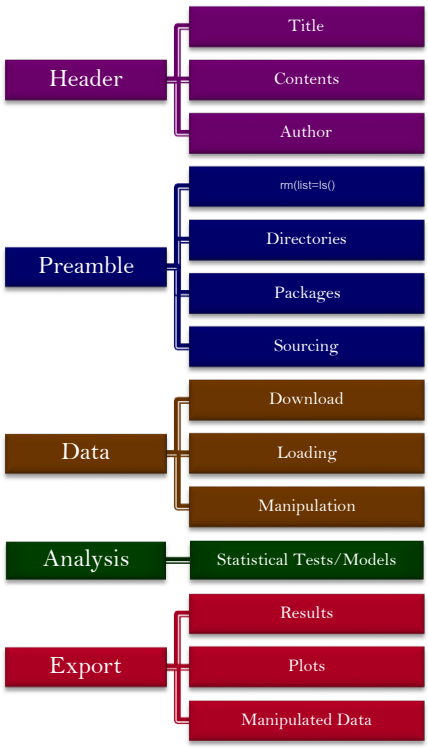
CODING PRACTICES

Life with R

Erik Kusch (erik.kusch@bio.au.dk), PhD Student
Department of Biology
Section for Ecoinformatics & Biodiversity
Center for Biodiversity Dynamics in a Changing World (BIOCHANGE)
Aarhus University

Code Structure



Code Folding

- Braced Regions ("{…}")
- Code Sections

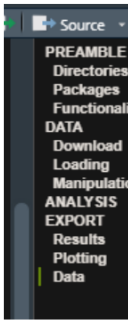
Code Sections

- Title followed by 4:
  - "#"
  - "="
  - "-"
- Section Hierarchy determined by preceding "#"
  - Requires Rstudio Version ≥ 1.4

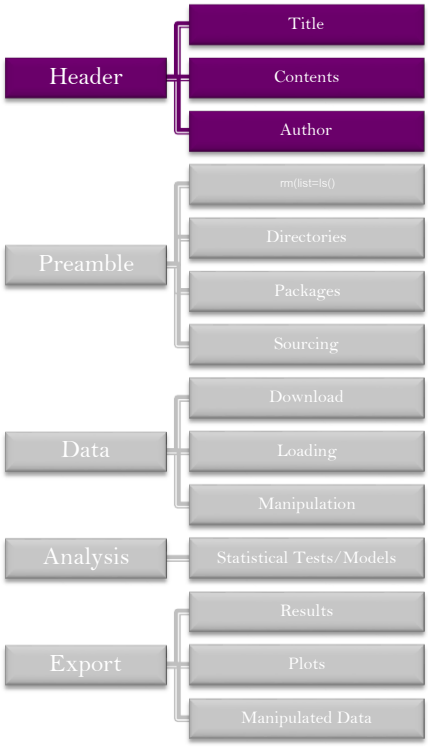
```
FUN.Addition <- function(Object1, Object
  Result <- Object1+Object2
  return(Result)
}
```

```
FUN.Addition <- function(Object1, Object2){
```

```
# PREAMBLE ============================

## Directories ---------------------
## Packages ------------------------
## Functionality -------------------

# DATA ================================

## Download ------------------------
## Loading -------------------------
## Manipulation --------------------

# ANALYSIS ============================

# EXPORT ==============================

## Results -------------------------
## Plotting ------------------------
## Data ----------------------------
```



