framework for data science.
    - See this 1-minute video summary of what R Markdown entails.

## Before we start

You need the following installed on your machine:

- **R** or Base-R.

  ”R is a freely available language and environment for statistical computing and graphics which provides a wide variety of statistical and graphical techniques: linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc.”

- **RStudio**

  ”RStudio is an integrated development environment (IDE) for R. It includes a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging and workspace management.”

- **Rtools**

  Select the Rtools download link for the relevant version of R installed on your machine. To determine the version currently installed, run the following code in your console. First, highlight the line of code you would like to run and Ctrl/Cmd + Enter to run.

```
sessionInfo()[1]$R.version$version.string

# IMPORTANT:
# Take care to check the box to have the installer 'edit your path'
```

To verify that we have installed Rtools properly, we need to make use of the `devtools` package.

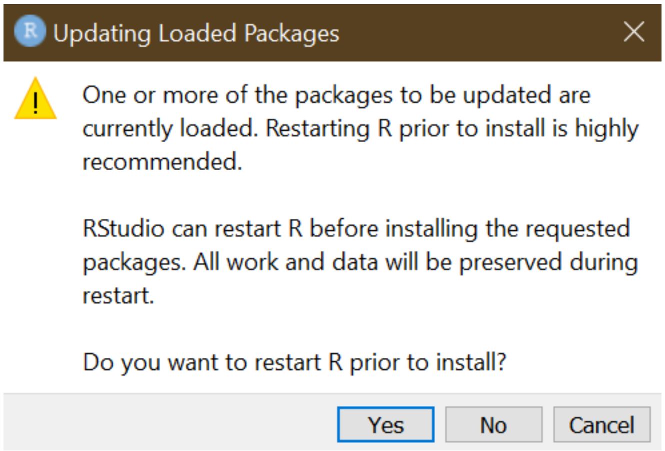
```
install.packages("devtools") # Install the package from CRAN.
library(devtools) # Load package into your current library.

find_rtools() # Run this command from the devtools package
# or
devtools::find_rtools()
# should return TRUE in your console
```

## Packages

Install and load a few packages that you would likely use often. Let’s use the `installr` package as an example.

```
# Packages need only be installed once
install.packages("installr")
# and can be loaded into your library with
library(installr)
```

However, if I have already installed `installr` before, `install.packages()` produces the following error:



One or more of the packages to be updated are currently loaded. Restarting R prior to install is highly recommended.

RStudio can restart R before installing the requested packages. All work and data will be preserved during restart.

Do you want to restart R prior to install?

It can be hard to keep track of all the packages that you have or have not installed on your machine. How, then, should we install packages?

```
# In short: if package has not yet been installed, run code to install
if (!require(installr)) {
    install.packages("installr")
    require(installr)
}
```

Instead, I propose using the `pacman` package. It enables us to easily install new- and load old packages from curated lists such as CRAN, or any open-source package from **GitHub** using `p_load()` and `p_load_gh()`, respectively. These commands install packages if they have not yet been installed, and subsequently load them into our library.

```
# Installs pacman from CRAN.
if (!require(pacman)) {
    install.packages("pacman")
    require(pacman)
}
# Load pacman into our library.
library(pacman)

# And finally...
pacman::p_load(installr)
```

Why did we load `installr` in the first place?

```
# Are you using the latest version of R?
check.for.updates.R()
# Download and run the latest R Version.
install.R()
# Copy your packages to the newest R installation
copy.packages.between.libraries()
```

What are the packages we need to install and load?

```
# From CRAN
pacman::p_load(fixest, tidyverse, huxtable, hrbrthemes, modelsummary, glue)

# From GitHub
pacman::p_load_gh("karthik/wesanderson", "BlakeRMills/MetBrewer")
                # "profile/repository name"
```

## Basics

### Projects

To improve your workflow in R, it is essential to work from a R project or directory.

`File > New project > New directory > New project > Choose directory location and name`

If you are planning on applying version control to your new project, it is useful to check `Create git repository`. Once you have created a project, its location will serve as your 'root folder' or reference directory for any future operations performed within this project. It is advisable to always reopen a R session by clicking the relevant `.Rproj` file, as it will keep track of your most current workspace and variable environment.

