Reproducible Research in R

Dries Debeer & Benjamin Becker

29. and 30. September 2022

FDZ Autumn Academy

Introduction

Introduction

Who are we?

Dries Debeer

Statistical Consultant at Ghent University (FPPW)

scDIFtest, permimp, eatATA, mstDIF

dries.debeer@ugent.be

Benjamin Becker

Researcher at IQB (Verbund Forschungsdaten)

eatGADS, eatDB, eatATA, pisaRT

b.becker@iqb.hu-berlin.de

Who are you?

1. Occupation, employer?
2. Previous knowledge and experience
   - with reproducible research?
   - with R?
3. Specific interest/motivation for this workshop?

Agenda

Day 1

- Conceptual Introduction
- Writing Reproducible R Code
- (Alternative) Reproducible Workflows

Day 2

- RMarkdown
- Version Control/git

Reproducible Research

Reproducibilty & Replicability

Reproducibility

- Same research question
- Same analysis
- **Same** data

=> *Same results*

Replicability

- Same research question
- Same analysis
- **New** data

=> *Same results*

Reproducibilty

The **replicability** crisis in psychological* research is partly caused by **reproducibility** issues.

- more than 70 percent of the researchers that tried to reproduce other scientist's experiments failed (Baker, 2016).
- more than 50 percent of the researchers that have tried to reproduce their own experiments failed (Baker, 2016).

*Psychology, educational sciences and social work

Baker, M. (2016). Reproducibility crisis. Nature, 533(26), 353-66.

Reproducibility

Threats to self-replication (see also Peikert et al., 2021)

- Mistakes in data preparation/analysis (e.g., syntax bugs)
- Inconsistent versions of code, data, or both
- Copy-and-paste errors
- ...

Peikert, A.,van Lissa, J., Brandmaier, A.M. (2021). Reproducible Research in R: A Tutorial on How to Do the Same Thing More Than Once. Psych 2021, 3, 836–867.

Reproducibility

Threats to replication of others

- ...
- Data not available
- Incomplete documentation/reporting
- ...

Reproducibility

Proposed solutions (see also Peikert et al., 2021)

- Clean code
- Version control
- Dynamic document generation
- (Independent reproduction)

Reproducibility

Workshop Goals

Be able to ...

- ... reproduce your own research (i.e., analyses)
- ... publish your code openly with your research
- ... create research which is reproducible for others
- ... spread the word

RStudio setup

RStudio setup

1. Save the course content to a directory on your machine
2. Open RStudio
3. Choose `File < New Project ...`
4. Choose `Existing Directory`
5. Browse to the directory on your machine where you saved the course content and select the "R-programming" folder as the `Project working directory`
6. Click `Open in new session`
7. Click `Create Project`

RStudio setup - optional

1. Choose `Tools < Global options`
2. Under `General`
   - DON'T `Restore .RData into workspace at startup`
   - NEVER `Save workspace to .Rdata on exit:`
   - Save the code instead!
   - Use `saveRDS()` and `readRDS()` for objects that require a long time to computate
3. Further personalize RStudio

Writing Reproducible R Scripts

Reproducible Workflow in R & Rstudio

Key Principles:

1. Save your code
2. Work in projects
    - Use multiple scripts
    - Use multiple projects
3. Frequently and completely restart

1. Save your code

- Save the code for preprocessing/data manipulation (don't save intermediate data sets)
- Save the code for visualizations (don't save the plots)
- Save the code for analyses (don't save the results)

May the source code be with you!

1. Save your code

- Save the code for preprocessing/data manipulation (don't save intermediate data sets)
- Save the code for visualizations (don't save the plots)
- Save the code for analyses (don't save the results)

May the source code be with you!

- Use `saveRDS()` and `readRDS()` for objects that require a long time to compute

1. Save your code

Within RStudio

1. Choose `Tools < Global options`
2. Under `General`
    - DON'T `Restore .RData into workspace at startup`
    - NEVER `Save workspace to .Rdata on exit:`
    - Save your script instead!
    - Use `saveRDS()` and `readRDS()` for objects that require a long time to compute
3. You can further personalize RStudio
    - Visually (`Tools < Global options < appearance`)
    - Keyboard shortcuts (`Tools < Modify Keyboard Shortcuts...`)
    - ...

