# Modernization of Statistical Production with R

Mark van der Loo

Statistics Netherlands, University of Leiden

2023-12-13

# Modernization

# General Motivation

Less control

More volatile

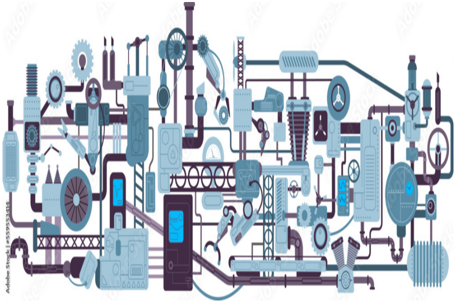
Harder to get

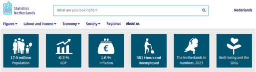
Survey

Administrative

Web/Big/Sensor

Privately Held





eurostat OECD United Nations Statistics Division

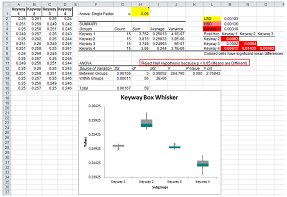
**Legal obligations**
Comparable over time and space


**Current events**
Quick Response

**Input** | **Throughput** | **Output**

uRosConf

# Modernization at Statistics Netherlands




**Modular**
infrastructure, software, methods, processes

**Standardized**
(meta)data, infrastructure, software, methods, processes

**Minimize Human Effort**

**Rule-based**
processes

**Re-use**
(meta)data, infrastructure, software, methods, processes

**Creating statistics**

**Monitoring automated processes**

# The rest of this talk

My approach to developing packages

Why I think R has great features for

- Standardization and Reuse
- Rule-based data processing
- Modularity
- Building User Interfaces

# General Approach

**Practical Issue**

Drill down to a fundamental (math) problem

$R = \{\}$;
**for** $a \in A$ **do**
  $i = 0$ ;
  $d = f^*(\{a\})$ ;
  **while** $i < n \wedge \neg\beta(d)$ **do**
    $i = i + 1$ ;
    $d = (f^* \circ F_i^* \circ F_i)(a)$ ;
  **end**
  **if** $i < n \vee \beta(d)$ **then**
    $R = R \cup \{(a, i, \phi(d))\}$;
  **end**
**end**

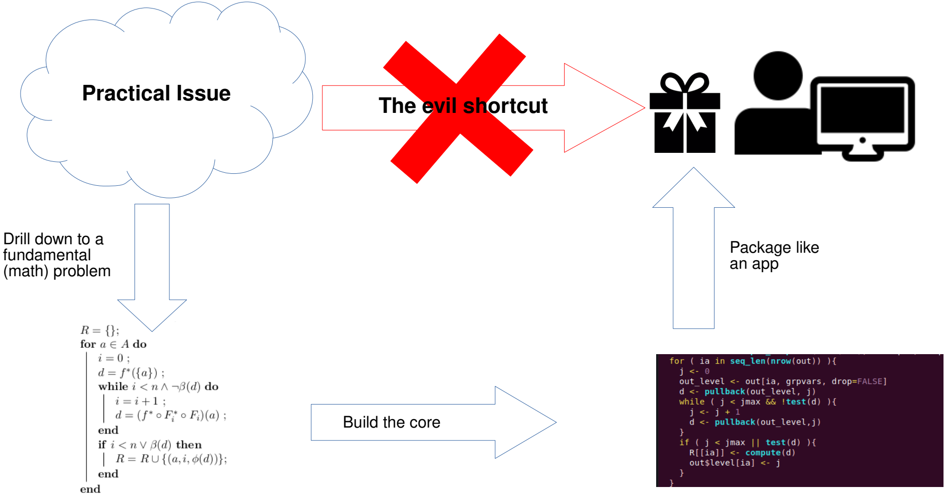
Build the core

```
for ( ia in seq_len(nrow(out)) ){
  j <- 0
  out_level <- out[ia, grpvars, drop=FALSE]
  d <- pullback(out_level, j)
  while ( j < jmax && !test(d) ){
    j <- j + 1
    d <- pullback(out_level,j)
  }
  if ( j < jmax || test(d) ){
    R[[ia]] <- compute(d)
    out$level[ia] <- j
  }
}
```