### Workspace

When operating from a new project, you ought to observe a workspace divided into quadrants as summarised in Table 1. As with Stata, code can be typed and executed directly from your console (previously "Command"). Alternatively, this code can be stored in—and later relied upon—a script or `.R` file (like Stata do-file).

Table 1: Workspace

| **Location** | **Function** |
|---|---|
| Top left | Script/R Markdown |
| Top right | Global environment |
| Bottom left | Console |
| Bottom right | Files/Plots/Packages/Help |

R is an object-orientated language. Objects of various types (scalars, matrices, data frames, vectors, etc.) can be stored in memory for later use. Once named and saved, these objects will appear in your global environment. In R, we use the assignment operator `<-` to name and save objects (Tip: **Alt/Option** + **-**). For example:

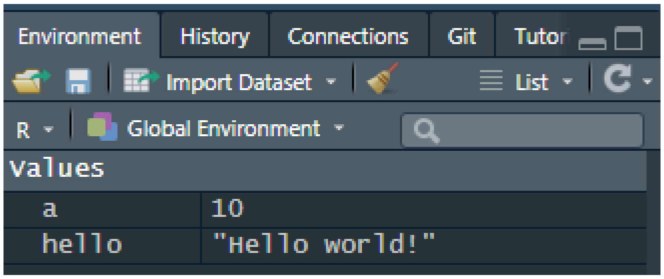
```
# object name <- value(s)
a <- 10
hello <- "Hello world!"

# If you want to determine the type/class of an object
class(a)
```

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

```
class(hello)
```

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

By highlighting and executing the relevant code (Ctrl/Cmd + Enter), `a` and `hello` should appear in your global environment like this:



| | |
|---|---|
| a | 10 |
| hello | "Hello world!" |

We are able to report these variables in our console (or R Markdown output) by running the following:

```
a
```

```
## [1] 10
```

```
# or
print(hello)
```

```
## [1] "Hello world!"
```

```
# or by using the glue package for something more fancy
glue::glue("I saved a variable which contains {hello} and I stored the number {a}.")
```

```
## I saved a variable which contains Hello world! and I stored the number 10.
```

Finally, your directories can be viewed to the bottom right, in addition to plot outputs, currently loaded packages, and help files. Should you ever require help or additional information regarding a specific command, add a `?` before that command and run the code. For example:

```
?glue::glue()
```

### Arrays

In R, an "array" object is basically a vector of values in the form of integers, doubles, string, etc. An array can be created by using the **concatenate** function or `c()`.

```
x <- c(1, 2, 3)
y <- c(4, 5, 7)
z <- c(7, 8, 9)

# Useful functions to perform on arrays/vectors
```

```
sum(x)
min(x)
median(x)
summary(x)
# the latter provides a summary of the functions above
```

### Data frames

Data frames are the workhorse of statistical analysis in R. Data frames are essentially tables of data consisting of rows and columns. Both rows and columns can also be assigned headings. They can be constituted in a variety of ways.

```
# data.frame() can create columns from arrays and assign column names
df_1 <- data.frame(A = x, B = y, C = z)
# arrays must be of the same length!
```

Specific rows, columns, and cells can be referenced as follows:

```
# Return column "A" as a vector
df_1$A # "$" preceding the name of column "A"
df_1[ , 1] # [all rows, column = 1] - empty reference implies all
df_1 %>% pull(A) # Ctrl/Cmd + Shift + M for %>%

# Return row 2 as single row data frame
df_1[2, ]

# Return row 2-3 as two row data frame
df_1[2:3, ]

# Return cell in row 2 column 1
df_1[2, 1]

# Create a new column "D" that is the sum of A and B
df_1$D <- df_1$A + df_1$B # notice the assignment operator
# or
df_1 <- df_1 %>% mutate(D = A + B)
```

As you can see, there's more than one way to skin a cat in R.