1. Save your code

A typical workflow:

- Write your code in scripts in text editor.
- Execute lines (or chunks) of code by sending them to the R console.
- Add comments using the hash tag #
- Add sections to long scripts (or use multiple scripts)
    - No lines of code are lost when R is shut down or crashes
    - Assures reproducible code/coding
    - Makes it easy to share your code with colleagues or reviewers
    - ...

2. Work in projects

What is a project?

- One designated folder containing all files related to a single (research) project.
- When necessary, add sub folders for ...
    - data
    - R scripts
    - figures and graphs
    - manuscripts
    - presentations
    - ...
- The folder contains all relevant files, nothing more.

2. Work in projects

The idea of a project is formalized in an RStudio project.

- Technically it is a small text file with (.Rproj) extension,
- which is associated with RStudio.
- RStudio recognizes the “parent folder” of this file as the project folder

2. Work in projects

Within an RStudio project...

- the project folder is automatically set as the *working directory*.
  try `getwd()` within a project
- **Don't use**

```
setwd("path\to\your\local\folder")
```

2. Work in projects

Use relative paths for reading and writing data. Avoid ...

```
cannot open file 'path\that\only\works\on\my-computer': No such file
or directory
```

- With relative paths, code works when ...
  - the project folder is moved (or renamed)
  - you are working on a shared drive
  - you send your project in a ZIP-folder
- useful functions:
  - `?list.dirs`
  - `?list.files`
- (Or use the here-package)

2. Work in projects

Keep it clean!

- One folder per project
- No outside code
- No outside computing
- Use separate RStudio projects for each project
- Use separate Rstudio instances for each Rstudio project

2. Work in projects

How to create an Rstudio project?

1. Open Rstudio
2. Choose `File < New Project ...`
3. Choose `Existing Directory`
4. Browse to the directory on your machine where you saved the course content and select the "example-project-2" folder as the `Project working directory`
5. Click `Open in new session`
6. Click `Create Project`

2. Work in projects

Within a project, use multiple source files (scripts) for:

1. Preprocessing data
2. Descriptive analysis
3. Building models
4. Hypothesis testing
5. Making graphs
6. ...

3. Frequently and Completely Restart

Regularly completely restart your session.
Don't use

```r
rm(list = ls())
```

- start clean, every time
- restart to make sure everything reproduces
- `.rs.restartR()`
- `Ctrl + Shift + F10`

How To?

How to … read data

To analyse data, you have to *read* the data from some file (or connection) and make it an *object* in R.

Almost any type of file can be read by R, via specific functions and packages (`reader`, `haven`, `readxl`, …).

- .txt → `read.table()`
- .csv → `read.csv()`, `read.csv2()`
- .xls → `readxl::read_xls()`
- .xlsx → `readxl::read_xlsx()`
- .sav → `haven::read_sav()`
- .por → `haven::read_por()`
- .sas → `haven::read_sas()`
- …

How to ... preproces data

Useful functions in **Base R**

- ?'[' and ?'[['
- ?merge, ?reshape
- ?apply, ?tapply, ?aggregate,...
- ?sort, ?order
- ...

Useful functions in dplyr

- ?filter, ?select, ?slice
- ?mutate, ?rename
- ?group_by, ?summarize, ?ungroup
- ?arrange
- ?inner_join, ?left_join, ...

How to ... preproces data

For fast and efficient data wrangling with VERY big data, the `data.table`-package can be helpful.

How to ... preproces data

Never change raw data!

If you made some mistakes while preprocessing raw data → change your code and re-run it.

Note that:

- R “reads” the data and loads it in the work space.
- Hence, manipulating data within R(Studio) does not change the data on your machine. Only the loaded data within the work space is changed.

How to ... write data

When the data (a data.frame, a fitted model, ..) you want to save is for use in R only, use `saveRDS()` `readRDS()`.

When the data (a data.frame) is for use by software, several options are available:

- .txt → `write.table()`
- .csv → `write.csv()`, `write.csv2()`
- .sav → `haven::write_sav()`
- .por → `haven::write_por()`
- .sas → `haven::write_sas()`
- ...

WARNING:

R does not prompt a warning when you are about to overwrite an existing file.

How to ... save plots

Only use the RStudio “Plot”-window for interactive plot making. Don’t use it for saving plots.

For reproducible figures, use `pdf()`, `png()`, `jpeg()`, `tiff()`, ... instead. See `?jpeg`.
Adjust size and aspect ratio using the arguments:

- `width = ...`
- `height = ...`
- `units = ...` → the units for the `width` and `height` arguments
- ...