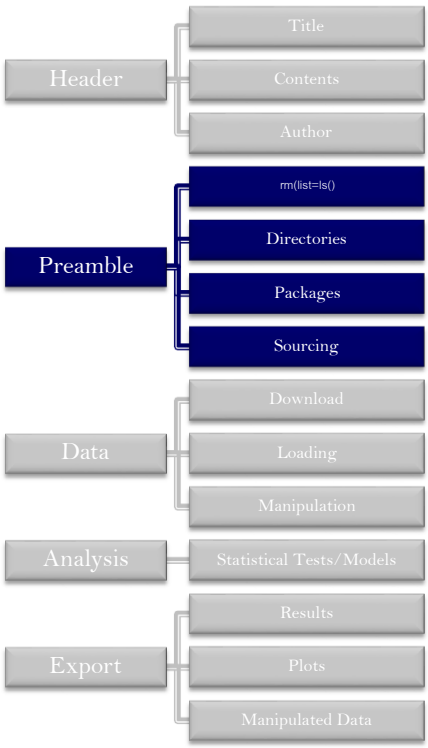
Header




```
#' ##################################################################
#' PROJECT: [PhD - Plant Functional Trait Networks]
#' CONTENTS:
#'   - Data Retrieval
#'   - Data Cleaning
#'   - Intrinsic Fitness Model Building
#'   - Model Object Export
#'   - Interaction Network Plotting
#'  DEPENDENCIES.
#'   - 0 - Preamble.R
#'   - X - Functions_Plotting.R
#'   - X - Functions_Bayes.R
#' AUTHOR: [Erik Kusch]
#' ##################################################################
```

- Information of:
  - Project Membership
  - Contents
  - Dependencies
  - Authorship
- Can be used to track edit dates, but file versioning is better for this

Preamble



rm(list=ls())

- Clears working directory

→Assures your code is self-contained

Directories

- Avoid hard-coding directories

→Use getwd() and file.path() for soft-coded directories

Packages

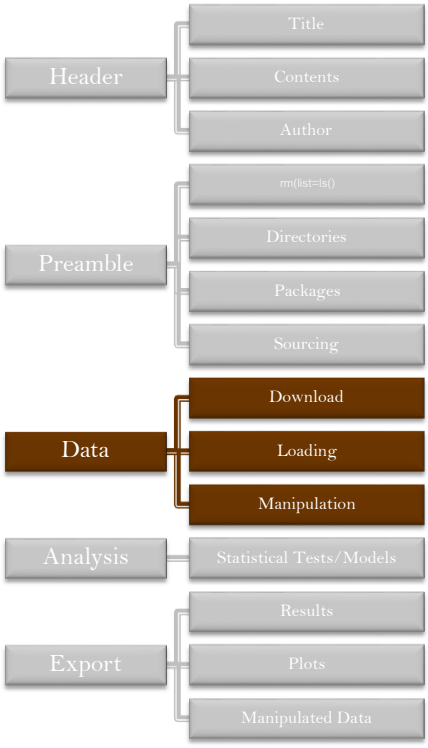
- Do not use library() to load packages

→Automatically install packages that are needed, but not installed

Sourcing/Functionality

- Source other code documents
- Create small helper functions

Data



File Checks

- Use file.exists(...) to check if data files are present

Download

- httr::GET(...) for data download given a link
- utils::unzip(...) for extraction of .zip archives

Loading

- Do not load data via dialogue boxes!