### Reading and writing data

In practice, you will likely be loading data from external files, such as a `.csv` file. For instance, create a data frame from the external file `Ireland_energy.csv` which contains Ireland's energy consumption data for 1980-2018. Do the same for the corresponding file for Ireland's population, `Ireland_population.csv`.

```
ire_energy <- read.csv(file = "data/Ireland_energy.csv", header = TRUE)
# "file" refers to the file path originating from your root directory
# "header" is set to true because the .csv file contains column headings

ire_pop <- read.csv(file = "data/Ireland_population.csv", header = TRUE)
```

Merge the two data frames to create a single data frame. Subsequently, create a new variable (or column) for the natural logarithm of Ireland's per capita energy consumption.

```r
ireland_df <- merge(x = ire_energy, y = ire_pop, by.x = "Year", by.y = "Year") %>%
  # Create new data frame ireland_df
  # merge() allows specification of two constituent data frames (x & Y)
  # as well as their common column on the basis of which matching occurs (by.x & by.y)

  mutate(ln_energy_pc = log(GJ/Population)) %>%
  # tidyverse piping ( %>% )
  # mutate(new_variable_name = transformation with existing columns)
  # log()'s defualt setting implies natural logarithm

  mutate(Year = glue::glue("{Year}-01-01"), Year = as.Date(Year,  format = "%Y-%m-%d"))
  # Using glue to code years as e.g. 1980-01-01 instead of 1980
  # Render new Year into a date format
```

Table 2: Ireland's energy consumption

| Year | GJ | Population | ln_energy_pc |
|---|---:|---:|---:|
| 2018-01-01 | 695195813.70 | 4876547 | 4.96 |
| 2017-01-01 | 674283626.40 | 4813138 | 4.94 |
| 2016-01-01 | 665078014.40 | 4762595 | 4.94 |
| 2015-01-01 | 632644273.00 | 4708040 | 4.9 |
| 2014-01-01 | 603311823.70 | 4662713 | 4.86 |

We can now write this data frame back to a `.csv` file.

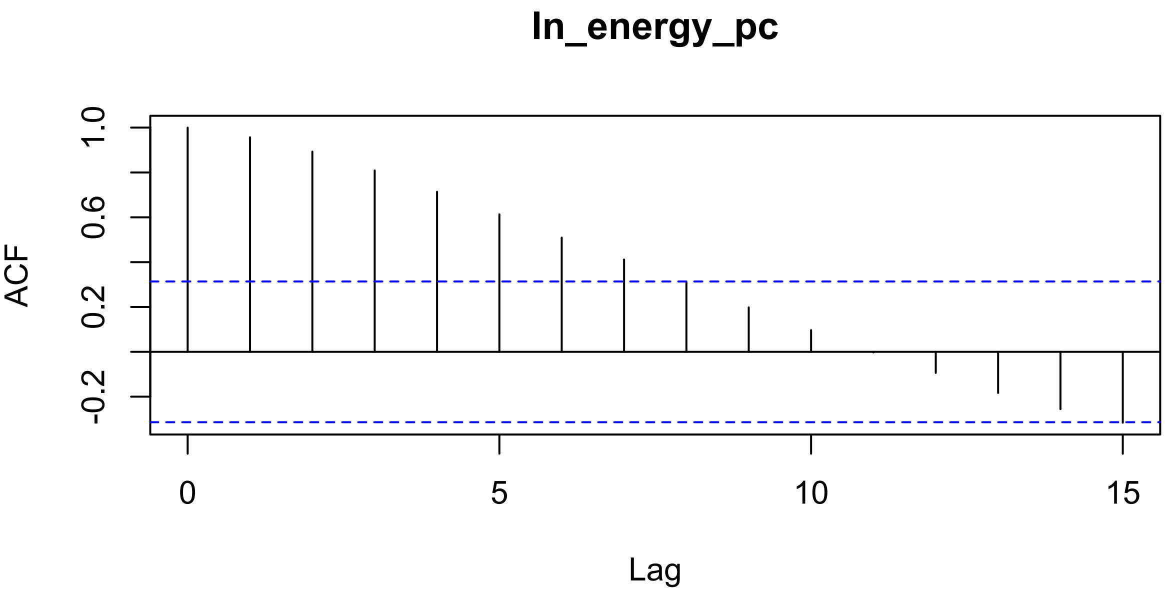
```r
write.csv(ireland_df, file = "data/ireland_complete.csv", append = FALSE,
          col.names = TRUE, row.names = FALSE)
# data frame to be written to csv
# file path
# "append" set to FALSE to imply that any file with the same name will be overwritten
# "_.names" set TRUE if we have assigned names
```