How to ... save plots

```
z_values <- rnorm(1e+4)
sig_z_values <- z_values[pnorm(abs(z_values),
                               lower.tail = FALSE) < 0.025]
pdf("name_for_this_figure.pdf", width = 10, height = 7)
{
  # all the code that creates the figure
  hist_data <- hist(z_values,
                    main = "Significant Z-values \n(stupid plot)",
                    xlab = "Z-values")
  hist(sig_z_values,
       breaks = hist_data$breaks,
       add = TRUE,
       col = "skyblue")
}
dev.off()
```

Alternative Reproducible Workflows

LaTeX

Typesetting System

- Engines (pdflatex, xelatex, lualatex, ...)
- Distributions (OS dependent: Miktex, tinytex, ...)
- Text editors (Texlive, Texmaker, RStudio...)

LaTeX vs. Markdown

Advantages

- APA6 and APA7 document classes
- Better support for formulas
- Looks nicer
- Better customization
- Designated output format: .pdf

https://yihui.org/en/2013/10/markdown-or-latex/

LaTeX vs. Markdown

Disadvantages

- More complex
- Sometimes complicated setup
- Designated output format: .html

LaTeX and R

knitr

- substitutes `sweave`
- allows embedding R chunks in .tex code (.Rnw)
- powerful tool for automated document generation

Producing reproducible Word Tables via R

Different packages

- apaTables
- sjPlot
- stargazer

apaTables

Descriptive statistics for a data set

```
apaTables::apa.cor.table(iris,
                         filename = "iris_apatable_descr.doc",
                         landscape = FALSE)
```

apa Tables

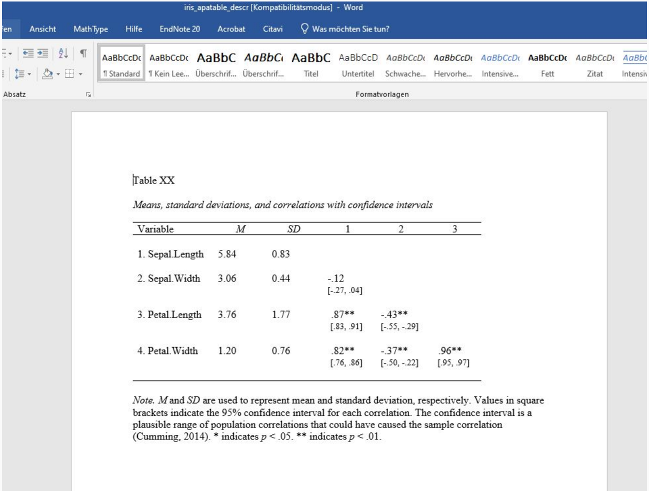


Table XX

Means, standard deviations, and correlations with confidence intervals

Variable	*M*	*SD*	1	2	3
1. Sepal.Length	5.84	0.83			
2. Sepal.Width	3.06	0.44	-.12 [-.27, .04]		
3. Petal.Length	3.76	1.77	.87** [.83, .91]	-.43** [-.55, -.29]	
4. Petal.Width	1.20	0.76	.82** [.76, .86]	-.37** [-.50, -.22]	.96** [.95, .97]

Note. *M* and *SD* are used to represent mean and standard deviation, respectively. Values in square brackets indicate the 95% confidence interval for each correlation. The confidence interval is a plausible range of population correlations that could have caused the sample correlation (Cumming, 2014). * indicates $p < .05$. ** indicates $p < .01$.

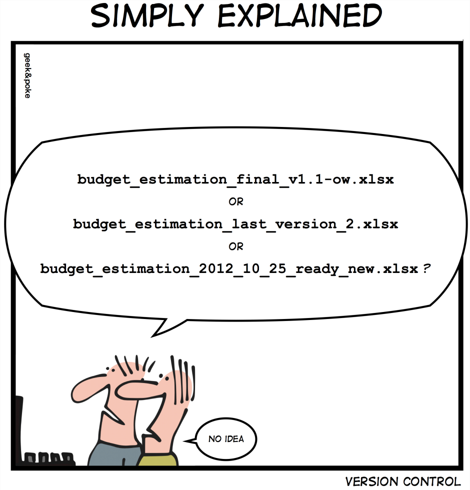
RMarkdown

Version Control via Git and Github

Version Controlling

- Motivation
- Setup
- Work Flows
- Recommendations
- Resources

Motivation


SIMPLY EXPLAINED
geek&poke
budget_estimation_final_v1.1-ow.xlsx
OR
budget_estimation_last_version_2.xlsx
OR
budget_estimation_2012_10_25_ready_new.xlsx ?
NO IDEA
VERSION CONTROL

Motivation 1

Single Author Projects

- Implementation of long term change history
  - What has been changed?
  - When was it changed?
- No ridiculous file names
- No archive sub folder
- Accessibility for others ('Open Science')
- Additional safety net
- ...