Package like an app

uRosConf

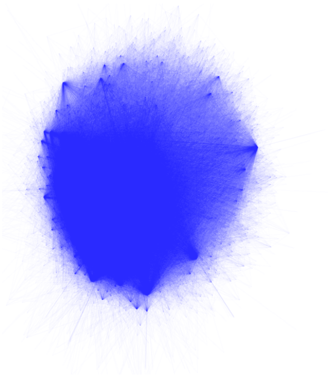
# Approach




uRosConf

# Standardization and Reuse

# Standardization and Reuse with R

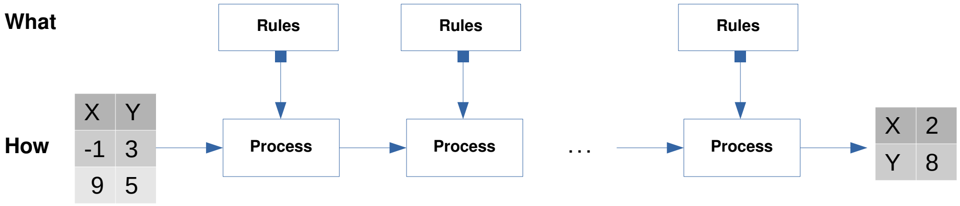


CRAN dependency network
(giant component)

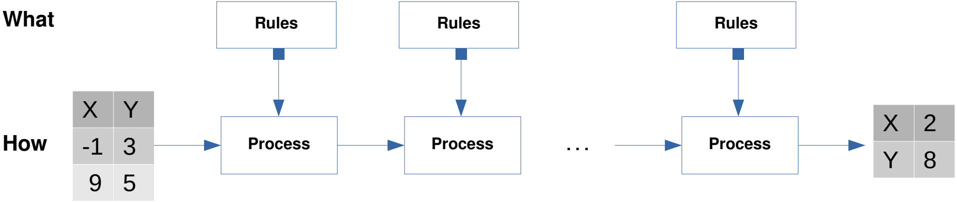
- 19k packages
- 108k hard dependencies
- Higly standardized
  - Description and organization
  - Dependencies
  - Documentation
- Each package passes checks:
  - tests, examples
  - documentation, urls
  - code sanity
- Works on Windows, Linux, MacOS
- Works for previous, current, and development release of R
- **All dependencies resolve**
- **4 volunteers** + payed students

# Rule-based data processing

# Rule-based data processing

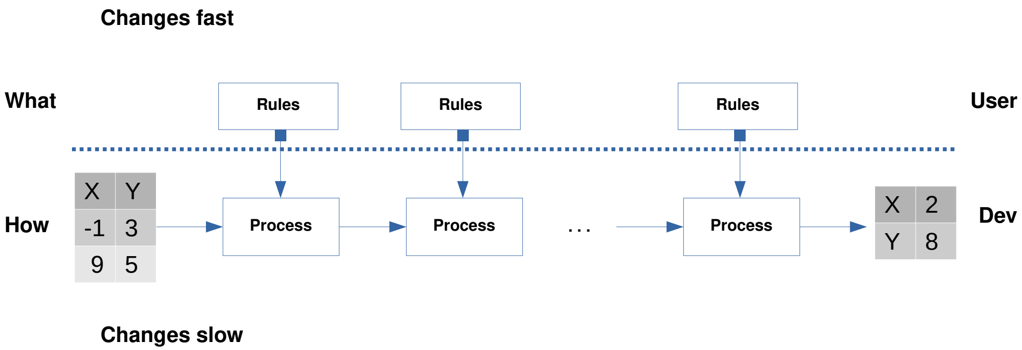


# Rule-based data processing




```
data |>
  dplyr::filter(x > 0) |>
  dplyr::summarise(X = mean(X), Y=mean(Y)) |>
  t()
```

# Rule-based data processing




Rules should be documented; version controlled; stored and maintained externally.