## Time series analyses

Let us perform some time series analysis on our data frame. Compute and plot the autocorrelation- and partial autocorrelation function of Ireland's per capita energy consumption. For this we need the `stats` package.

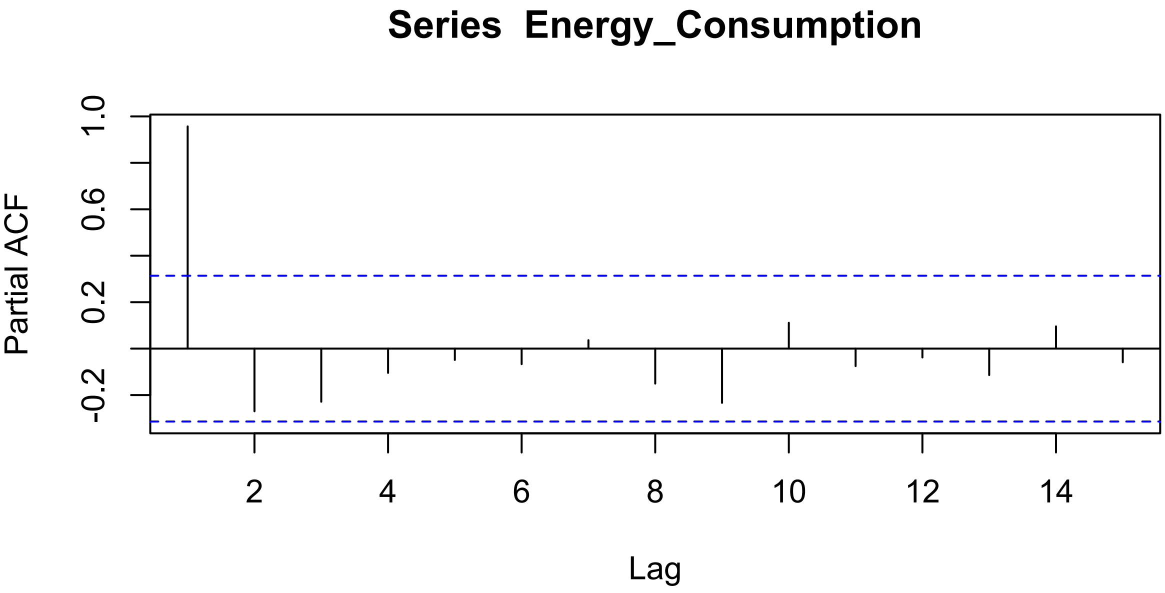
```r
pacman::p_load(stats)

# Confirm that data is chronological
ireland_df <- ireland_df %>% arrange(Year)
```

```r
# Isolate the column - notice the use of dplyr::
Energy_Consumption <- ireland_df %>% dplyr::select(ln_energy_pc)

# Use acf() and store our ACF and PACF in memory
my_acf <- stats::acf(
  x = Energy_Consumption,
  plot = T, # automatically create a plot
  type = "correlation") # we want the standard AF
```



ln_energy_pc

```r
my_pacf <- stats::acf(
  x = Energy_Consumption,
  plot = T,
  type = "partial") # we want the PACF option
```

