**################################################################**

**## ##**

**## CUTE LITTLE FUNCTIONS v1.2 ##**

**## ##**

**## Gael A. Millot ##**

**## ##**

**## Compatible with R v3.5.1 ##**

**## ##**

**################################################################**

**################################ OUTLINE ################################**

[################ Object analysis 1](#_Toc519288598)

[######## fun\_param\_check() #### Checking class, type, length, etc. of objects 1](#_Toc519288599)

[######## fun\_object\_info() #### Recovering object information 6](#_Toc519288600)

[######## fun\_1D\_comp() #### comparison of two 1D datasets (vectors, factors, 1D tables) 7](#_Toc519288601)

[######## fun\_2D\_comp() #### comparison of two 2D datasets (row & col names, dimensions, etc.) 11](#_Toc519288602)

[######## fun\_list\_comp() #### comparison of two lists 15](#_Toc519288603)

[################ Object modification 17](#_Toc519288604)

[######## fun\_dataframe\_flipping() #### flipping a data frame to have column name as a qualitative column and vice-versa 17](#_Toc519288605)

[######## fun\_refactorization() #### remove classes that are not anymore present in factors or factor comns in data frames 20](#_Toc519288606)

[######## fun\_rounding() #### Rounding number if decimal present 22](#_Toc519288607)

[######## fun\_90clock\_matrix\_rot() #### 90° clockwise matrix rotation 23](#_Toc519288608)

[######## fun\_hexa\_hsv\_color\_matrix() #### Conversion of a numeric matrix into hexadecimal color matrix 24](#_Toc519288609)

[################ Graphics 27](#_Toc519288610)

[######## fun\_window\_width\_resizing() #### window width depending on classes to plot 27](#_Toc519288611)

[######## fun\_open\_window() #### Open a GUI or pdf graphic window 28](#_Toc519288612)

[######## fun\_graph\_param\_prior\_plot() #### Graph param before plotting 31](#_Toc519288613)

[######## fun\_feature\_post\_plot() #### Graph param before plotting 34](#_Toc519288614)

[######## fun\_close\_specif\_window() #### Closing specific graphic windows 43](#_Toc519288615)

[################ Exporting results (text & tables) 44](#_Toc519288616)

[######## fun\_export\_data() #### Print string or data object into output file 44](#_Toc519288617)

**################################ FUNCTIONS ################################**

################ Object analysis

######## fun\_param\_check() #### Checking class, type, length, etc. of objects

**# Check OK: clear to go Apollo**

**fun\_param\_check <- function(data, data.name = NULL, class = NULL, typeof = NULL, mode = NULL, length = NULL, prop = NULL, double.as.integer.allowed = FALSE, options = NULL, all.options.in.data = FALSE, na.contain = FALSE, neg.values = TRUE, print = TRUE){**

*# AIM:*

*# Check the class, type, mode and length of the data argument*

*# Mainly used to check the arguments of other functions*

*# Check also other kind of data parameters, is it a proportion? Is it type double even if it is an integer?*

*# Check also the options of an argument of function*

*# If options = NULL, then at least class, type, mode or length must be non null*

*# If options is non null, then class, type and mode must be NULL, and length can be NULL or specified*

*# REQUIRED FUNCTIONS*

*# none*

*# ARGUMENTS*

*# data: object to test*

*# data.name: name of the object to test. If NULL, use the name of the object assigned to the data argument*

*# class: one of the class() result*

*# typeof: one of the typeof() result*

*# mode: one of the mode() result (for non vector object)*

*# length: length of the object*

*# prop: logical, are the numeric values between 0 and 1 (proportion)?*

*# double.as.integer.allowed: logical. If TRUE, no error is reported if argument is set to typeof = "integer" or class = "integer", while the reality is typeof = "double" or class = "numeric" but the numbers have a zero as modulo (remainder of a division). This means that i<-1 , which is typeof(i) -> "double" is considered as integer with double.as.integer.allowed = TRUE*

*# options: a vector of possible values for data*

*# all.options.in.data: If TRUE, all of the options must be present at least once in data, and nothing else. If FALSE, some of the options must be present in data, and nothing else*

*# na.contain: can data contains NA?*

*# neg.values: are negative numeric values authorized? BEWARE: only considered if set to FALSE, to check for non negative values when class is set to "numeric", "matrix", "array", "data.frame", "table", or typeof is set to "double", "integer", or mode is set to "numeric"*