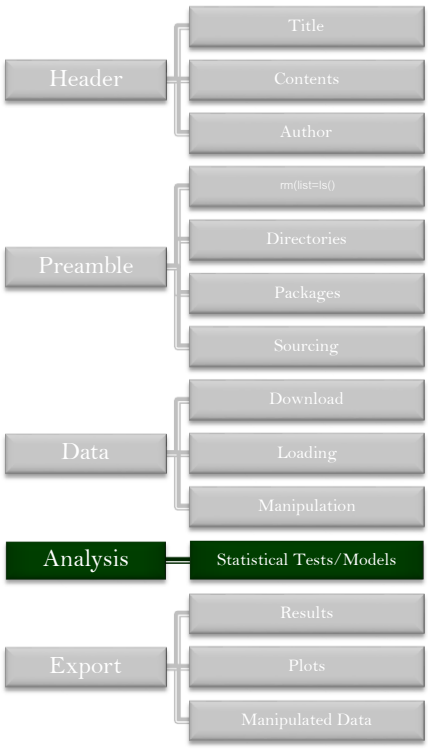
→Use code to load data

Manipulation

- Do not manipulate data outside of R

→ Make all data manipulation traceable through R-Code

Analysis



Data

- No more manipulation of data
- Subsetting is permissible

Analyses

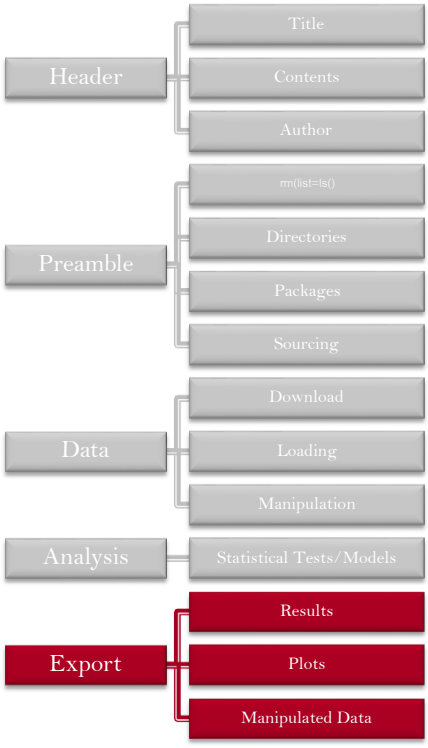
- Variable Selection
- Analyses
- Model Comparison
- Model Evaluation

Naming

- Do not override model objects

→Use unique names for model objects/outputs

Export



Results

- Export model objects as .Rdata with save(...)
- Export model summaries as .txt with sink(...) ... sink()

Plotting

- Export ggplot2 figures with ggsave(...)
- Export base plot figures with png(...) ... dev.off()

Manipulated Data

- Do not override original data files!

→Append suffix to data name (e.g. "_clean")

- Do not save as excel-readable files unless strictly necessary

→Use saveRDS(...) to store .rds files

Naming Schemes

Specificity

- What does the object contain?

Spaces

- "_"for objects and "." for functions

Capitalisation

- Choose your style, but be consistent

Classes

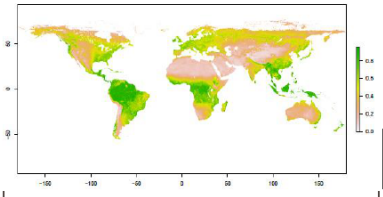
- Suffix/Prefix to indicate object class

Iterators

- Avoid single-letter iterators

Logical Operators

- Do not use "T" or "F"



```
FUN.Addition <- function(Object1, Obj
  Result <- Object1-Object2
  return(Result)
}
```

	Object	Function
Bad Name	mydata	fun
Specificity	NDVI1981	Additionfunction
Spaces	NDVI_1981	Addition.function
Capitalisation	NDVI_1981	Addition.Function
Classes	NDVI1981_ras	FUN.Addition

Comments

What?

- Comments

→Start with "#"

→Are not executed as code

Why?

- Convey goal of the code lines
- Revisit code

How?

- Good comments:
  - Specify (what the code is doing)
  - Justify (what is being done)


```
# Data Manipulation
```
BAD!

```
# Z-Score calculation for comparability
```
GOOD!

Quality of Life with R

Object Assignment

- Use "<-" instead of "="

Readability

- Spaces in function specification

```
c(1,2,rep(3,5)) # bad
c(1, 2, rep(3, 5)) # good
```

- Line breaks between individual chunks of code
- Soft-wrap in RStudio to prevent scrolling side-ways

Consistency

- Develop a style and be consistent in using it

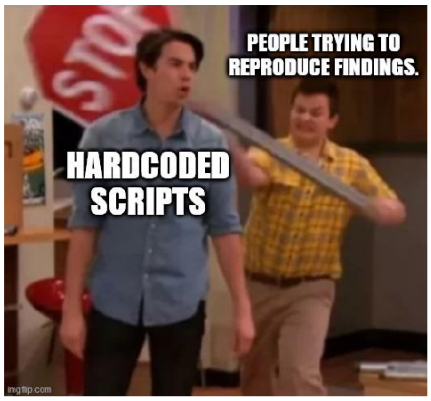
Reproducibility

Why Care?

- Reproducible research as gold-standard of science
- Journal Requirements
  - Code sharing
  - Data sharing

How?

- STOP. HARDCODING.
- Test your scripts on different machines.