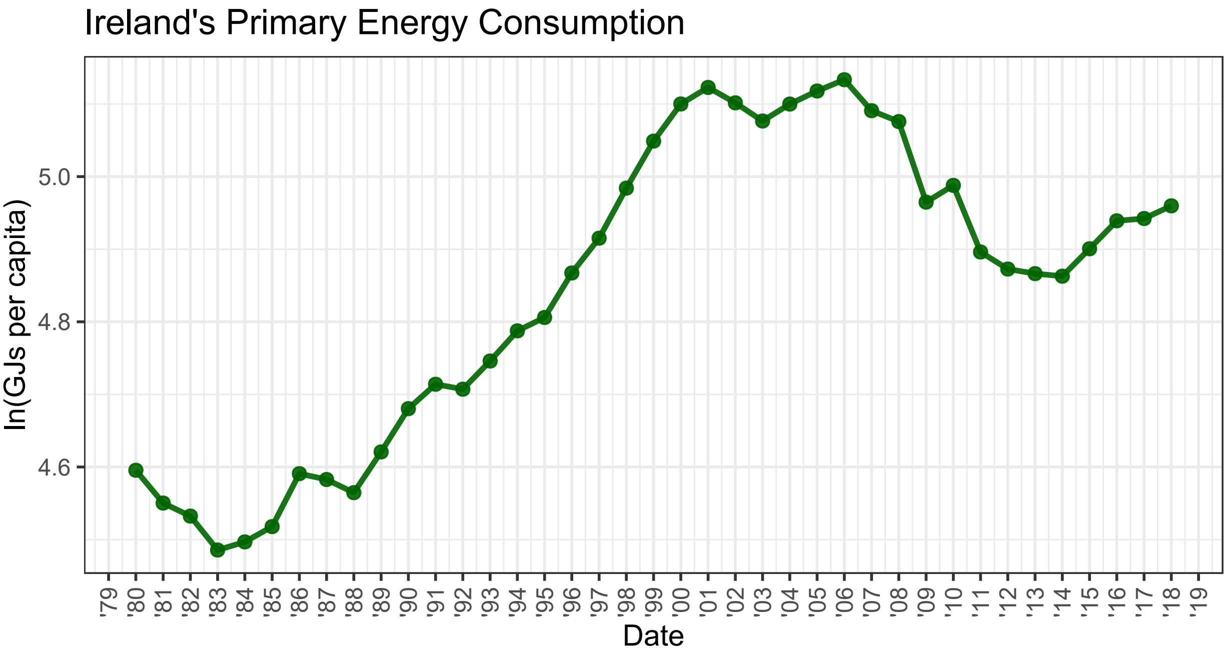


Series Energy_Consumption

```
# Remember to unload this package - Why?
pacman::p_unload(stats)
```

I have a strong preference for ggplot's plotting capabilities. Data visualisation is made easy in R, however it deserves an entire lecture in and of itself. As many of the functions you will be using, ggplot is a part of the so-called tidyverse. To illustrate, let us make a line graph of our variable of interest.

```
ireland_df %>% ggplot() + # creates the 'canvas'
  theme_bw() + # choose on of many existing themes
  geom_line(aes(x = Year, y = ln_energy_pc), size = 1, alpha = 0.9, color = "darkgreen") +
  # creates the line on the canvas with aes() coordinates
  geom_point(aes(x = Year, y = ln_energy_pc), size = 2, alpha = 0.9, color = "darkgreen") +
  # similarly for points
  scale_x_date(date_labels = "'%y", date_breaks = "year") +
  # make x axis labels
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust=1)) +
  # rotate x axis labels
  labs(title = "Ireland's Primary Energy Consumption", y = "ln(GJs per capita)", x = "Date")
```



```
  # add axis titles and main title
```