# The `validate` package

DEMO

# The `validate` package

## Approach

1. Theory: $v : D^K \hookrightarrow \{\mathsf{T}, \mathsf{F}\}$
2. Implementation of core theory into `R`
3. Design user interface / workflow
   - CRUD operations
   - Selecting
   - Investigating (e.g. `variables()`)
   - Validation + summarization of results

# The `validate` package

## Approach

1. Theory: $v : D^K \hookrightarrow \{\mathsf{T}, \mathsf{F}\}$
2. Implementation of core theory into `R`
3. Design user interface / workflow
   - CRUD operations
   - Selecting
   - Investigating (e.g. `variables()`)
   - Validation + summarization of results

## Result

1. User-defined rules.
2. Easy to learn/get started.
3. Backend extensible (e.g. `validatedb`)
4. Use rules as input for other packages. (e.g. `errorlocate`)
5. Reason, investigate, manipulate rules and rulesets.

# The `validate` package

## Approach

1. Theory: $v : D^K \hookrightarrow \{\mathsf{T}, \mathsf{F}\}$
2. Implementation of core theory into `R`
3. Design user interface / workflow
   - CRUD operations
   - Selecting
   - Investigating (e.g. `variables()`)
   - Validation + summarization of results

## Result

1. User-defined rules.
2. Easy to learn/get started.
3. Backend extensible (e.g. `validatedb`)
4. Use rules as input for other packages. (e.g. `errorlocate`)
5. Reason, investigate, manipulate rules and rulesets.

## NB

In a 2020 we demonstrated that all ‘generic’ data validation rules for the ESS be expressed with `validate`. https://github.com/SNStatComp/GenericValidationRules

# Contrasting approach (e.g. `PointBlank`, `DataMaid`)

## Approach

1. Collect individual validation cases.
2. Implement each case in a function.

# Contrasting approach (e.g. `PointBlank`, `DataMaid`)

### Approach

1. Collect individual validation cases.
2. Implement each case in a function.

### Result

1. No user-defined rules
2. Need to read manual for each function
3. Rules not reusable accros packages
4. No (systematic) manipulation or investigation of rules and rulesets.

# Why R?

## R has the 'reflexive' property

```r
head(cars, 3)
```

```
  speed dist
1     4    2
2     4   10
3     7    4
```

```r
r <- expression(dist > speed)
eval(r, head(cars,3))
```

```
[1] FALSE  TRUE FALSE
```

```r
r[[1]][[1]] == ">"
```

```
[1] TRUE
```

# Why R?

## R has the 'reflexive' property

```r
head(cars, 3)
```

```
  speed dist
1     4    2
2     4   10
3     7    4
```

```r
r <- expression(dist > speed)
eval(r, head(cars,3))
```

```
[1] FALSE  TRUE FALSE
```

```r
r[[1]][[1]] == ">"
```

```
[1] TRUE
```

## Anatomy of validate (pseudocode)

```r
# parse
rules <- parse(file="rulefile.R")
v <- new_validator()
for ( r in rules ){
  if (is_validating(r)){
    v <- v + r
  }
}

# eval
out <- new_validation()
for (r in v){
  out <- out + eval(r, data)
}
```

# More information

MPJ van der Loo, E de Jonge (2020). Data Validation. In Wiley StatsRef: Statistics Reference Online, pages 1-7. American Cancer Society.

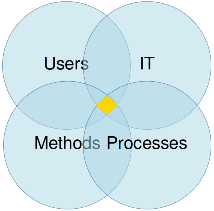
MPJ van der Loo, E de Jonge (2021). Data Validation Infrastructure for R. Journal of Statistical Software 1–22 97

M.P.J. van der Loo, E. de Jonge, K.O. ten Bosch (2022). Rule Management. UNECE Expert Meeting on Statistical Data Editing.