Reproducibility – Cardinal Sins

Packages

What People Do

- Use library(...) to load packages

→Not everyone has the same packages installed

- Force installation via install.packages(...)

→May disrupt local package versions

How To Do Better

- Identify packages that need installing and only install those prior to loading

```
install.load.package <- function(x) {
  if (!require(x, character.only = TRUE))
    install.packages(x, repos='http://cran.us.r-project.org')
  require(x, character.only = TRUE)
}
package_vec <- c(
  # names of the packages required placed here as character objects
)
sapply(package_vec, install.load.package)
```

Directories

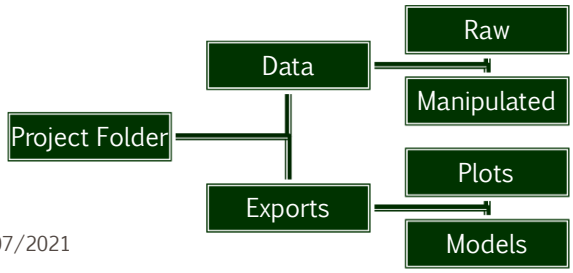
What People Do

- Hardcode working directories with setwd(...)

→ Others do not have access to your hard drive

How To Do Better

- Organise projects in separate folders
- Softcode working directories:
  - getwd(...) to index project folder
  - file.path(...) to index sub-directories



File Checks

What People Do

- Overwrite existing data files

→Can possibly break reproducibility and cement errors

- Carry out analyses whose output is already present

→ Waste of processing time and power

How To Do Better

- Check whether files are present with file.exists(...)

→Load/Manipulate

→Load/Write

→Load/Analyse

Random Processes

What People Do

- Sample data sets
- Partition data into test and training data sets

→May induce severe bias

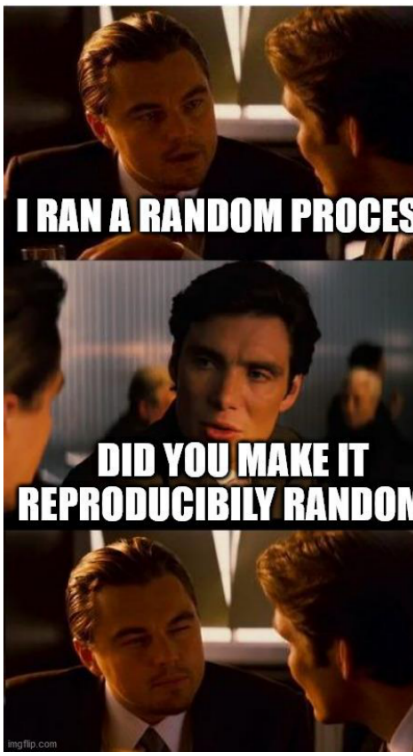
How To Do Better

- Set a seed to make random processes reproducible

→set.seed(...)

- Sample multiple times and average out to remove bias

→Bootstrapping



Reproducibility - Reporting

Code is as essential as your written report.

Reporting Code

Using Rmarkdown for your research comes with a multitude of advantages:

1. Entire **workflow in one program** (RStudio)
2. **Research** and reports **reproducible** at the click of **one button**
3. **Combines** R functionality and LATEX formatting (if desired)
4. **Consistent formatting**
5. **Clear presentation of code**
6. **Dynamic documents** (you can generate various output document types)
7. Applicable for **almost all document types** you may desire as an output

More Quality of Life Improvements

Everything hereafter is nice-to-have or nice-to-do, but not essential.

Functions & Sourcing

Big projects lend themselves well to a **multi-document workflow**:

Functions

`Fun <- function(...){}` to create a custom function

Useful for:
- Soft-coding analyses
- Repeating code for different input parameters

Should be:
- internally consistent
- well-documented
- easy to understand

Sourcing

`Source(...)` command to load/execute `R` scripts

Useful for:
- Keeping code structured and concise
- Storing extra functionality

Should:
- Use sensible file names
- Sourced functions need to be called

Progress Bar & Estimators

Progress Bar

- Updates on how code is progressing
- Especially useful when your code involves loops

```
Data_vec <- 1:100 # a vector on integers from 1 to 100
# creating progress bar
ProgBar <- txtProgressBar(min = 0, max = length(Data_vec),
                          style = 3)
# looping over contents of Data_vec
for(Iter_ProgBar in 1:length(Data_vec)){
  setTxtProgressBar(ProgBar, Iter_ProgBar) # updating ProgBar
} # end of Data_vec loop
```

```
|======================================================================| 100%
```