Does this time series look stationary to you? Use the urca package to perform Augmented Dickey-Fuller (ADF) tests with the command ur.df().

```
pacman::p_load(urca)

# Currently, Energy_Consumption is still a data frame and y should be a vector
test_vector <- Energy_Consumption %>% pull(ln_energy_pc)

my_ADF1 <- ur.df(
  y = test_vector, # vector
  type = "trend", # type  of ADF - trend + constant
  lags = 5, # max lags
  selectlags = "AIC" # lag selection criteria
  )

# use Base-R's summary() to present ADF object
summary(my_ADF1)
```

```
##
## ###############################################
## # Augmented Dickey-Fuller Test Unit Root Test #
## ###############################################
##
## Test regression trend
##
##
## Call:
## lm(formula = z.diff ~ z.lag.1 + 1 + tt + z.diff.lag)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.087796 -0.019856  0.009542  0.029923  0.045227
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.3937065  0.2278056   1.728   0.0950 .
## z.lag.1     -0.0797608  0.0498511  -1.600   0.1208
## tt           0.0001319  0.0010831   0.122   0.9039
## z.diff.lag1  0.1723326  0.1771983   0.973   0.3391
## z.diff.lag2  0.3115141  0.1794628   1.736   0.0936 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.03842 on 28 degrees of freedom
## Multiple R-squared:  0.296,  Adjusted R-squared:  0.1954
## F-statistic: 2.943 on 4 and 28 DF,  p-value: 0.03778
##
##
## Value of test-statistic is: -1.6 1.6437 2.1039
##
## Critical values for test statistics:
##       1pct  5pct 10pct
## tau3 -4.15 -3.50 -3.18
## phi2  7.02  5.13  4.31
## phi3  9.31  6.73  5.61
```

Is the null hypothesis rejected? How about an ADF test that specifies only a constant/drift?

```
my_ADF2 <- ur.df(
  y = test_vector,
  type = "drift", # type  of ADF - with drift
  lags = 5,
  selectlags = "AIC"
  )

summary(my_ADF2)
```

```
##
## ###############################################
## # Augmented Dickey-Fuller Test Unit Root Test #
## ###############################################
##
## Test regression drift
##
##
## Call:
## lm(formula = z.diff ~ z.lag.1 + 1 + z.diff.lag)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.088270 -0.019198  0.009114  0.030175  0.044932
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.37689    0.17811   2.116   0.0430 *
## z.lag.1     -0.07569    0.03633  -2.083   0.0461 *
## z.diff.lag1  0.16583    0.16608   0.999   0.3263
## z.diff.lag2  0.30364    0.16454   1.845   0.0752 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.03776 on 29 degrees of freedom
## Multiple R-squared:  0.2956, Adjusted R-squared:  0.2228
## F-statistic: 4.057 on 3 and 29 DF,  p-value: 0.01595
##
##
## Value of test-statistic is: -2.0834 2.5446
##
## Critical values for test statistics:
##       1pct  5pct 10pct
## tau2 -3.58 -2.93 -2.60
## phi1  7.06  4.86  3.94
```

## Cross section analyses

We loaded the `fixest` and `huxtable` packages for a good reason. The packages ease the estimation and presentation of a host of standard and complex regression models. For good measure, we can load them again with `pacman::p_load(fixest, huxtable)`. Before we explore some examples, we need a data frame. `.csv` files can be downloaded directly from the internet and stored as a data frame as before.

```
# We replace a local file path with a web address
cs_df <- read.csv(
  'https://raw.githubusercontent.com/stata2r/stata2r.github.io/main/data/cps_long.csv')
# We do not need to specify header = TRUE as this is also the default
# Explore the new data frame
view(cs_df)
```

Let us consider how to perform standard OLS regressions with `fixest`'s `feols()` function. At the very least, this function requires two arguments, A) a formula in the format `y ~ x1 + x2` and B) a data set/table/frame or `data`. For example, what bearing does education have on wages of those observations for 1974?

```
# First, subset the data to those observations for 1974
new_cs_df <- cs_df %>% filter(year == 1974)

# Our first model
model1 <- feols(wage ~ educ, data = new_cs_df)

# Adding an explanatory continuous variable
model2 <- feols(wage ~ educ + age, data = new_cs_df)

# Adding a categorical variable
model3 <- feols(wage ~ educ + age + i(marr), data = new_cs_df)

# Print your models using huxtable's huxreg()
# List the models you would like to appear in the table
huxreg(model1, model2, model3)
```

|  | (1) | (2) | (3) |
|---|---|---|---|
| (Intercept) | 10280.103 *** | -4194.788 *** | -4175.649 *** |
|  | (324.550) | (381.879) | (365.526) |
| educ | 310.679 *** | 493.332 *** | 439.230 *** |
|  | (26.247) | (23.960) | (22.977) |
| age |  | 369.539 *** | 256.760 *** |
|  |  | (6.228) | (6.650) |
| marr::1 |  |  | 6152.185 *** |
|  |  |  | (160.809) |
| N | 15992 | 15992 | 15992 |
| R2 | 0.009 | 0.188 | 0.256 |
| logLik | -169209.949 | -167618.468 | -166918.049 |
| AIC | 338423.898 | 335242.937 | 333844.097 |