# Modularity

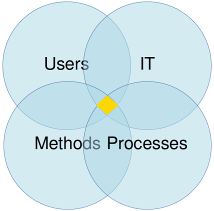
# Why Modularity is Difficult

1. Perspectives on Modules



# Why Modularity is Difficult

## 1. Perspectives on Modules

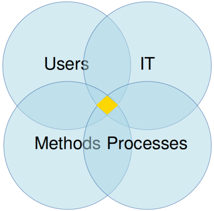



## 2. Modularity $\neq$ Composability

$$\begin{tikzcd} a \arrow[r, "f"] \arrow[rr, bend left, "g \circ f"] & b \arrow[r, "g"] & c \end{tikzcd}$$

# Why Modularity is Difficult

## 1. Perspectives on Modules



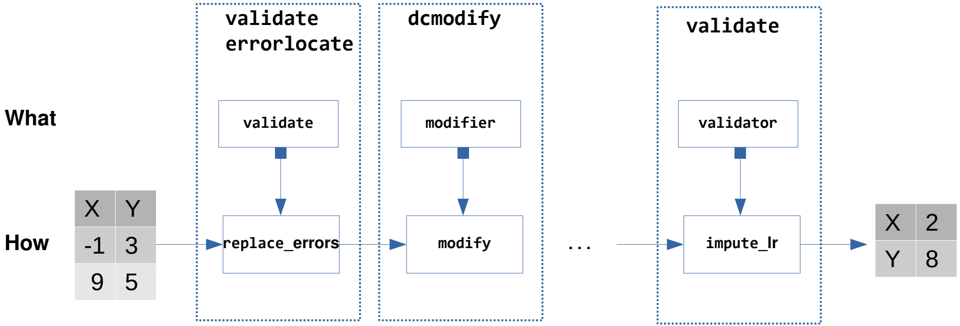

## 2. Modularity $\neq$ Composability

$$\begin{tikzcd} a \arrow[r, "f"] \arrow[rr, bend left, "g \circ f"] & b \arrow[r, "g"] & c \end{tikzcd}$$

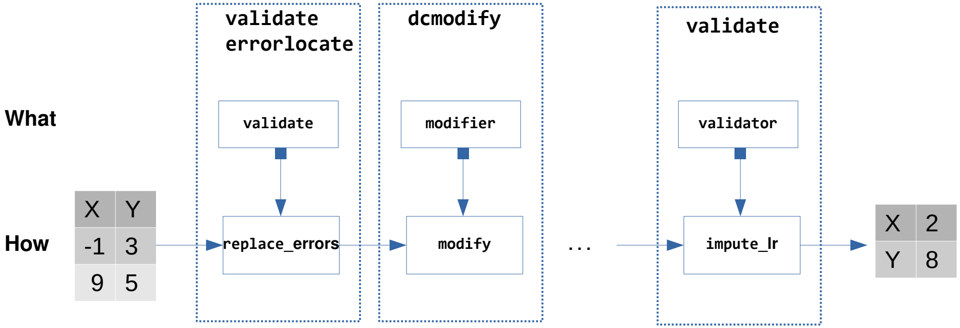
### Ideal Modules

Are easy to understand and use (app-like); composable; implement a single piece of methodology; are controllable from the outside (rules/parameters).

# Examples of modules for data processing



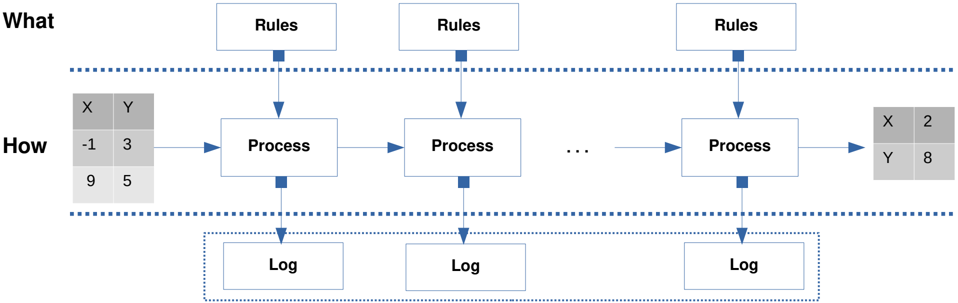
# Examples of modules for data processing