Estimators

- When to expect code to finish processing
- Most useful in loop-based approaches as they need a baseline for the estimation

```
Data_vec <- 1:100 # a vector on integers from 1 to 100
T_Begin <- Sys.time() # record time
# looping over contents of Data_vec
for(Iter_Est in 1:length(Data_vec)){
  Sys.sleep(.1) # pause for .1 seconds
  # estimator produced on first iteration
  if(Iter_Est == 1){
    T_End <- Sys.time() # record time
    Duration <- as.numeric(T_End)-as.numeric(T_Begin) # time difference
    print(paste("Estimated time to finish:",
                as.POSIXlt(T_Begin + Duration*length(Data_vec),
                           tz = Sys.timezone(location=FALSE))
                ))
  } # end of estimator check
} # end of Data_vec loop
```

```
## [1] "Estimated time to finish: 2020-03-25 00:33:50"
```


Errors, Warnings & User-Inputs

Errors

- Stop code execution
- Report message to console

```
stop("This is an error message which breaks the code execution.")
```

```
> stop("This is an error message which breaks the code execution.")
Error: This is an error message which breaks the code execution.
```

Warnings

- Do not stop code execution
- Report message to console

```
warning("This is a warning which is displayed once execution of code is terminate
```

```
> warning("This is a warning which is displayed once execution of code is terminate
Warning message:
This is a warning which is displayed once execution of code is terminated.
```

User-Input

- Pause code execution
- Wait for user-input in console

```
Input <- readline("This prompts an input from the user in the console
                  and halts code execution until input is supplied.")
```

```
> Input <- readline("This prompts an input from the user in the console
+                   and halts code execution until input is supplied.")
This prompts an input from the user in the console
                  and halts code execution until input is supplied.3
```

```
Values
  Input                                "3"
```

Parallel Processing

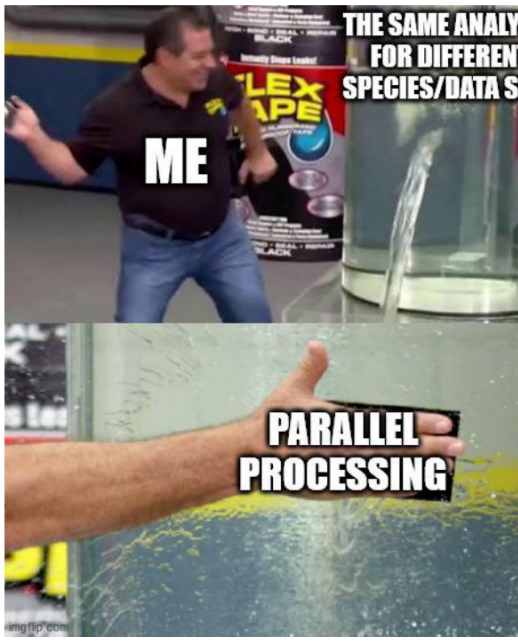
- By default, R only uses one core

→Inefficient use of computational power

- Use **parallel processing** to make use of remaining cores

→ Particularly powerful when paired with eval(parse(text="…"))

```
library(doParallel) # for registering clusters
library(foreach) # for parallel processing
Cores <- detectCores() # identify the number of cores in your machine
cl <- makeCluster(Cores) # create virtual cluster
registerDoParallel(cl) # register cluster of cores
# parallel processing
foreach(Iter_Par = 1:length(Data_vec)) %dopar% {
  # your function here
} # end of parallel processing
stopCluster(cl) # stop cluster
```



Versioning

Use **GitHub** to:

- **Track** code and development

→You may revert to previous versions of code if you broke something

- **Share** code with others

→You can make available R functionality and packages on GitHub

- **Collaborate** with others

→Pull requests and the issue feature allow for collaborative code development