*# print: print the error message if $problem is TRUE?*

*# RETURN*

*# a list containing:*

*# $problem: logical. Is there any problem detected ?*

*# $text: the problem detected*

*# $param.name: name of the checked parameter*

*# EXAMPLES*

*# test <- 1:3 ; fun\_param\_check(data = test, data.name = NULL, print = TRUE, options = NULL, all.options.in.data = FALSE, class = NULL, typeof = NULL, mode = NULL, prop = TRUE, double.as.integer.allowed = FALSE, length = NULL)*

*# test <- 1:3 ; fun\_param\_check(data = test, print = TRUE, class = "numeric", typeof = NULL, double.as.integer.allowed = FALSE)*

*# DEBUGGING*

*# data = 1:3 ; data.name = NULL ; print = TRUE; options = NULL ; all.options.in.data = FALSE ; class = "numeric" ; typeof = NULL ; mode = NULL ; prop = NULL ; double.as.integer.allowed = TRUE ; length = NULL # for function debugging*

*# argument checking*

*# source("C:/Users/Gael/Documents/Sources/debugging\_tools\_for\_r\_dev-v1.2/r\_debugging\_tools-v1.2.R") ; eval(parse(text = str\_basic\_arg\_check\_dev))*

**if( ! is.null(data.name)){**

**if( ! (length(data.name) == 1 & class(data.name) == "character")){**

**tempo.cat <- paste0("\n\n================\n\nERROR: data.name ARGUMENT MUST BE A SINGLE CHARACTER ELEMENT AND NOT ", paste(data.name, collapse = " "), "\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}**

**if(is.null(options) & is.null(class) & is.null(typeof) & is.null(mode) & is.null(prop) & is.null(length)){**

**tempo.cat <- paste0("\n\n================\n\nERROR: AT LEAST ONE OF THE options, class, typeof, mode, prop, OR length ARGUMENT MUST BE SPECIFIED\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**if( ! is.null(options) & ( ! is.null(class) | ! is.null(typeof) | ! is.null(mode) | ! is.null(prop))){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE class, typeof, mode AND prop ARGUMENTS MUST BE NULL IF THE option ARGUMENT IS SPECIFIED\nTHE option ARGUMENT MUST BE NULL IF THE class AND/OR typeof AND/OR mode AND/OR prop ARGUMENT IS SPECIFIED\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**if( ! (all(class(neg.values) == "logical") & length(neg.values) == 1 & any(is.na(neg.values)) != TRUE)){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE neg.values ARGUMENT MUST BE TRUE OR FALSE ONLY\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**if(neg.values == FALSE & is.null(class) & is.null(typeof) & is.null(mode)){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE neg.values ARGUMENT CANNOT BE SWITCHED TO FALSE IF class, typeof AND mode ARGUMENTS ARE NULL\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**if( ! is.null(class)){**

**if( ! all(class %in% c("logical", "integer", "numeric", "complex", "character", "matrix", "array", "data.frame", "list", "factor", "table", "expression", "name", "symbol", "function") & any(is.na(class)) != TRUE)){** *# not length == 1 here because ordered factors are class "factor" "ordered" (length == 2)*

**tempo.cat <- paste0("\n\n================\n\nERROR: class ARGUMENT MUST BE ONE OF THESE VALUE:\n\"logical\", \"integer\", \"numeric\", \"complex\", \"character\", \"matrix\", \"array\", \"data.frame\", \"list\", \"factor\", \"table\", \"expression\", \"name\", \"symbol\", \"function\" \n\n================\n\n")**

**stop(tempo.cat)**

**}**

**if(neg.values == FALSE & ! any(class %in% c("numeric", "matrix", "array", "data.frame", "table"))){**

**tempo.cat <- paste0("\n\n================\n\nERROR: class ARGUMENT CANNOT BE OTHER THAN \"numeric\", \"matrix\", \"array\", \"data.frame\", \"table\" IF neg.values ARGUMENT IS SWITCHED TO FALSE\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}**

**if( ! is.null(typeof)){**

**if( ! (all(typeof %in% c("logical", "integer", "double", "complex", "character", "list", "expression", "name", "symbol", "closure", "special", "builtin")) & length(typeof) == 1 & any(is.na(typeof)) != TRUE)){**

**tempo.cat <- paste0("\n\n================\n\nERROR: typeof ARGUMENT MUST BE ONE OF THESE VALUE:\n\"logical\", \"integer\", \"double\", \"complex\", \"character\", \"list\", \"expression\", \"name\", \"symbol\", \"closure\", \"special\", \"builtin\" \n\n================\n\n")**