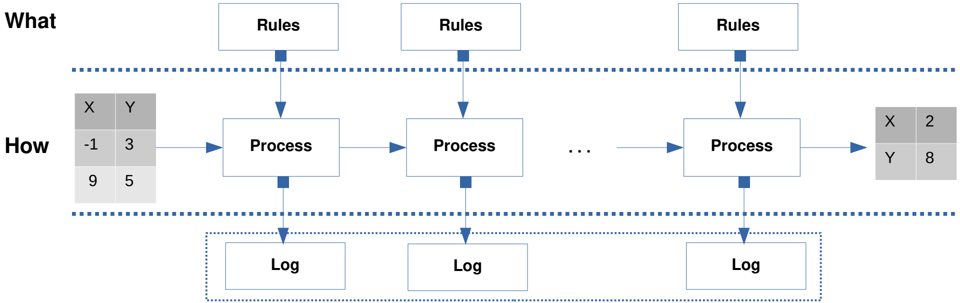

## Note

- Composing from left to right
- Modules can be independently added or removed

# A module for process monitoring (logging)?



# A module for process monitoring (logging)?




## Question

- How to implement a module that can be independently added for logging?

# It can be done: `lumberjack`

DEMO

# How does that work? 1: Adding a second data stream

## Concept

A sequence of R expressions $e_i;\, i = 1, 2, \ldots n$, executed in order can be considered as one long composed expression

$$e_n \circ e_{n-1} \circ \cdots \circ e_1.$$

The idea is to replace this boring composition operator with an interesting one.

# How does that work? 1: Adding a second data stream

`source()` (schematically)

```
expressions <- parse("myfile.R")
for (e in expressions){
  eval(e)
}
```

# How does that work? 1: Adding a second data stream

## `source()` (schematically)

```
expressions <- parse("myfile.R")
for (e in expressions){
  eval(e)
}
```

## `run_file()` (schematically)

```
expressions <- parse("myfile.R")
n <- 0
for (e in expressions){
  eval(e)
  n <- n + 1
}
printf("Counted %d expressions\n",n)
```

# How does that work? 2: communication from user stream to log stream

```r
# myfile.R
x <- runif(1)
start_counting()
y <- 20
z <- x + y
```

# How does that work? 2: communication from user stream to log stream

```
# myfile.R
x <- runif(1)
start_counting()
y <- 20
z <- x + y
```

## Concept

A special expression (like `start_counting()`) changes the compostion operator.

$$e_n \circ e_{n-1} \circ \cdots \circ \mathbf{e_k} \circ \cdots \circ e_1.$$

# How does that work? 2: communication from user stream to log stream

```
# myfile.R
x <- runif(1)
start_counting()
y <- 20
z <- x + y
```

## Concept

A special expression (like `start_counting()`) changes the compostion operator.

$$e_n \circ e_{n-1} \circ \cdots \circ \mathbf{e_k} \circ \cdots \circ e_1.$$

- ~~Analyze Abstract Syntax Tree?~~ `if(x >= 0.5) start_counting()`
- Smuggle information from the special expression to the file runner.

# Creating a smuggler with higher order functions

```
make_smuggler <- function(fun, env){
  function(...){
    env$result <<- fun(...)  # smuggle the result of 'fun' to 'env'
    env$result
  }
}
```

# Creating a smuggler with higher order functions

```
make_smuggler <- function(fun, env){
  function(...){
    env$result <<- fun(...)  # smuggle the result of 'fun' to 'env'
    env$result
  }
}
```

```
store <- new.env()
mymean <- make_smuggler(mean, store)

mymean(1:3) # works just like 'mean'
```

```
[1] 2
```

```
store$result # but the result is also copied into 'store'
```

```
[1] 2
```

# Using a smuggler in `run_file()`

```
start_counting <- function() return(TRUE)
```

```
run_file <- function(rfile){
 runtime <- new.env()
 store   <- new.env()

 runtime$start_counting <- smuggler(start_counting, store)

 expressions <- parse(rfile)
 n <- 0
 for (e in expressions){
   eval(e, runtime)
   if (isTRUE(store$result)) n <- n + 1
 }
 printf("Executed %d expressions", n)
}
```

# Design of `lumberjack`

## Approach

1. Theory:
    - custom file runner
    - local side effect (smuggler)
    - local masking
2. Implement core into R
3. Design user interface
    - start/stop/pause/dump
    - Logger objects and interface

# Design of `lumberjack`

## Approach

1. Theory:
   - custom file runner
   - local side effect (smuggler)
   - local masking
2. Implement core into R
3. Design user interface
   - start/stop/pause/dump
   - Logger objects and interface

## Result

1. Easy to get started/learn
2. Extensible with new loggers
   - Integration with validate
3. Composable (1 LoC)
4. Clean separation between runtime environment and logging environment.