*** $p < 0.001$; ** $p < 0.01$; * $p < 0.05$.

You can find guides to the `huxtable` package and the `huxreg()` function here and here, respectively. `huxreg()` is highly adaptable and easy to use. For example:

```
huxreg("Pretty 1" = model1, "Pretty 2" = model2, "Pretty 3" = model3,
       # choose statistics to display
       statistics = c("N" = "nobs", "R-squared" = "r.squared"),
       # change stars criteria
       stars = c(`*` = 0.1, `**` = 0.05, `***` = 0.01, `****` = 0.001),
       # rounded to 2 decimals
       number_format = 2
       ) %>%
# The following are huxtable commands and are also piped
set_font_size(8) %>%
set_align(10:11, 2:4, value = "center") %>%
set_caption("My pretty regressions")
```

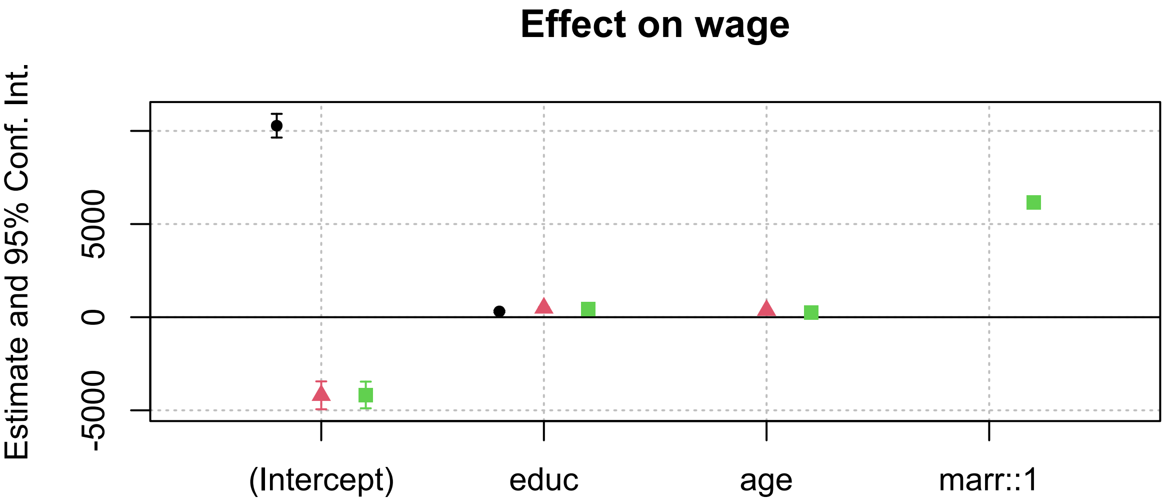
Table 3: My pretty regressions

| | Pretty 1 | Pretty 2 | Pretty 3 |
|---|---|---|---|
| (Intercept) | 10280.10 **** | -4194.79 **** | -4175.65 **** |
| | (324.55) | (381.88) | (365.53) |
| educ | 310.68 **** | 493.33 **** | 439.23 **** |
| | (26.25) | (23.96) | (22.98) |
| age | | 369.54 **** | 256.76 **** |
| | | (6.23) | (6.65) |
| marr::1 | | | 6152.18 **** |
| | | | (160.81) |
| N | 15992 | 15992 | 15992 |
| R-squared | 0.01 | 0.19 | 0.26 |

**** p < 0.001; *** p < 0.01; ** p < 0.05; * p < 0.1.

`fixest` also contains plotting functions that create, among others, coefficient plots and interaction plots. For example:

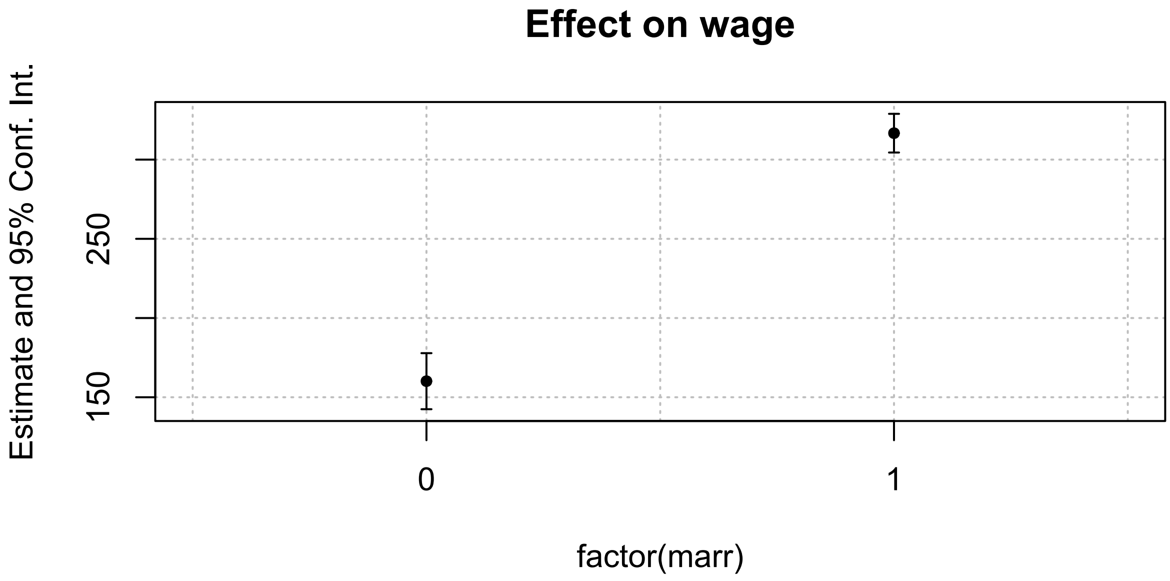
```
# Combine the coefficient plots for each model
# Notice that models need to be entered as a list() object
coefplot(list(model1, model2, model3))
```



```
# Now create a new model with an interaction term
model4 = feols(wage ~ i(factor(marr), age), data = new_cs_df)
# factor() tells the model that marr categorical
# so does i() unless it contains more than one variable
# then it indicates an interaction

# What do the results say?
summary(model4)
```

```
## OLS estimation, Dep. Var.: wage
## Observations: 15,992
## Standard-errors: IID
##                      Estimate Std. Error t value  Pr(>|t|)
## (Intercept)          4649.710  228.56675 20.3429 < 2.2e-16 ***
## factor(marr)::0:age   160.149    9.02489 17.7453 < 2.2e-16 ***
## factor(marr)::1:age   316.684    6.22663 50.8596 < 2.2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## RMSE: 8,517.8   Adj. R2: 0.207631
```

```
# Show how age's effect on wages differs by marriage status
iplot(model4)
```


Effect on wage
Estimate and 95% Conf. Int.
150
250
0
1
factor(marr)

## Acknowledgements and further reading

Lecture notes are compiled from the following resources:

- *R intro* (2018) by Grant R. McDermott and Ed Rubin.
- *Data Science for Economics and Finance: Getting you staRted* (2021) by N.F. Katzke.
- Stata2R

Should you need additional resources to get started, try the following:

- Quick-R
- Data Science Programming Methods (STAT 447) by Dirk Eddelbuettel (University of Illinois)
- RStudio Cheatsheets
- Data Science for Economists (EC 607) by Grant McDermott (University of Oregon)
- Use your student credentials to sign up for a GitHub Pro account.
- Download GitHub Desktop for free and use version control for all your projects.
- Wonderful data visualisation guide with R code, Data to Viz.