Motivation 2

Collaborations

- Who has changed what when exactly?
- Clear, current project state
- No annoying mail attachments or file-sharing platforms
- Parallel work easily possible
- Possibility of hierarchical responsibilities
- ...

But…

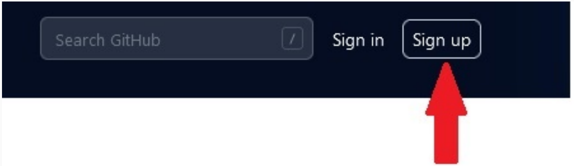

THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Prerequisits

- Git-Installation
- RStudio-Installation
  $\rightarrow$ Alternatives: Shell, Gitkraken, SmartGit, ...
- Github account
  $\rightarrow$ Alternatives: Bitbucket, Gitlab, ...
- Connect everything

Register at Github

https://github.com/

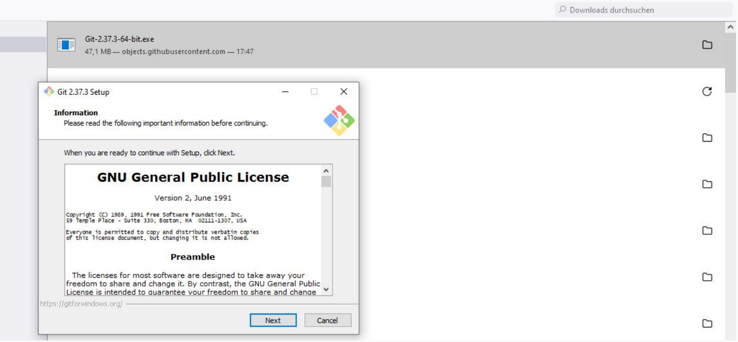


Download git

`https://git-scm.com/download/win`

Install git

Install git into a folder in which you have sufficient rights (user folder if necessary)



Configure git

git can be configured for Github via R

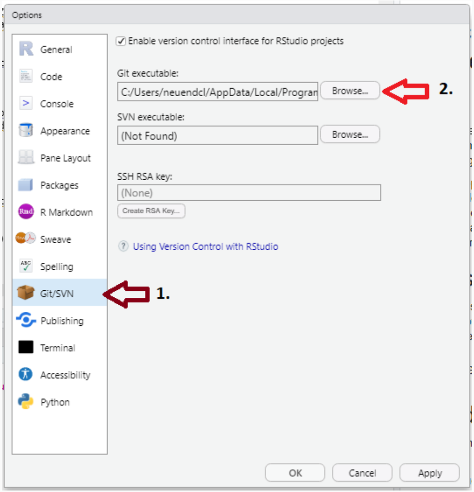
```
# if necessary, install usethis
# install.packages("usethis")

library(usethis)
use_git_config(user.name = "Username",
               user.email = "user@gmail.com")
```

Configure git

Check if the configuration was successful via the command line (cmd).

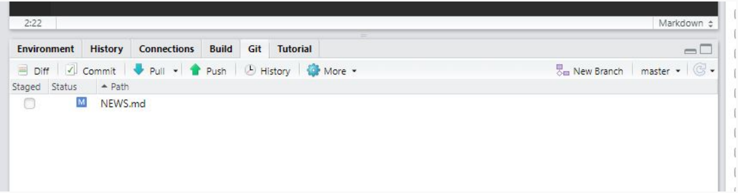
```
git config -global -list
```

Connect git and RStudio



RStudio

The RStudio user interface to Git looks like this



Work flow 1

Creating a repository

- Create an **online repository** (e.g. on Github)
  - Use an R specific `.gitignore`
  - Initialize with a short readme (`.md`)
- Clone the repository to your local machine via RStudio as a new project
- An R-Project is added automatically to the existing repository