# Contrasting approaches: most logging packages

## Approach

1. Define specific logging use cases
   - fixed output format/type
   - fixed output target/type
   - logging level
2. Implement as R functions

# Contrasting approaches: most logging packages

## Approach

1. Define specific logging use cases
   - fixed output format/type
   - fixed output target/type
   - logging level
2. Implement as R functions

## Result

1. Easy to get started/learn
2. Need to insert code in several places
3. Usually not extensible w/new loggers
4. Logging takes place in runtime environment

NB:

5. Same approach used in `tinytest`

# Modularity: Why R?

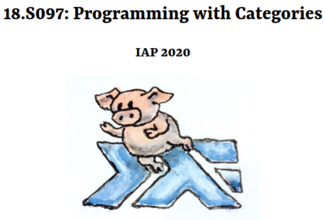
## All modules

- R package system

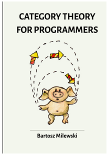
## 'cross-cutting' modules, like logging

- Higher-order functions (func. lang.)
- Reflexive: parse/eval and masking
- Environment as smugglers' path

# More information




B. Fong, B Milewski, D. Spivak (2020)
Programming with categories; online lectures
http://brendanfong.com/programmingcats.html




B. Milewski (2019) Category theory for programmers. Blurb.com.

CONTRIBUTED RESEARCH ARTICLES 42

**A Method for Deriving Information from Running R Code**

*by Mark P.J. van der Loo*

**Abstract** It is often useful to tap information from a running R script. Obvious use cases include monitoring the consumption of resources (time, memory) and logging. Perhaps less obvious cases include tracking changes in R objects or collecting the output of unit tests. In this paper, we demonstrate an approach that abstracts the collection and processing of such secondary information from the running R script. Our approach is based on a combination of three elements. The first element is to build a customized way to evaluate code. The second is labeled *local masking* and it involves temporarily masking a user-facing function so an alternative version of it is called. The third element we label *local side effect*. This refers to the fact that the masking function exports information to the secondary information flow without altering a global state. The result is a method for building systems in pure R that lets users create and control secondary flows of information with minimal impact on their workflow and no global side effects.

**Introduction**

The R language provides a convenient language to read, manipulate, and write data in the form of scripts. As with any other scripted language, an R script gives a description of data manipulation activities, one after the other, when read from top to bottom. Alternatively, we can think of an R script as a one-dimensional visualization of data flowing from one processing step to the next, where intermediate variables or pipe operators carry data from one treatment to the next.

We run into limitations of this one-dimensional view when we want to produce data flows that are somehow 'orthogonal' to the flow of the data being treated. For example, we may wish to follow the state of a variable while a script is being executed, report on progress (logging), or keep track of resource consumption. Indeed, the sequential (one-dimensional) nature of a script forces one to introduce extra expressions between the data processing code.

As an example, consider a code fragment where the variable x is manipulated.

```
x[x > threshold] <- threshold
x[is.na(x)]  <- median(x, na.rm=TRUE)
```

In the first statement, every value above a certain threshold is replaced with a fixed value, and next, missing values are replaced with the median of the completed cases. It is interesting to know how an aggregate of interest, say the mean of x, evolves as it gets processed. The instinctive way to do this is to edit the code by adding statements to the script that collect the desired information.

```
meanx <- mean(x, na.rm=TRUE)
x[x > threshold] <- threshold
meanx <- c(meanx, mean(x, na.rm=TRUE))
x[is.na(x)]  <- median(x, na.rm=TRUE)
meanx <- c(meanx, mean(x, na.rm=TRUE))
```

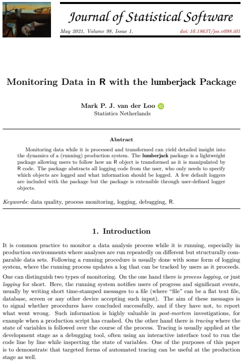
This solution clutters the script by inserting expressions that are not necessary for its main purpose. Moreover, the tracking statements are repetitive, which validates some form of abstraction.

A more general picture of what we would like to achieve is given in Figure 1. The 'primary data flow' is developed by a user as a script. In the previous example, this concerns processing x. When the script runs, some kind of logging information, which we label the 'secondary data flow' is derived implicitly by an abstraction layer.

Creating an abstraction layer means that concerns between primary and secondary data flows are separated as much as possible. In particular, we want to prevent the abstraction layer from inspecting or altering the user code that describes the primary data flow. Furthermore, we would like the user to have some control over the secondary flow from within the script, for example, to start, stop, or parameterize the secondary flow. This should be done with minimum editing of the original user code, and it should not rely on global side effects. This means that neither the user nor the abstraction layer for the secondary data flow should have to manipulate or read global variables, options, or other environmental settings to convey information from one flow to the other. Finally, we want to treat the availability of a secondary data flow as a normal situation. This means we wish to avoid using signaling conditions (e.g., warnings or errors) to convey information between the flows unless there is an actual exceptional condition such as an error.

The R Journal Vol. 13/1, June 2021 ISSN 2073-4859

Mark P.J. van der Loo (2021). A Method for Deriving Information from Running R Code. The R Journal 13 42–52



*Journal of Statistical Software*

*May 2021, Volume 98, Issue 1.* *doi: 10.18637/jss.v098.i01*

**Monitoring Data in R with the lumberjack Package**

**Mark P. J. van der Loo**
Statistics Netherlands

**Abstract**

Monitoring data while it is processed and transformed can yield detailed insight into the dynamics of a (running) production system. The **lumberjack** package is a lightweight package allowing users to follow how an R object is transformed as it is manipulated by R code. The package abstracts all logging code from the user, who only needs to specify which objects are logged and what information should be logged. A few default loggers are included with the package but the package is extensible through user-defined logger objects.

*Keywords*: data quality, process monitoring, logging, debugging, R.

**1. Introduction**

It is common practice to monitor a data analysis process while it is running, especially in production environments where analyses are run repeatedly on different but structurally comparable data sets. Following a running procedure is usually done with some form of logging system, where the running process updates a log that can be tracked by users as it proceeds.

One can distinguish two types of monitoring. On the one hand there is *process logging*, or just *logging* for short. Here, the running system notifies users of progress and significant events, usually by writing short time-stamped messages to a file (where "file" can be a flat text file, database, screen or any other device accepting such input). The aim of these messages is to signal whether procedures have concluded successfully, and if they have not, to report what went wrong. Such information is highly valuable in *post-mortem* investigations, for example when a production script has crashed. On the other hand there is *tracing* where the state of variables is followed over the course of the process. Tracing is usually applied at the development stage as a debugging tool, often using an interactive interface tool to run the code line by line while inspecting the state of variables. One of the purposes of this paper is to demonstrate that targeted forms of automated tracing can be useful at the production stage as well.

MPJ van der Loo (2021). Monitoring data in R with the lumberjack package. Journal of Statistical Software 98 1–11

# User interface

# Example: `simputation`

## Example data

```
  staff turnover other.rev total.rev staff.costs
1    75       NA        NA      1130          NA
2     9     1607        NA      1607         131
3    NA     6886       -33      6919         324
```

# Example: simputation

## Example data

|   | staff | turnover | other.rev | total.rev | staff.costs |
|---|---|---|---|---|---|
| 1 | 75 | NA | NA | 1130 | NA |
| 2 | 9 | 1607 | NA | 1607 | 131 |
| 3 | NA | 6886 | -33 | 6919 | 324 |

## Issues

- Need for fall-through scenario for imputation
- Multiple variables to impute
- Multiple models