Excursion: gitignore

- Plain text file
- Which files should not be tracked by git?
  $\rightarrow$ These then only exist locally in their current version!
- Options
  - Single files
  - Folders
  - Specific data types
  - Combinations of the above
- Use cases
  - Large files (Data, images, ...)
  - Auxiliary files (e.g. created during latex compilation)

Work flow 2

Working with an existing repository

- Before working: Synch your local repo (**Pull** or **clone**)
- Perform changes in your local repository
  $\rightarrow$ Create/modify/delete files
- **Stage** your changes
- **Commit** your changes (aka new version)
- **Push** your commit(s) (online repository is updated)

Excursion: mergen

Conflicts between different updated versions

- Common when working collaboratively
- Discrepancies between your own different local repos $\rightarrow$ Git communicates these and indicates conflicts
- Select the desired changes
- Stage selection, commit and push

Excursion: Branches

Multiple parallel versions of a project within one repository

- Common e.g. in areas like software development
- e.g. one stable and one development branch
- Only certain modifications should be made in the stable branch
- **Note**: RStudio GUI has limited support for this

Your impressions?

Recommendations

- Keep it simple!
    - If not necessary, no branches/forks/pull requests
- Have meaningful commits
- Keep it lean (no big files)
- Avoid using the Github homepage working within the repository

Resources

Git + RStudio Resources

- Small Intro (`https://r-bio.github.io/intro-git-rstudio/`)
- Happy Git with R (`https://happygitwithr.com/`)
- R Packages and Git (`https://r-pkgs.org/git.html`)

General Git Resources

- Git Book (`http://git-scm.com/book/en/v2`)

Exercises

Exercises

- Create an example repository
- Create some descriptive analysis of the `mtcars` data set
- Version control your work

Good programming practices

"Write code for humans, not for machines!"

Code Style

Invest time in writing readable R-code.

- It will make collaborations easier
- It will make debugging easier
- It will make your analyses more reproducible

There is a complete *tidyverse* style-guide `https://style.tidyverse.org/`.

Go easy on your eyes

- with spaces before and after: `-` `+` `/` `*` `=` `<-` `<` `==` `>`
- always use `<-` for assignments
- only use `=` in function calls
- use indentation (largely automatic in RStudio)
- `CamelCaseNames` vs `snake_case_names`
- be consistent!
- wrap long lines at column 70-80 (Rstudio)

White space

```r
new_var=(var1*var2/2)-5/(var3+var4)

# versus

new_var <- (var1 * var2 / 2) - 5 / (var3 + var4)
```

Indentation

```r
for(name in names){formula=as.formula(paste0("y~.-",name))
fit<-lm(formula,data=my_data)
coefs[["name"]]=coef(fit)
print(name)
print(summary(fit))}

# versus

for(name in names){
  formula <- as.formula(paste0("y~.-", name))
  fit <- lm(formula, data = my_data)
  coefs[["name"]] <- coef(fit)
  print(name)
  print(summary(fit))
}
```

Wrap long lines

```r
final_results <- data.frame(first_variable =
sqrt(results$mean_squared_error), second_variable =
paste0(results$condition, results$class, sep = ":"),
third_variable = results$bias)

# versus

final_results <- data.frame(
  first_variable = sqrt(results$mean_squared_error),
  second_variable = paste0(results$condition,
                           results$class, sep = ":"),
  third_variable = results$bias)
```

Go easy on your mind

- use meaningful names: “self-explainable”
- always write the formal arguments in function calls (except the first)
- benefit from autocompletion (`<tab>`) => embrace longer names
- use `TRUE` and `FALSE` not `T` and `F`
- comment, comment, comment
    - NOT what (should be clear from the code)
    - but why
    - explain the reasoning, not the code

Use meaningful names

```r
V <- myFun(m1_B)

# versus

RMSE_age_gender <- get_RMSE(lm_age_gender)
```

Programming advice

Use verbs for functions and nouns for other objects.

Write formal arguments

Benefit from auto completion using tab

```
m1_B <- lm(outcome ~ age*gender,
           exp1, condition_1, freq)

# versus

lm_age_gender <- lm(outcome ~ age * gender,
                    data = exp1,
                    subset = condition_1,
                    weigths = freq)
```

Comment, comment, comment

```
## Start every Rscript with a comment that explains
##  what the code in the script does, why it does
##  this, and to which project it belongs.
##  Your future self will be very thankful!
##
## Mention which packages you are using in this Rscript.

## Use sections to separate chunks ----------------------

## Maybe even subsections ===============================

## Recode variables so that missings are coded as "NA"
dat[dat %in% c(99, 999)] <- NA  # missings coded 99 or 999
```

Keep your code slim

Try to limit your *package-dependencies*.