# Imputation in R

## Specialized packages

- 100+ available (VIM, mice, Amelia, mi, ...)
- Interfaces vary (a lot)

## DIY with model/predict

```
m                <- lm(Y ~ X, data = mydata)
ina              <- is.na(mydata$Y)
mydata[ina, "Y"] <- predict(m, newdata = mydata[ina,])
# YMMV, be cause how 'predict' works, depends on the model
```

## Result

- Lots of 'boilerplate' code needed, covering many cases
- Hard to experiment and test

# Idea of the simputation package

## Provide

- a *uniform interface*,
- with *consistent behaviour*,
- across *commonly used methodologies*

## To facilitate

- experimentation
- configuration for production

# The simputation interface

```
impute_<model>(data
  , <imputed vars> ~ <predictor vars>
  , [options])
```

# The simputation interface

```
impute_<model>(data
   , <imputed vars> ~ <predictor vars>
   , [options])
```

## Example: linear model imputation

```r
impute_lm(ret, other.rev ~ turnover) |> head(3)
```

```
  staff turnover other.rev total.rev staff.costs
1    75       NA        NA      1130          NA
2     9     1607  5427.113      1607         131
3    NA     6886   -33.000      6919         324
```

# Fall-through scenario by chaining imputations

```
ret |>
  impute_lm(other.rev ~ turnover + staff) |>
  impute_lm(other.rev ~ staff) |>
  head(3)
```

```
  staff turnover other.rev total.rev staff.costs
1    75       NA 13914.261      1130          NA
2     9     1607  6089.698      1607         131
3    NA     6886   -33.000      6919         324
```

# Other features

- Impute multiple variables with the same model
- Groupwise imputation
- Multivariate methods (e.g. missForest, EM)

# Approach

## Approach

1. Essentials of specification
2. Map to R features (formula-data)
3. Build on top of existing packages

# Approach

## Approach

1. Essentials of specification
2. Map to R features (formula-data)
3. Build on top of existing packages

## Result

1. Easy to learn and use :-)
2. Fall-through scenarios supported :-)
3. Not driven by external rules :-/
4. Hard to extend :-(

# Why R?

- Availabitlity of many existing imputation methods
- Formula-data interface for model specification
- Packaging model, and CRAN

# Summary

# In my experience

## Peeling off a practical issue until a math problem remains

- Is essential to achieve true modularity and composability;
- Truly separates concerns between user needs (domain knowledge) and programming.
- Yields extensible solutions;
- Creates interfaces that are almost automatically user-friendly

# What makes R specifically suited for modernization?

## High level of abstraction

Higher order functions, reflexivity facilitate true **modularity** and **composability**.

# What makes R specifically suited for modernization?

## High level of abstraction

Higher order functions, reflexivity facilitate true **modularity** and **composability**.

## Functional language and reflexivity

Allows for quick development of **domain-specific languages** embedded into R

# What makes R specifically suited for modernization?

## High level of abstraction

Higher order functions, reflexivity facilitate true **modularity** and **composability**.

## Functional language and reflexivity

Allows for quick development of **domain-specific languages** embedded into R

## Abstractions for model-building

Formula-data interface facilitate **user-friendly interfaces**.

# What makes R specifically suited for modernization?

## High level of abstraction

Higher order functions, reflexivity facilitate true **modularity** and **composability**.

## Functional language and reflexivity

Allows for quick development of **domain-specific languages** embedded into R

## Abstractions for model-building

Formula-data interface facilitate **user-friendly interfaces**.

## Packaging system and CRAN

Nothing beats CRAN in terms of consistency accross dependencies, quality checks and **standardization**.

# What makes R specifically suited for modernization?

## High level of abstraction

Higher order functions, reflexivity facilitate true **modularity** and **composability**.

## Functional language and reflexivity

Allows for quick development of **domain-specific languages** embedded into R

## Abstractions for model-building

Formula-data interface facilitate **user-friendly interfaces**.

## Packaging system and CRAN

Nothing beats CRAN in terms of consistency accross dependencies, quality checks and **standardization**.

## **The R community (you!)**

# Thank you

markvanderloo.eu/publications.html