Only load `library()` the packages that you absolutely need. If you are only using `dplyr`, it does not make sense to load the complete `tidyverse`.

Controversial: when possible, use the `::` operator (and consider not loading the package). `<package>::<function>`

- explicit dependencies
- less name conflicts

Never Attach

Forget about `attach()`!

Don't use it, unless you completely understand what happens (see `?attach`).

Use `with(data.frame, expression)` instead.

```
# using with()
n <- 2e+4
data <- data.frame(x = runif(n),
                   y = runif(n),
                   z = seq_len(n))
result <- with(data, exp(x) / log(z) + 5 * sqrt(y))
```

Testing R code

Writing code is error prone. Incorporate tests and checks in your workflow.

- minimal examples
- write tests and checks
- helpful packages: `testthat`, `RUnit`, `testit`, ...

Speed

Computing speed can become an issue. Avoid common pitfalls:

- don't grow, but replace
- vectorize where possible
- check the computing speed

?system.time, microbenchmark or profiling tools

```
n <- 2e+4
data <- data.frame(x = runif(n),
                   y = runif(n),
                   z = seq_len(n))
```

Speed

Don't grow!

```
system.time({
  new_data <- NULL

  for(row_nr in seq_len(NROW(data))){
    new_data <- cbind(
      data[row_nr,],
      result = exp(data$x[row_nr]) /
        log(data$z[row_nr]) +
        5 * sqrt(data$y[row_nr]))
  }
})


>    user  system elapsed
>    2.08    0.00    2.12
```

Speed

Replace!

```
system.time({
  n_rows <- dim(data)[1]
  data$result <- rep(NA, n_rows)

  for(row_nr in seq_len(n_rows)){
    data$result[row_nr] <- exp(data$x[row_nr]) /
      log(data$z[row_nr]) +
      5 * sqrt(data$y[row_nr])
  }
})


>    user  system elapsed
>    0.36    0.00    0.38
```

Speed

Vectorize!

```
system.time({
  data$result <- exp(data$x) / log(data$z) +
    5 * sqrt(data$y)
})

>    user  system elapsed
>       0       0       0
```

Speed

Compare the speed of different implementations using:

`microbenchmark::microbenchmark`

```
get_mean1 <- function(x){
  weight <- 1/length(x)
  out <- 0
  for(i in seq_along(x)){
    out <- out + x[i] * weight
  }
  return(out)
}

get_mean2 <- function(x){
  sum(x)/length(x)
}
```

Speed

Compare the speed of different implementations using:

`microbenchmark::microbenchmark`

```
x <- rnorm(500)
microbenchmark::microbenchmark(
  mean(x), get_mean1(x), get_mean2(x))


> Unit: nanoseconds
>          expr   min    lq  mean median    uq     max neval
>       mean(x)  2400  3500  5786   3900  4900  123000   100
>  get_mean1(x) 12000 22300 91504  24300 29400 6439200   100
>  get_mean2(x)   700  1200 17527   1300  1800 1583800   100
```

Speed

Programming advice

Don't worry about speed before it becomes an issue.

Wrap Up

General Advice

- Investing time in learning R pays off
- It's a steady learning curve
- Learn from masters
- Rewrite important code - the first attempt is usually not the best approach

General R Advice

- Document well
- Use a consistent style
- Write functions
- Split long functions in smaller ones
- Write wrappers
- Use Iteration (don't copy paste)
- Use matrix operations and vectorized functions instead of loops
- Use git

Literature Recommendations

R Resources

- Avanced R Ed. 1 (http://adv-r.had.co.nz/)
- Avanced R Ed. 2 (https://adv-r.hadley.nz/)
- R Inferno (https://www.burns-stat.com/pages/Tutor/R_inferno.pdf)
- R Packages (https://r-pkgs.org/)
- Clean Code (https://mooc.aptikom.or.id/pluginfile.php/1174/mod_resource/content/1/Clean%20Code_%20A%20Handbook%20of%20Agile%20Software%20C%20-%20Robert%20C.%20Martin.pdf)

Thank you for your attention!

Thank you for your attention!

Questions? Remarks?