**stop(tempo.cat)**

**}**

**if(neg.values == FALSE & ! typeof %in% c("double", "integer")){**

**tempo.cat <- paste0("\n\n================\n\nERROR: typeof ARGUMENT CANNOT BE OTHER THAN \"double\" OR \"integer\" IF neg.values ARGUMENT IS SWITCHED TO FALSE\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}**

**if( ! is.null(mode)){**

**if( ! (all(mode %in% c("logical", "numeric", "complex", "character", "list", "expression", "name", "symbol", "function")) & length(mode) == 1 & any(is.na(mode)) != TRUE)){**

**tempo.cat <- paste0("\n\n================\n\nERROR: mode ARGUMENT MUST BE ONE OF THESE VALUE:\n\"logical\", \"numeric\", \"complex\", \"character\", \"list\", \"expression\", \"name\", \"symbol\", \"function\"\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**if(neg.values == FALSE & mode != "numeric"){**

**tempo.cat <- paste0("\n\n================\n\nERROR: mode ARGUMENT CANNOT BE OTHER THAN \"numeric\" IF neg.values ARGUMENT IS SWITCHED TO FALSE\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}**

**if( ! is.null(length)){**

**if( ! (is.numeric(length) & length(length) == 1 & ! grepl(length, pattern = "\\.") & any(is.na(length)) != TRUE)){**

**tempo.cat <- paste0("\n\n================\n\nERROR: length ARGUMENT MUST BE A SINGLE INTEGER VALUE\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}**

**if( ! is.null(prop)){**

**if( ! (is.logical(prop) | length(prop) == 1 & any(is.na(prop)) != TRUE)){**

**tempo.cat <- paste0("\n\n================\n\nERROR: prop ARGUMENT MUST BE TRUE OR FALSE ONLY\n\n================\n\n")**

**stop(tempo.cat)**

**}else if(prop == TRUE){**

**if( ! is.null(class)){**

**if( ! any(class %in% c("numeric", "matrix", "array", "data.frame", "table"))){**

**tempo.cat <- paste0("\n\n================\n\nERROR: class ARGUMENT CANNOT BE OTHER THAN \"numeric\", \"matrix\", \"array\", \"data.frame\", \"table\" IF prop ARGUMENT IS TRUE\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}**

**if( ! is.null(mode)){**

**if(mode != "numeric"){**

**tempo.cat <- paste0("\n\n================\n\nERROR: mode ARGUMENT CANNOT BE OTHER THAN \"numeric\" IF prop ARGUMENT IS TRUE\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}**

**if( ! is.null(typeof)){**

**if(typeof != "double"){**

**tempo.cat <- paste0("\n\n================\n\nERROR: typeof ARGUMENT CANNOT BE OTHER THAN \"double\" IF prop ARGUMENT IS TRUE\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}**

**}**

**}**

**if( ! (all(class(double.as.integer.allowed) == "logical") & length(double.as.integer.allowed) == 1 & any(is.na(double.as.integer.allowed)) != TRUE)){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE double.as.integer.allowed ARGUMENT MUST BE TRUE OR FALSE ONLY\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**if( ! (is.logical(all.options.in.data) & length(all.options.in.data) == 1 & any(is.na(all.options.in.data)) != TRUE)){**

**tempo.cat <- paste0("\n\n================\n\nERROR: all.options.in.data ARGUMENT MUST BE A SINGLE LOGICAL VALUE (TRUE OR FALSE ONLY)\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**if( ! (all(class(na.contain) == "logical") & length(na.contain) == 1 & any(is.na(na.contain)) != TRUE)){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE na.contain ARGUMENT MUST BE TRUE OR FALSE ONLY\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**if( ! (all(class(print) == "logical") & length(print) == 1 & any(is.na(print)) != TRUE)){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE print ARGUMENT MUST BE TRUE OR FALSE ONLY\n\n================\n\n")**

**stop(tempo.cat)**

**}**

*# end argument checking*

**if(is.null(data.name)){**

**data.name <- deparse(substitute(data))**

**}**

**problem <- FALSE**

**text <- paste0("NO PROBLEM DETECTED FOR THE ", data.name, " PARAMETER")**

**if( ! is.null(options)){**

**text <- ""**

**if( ! all(data %in% options)){**

**problem <- TRUE**

**text <- paste0("PROBLEM: THE ", data.name, " PARAMETER MUST BE SOME OF THESE OPTIONS: ", paste(options, collapse = " "), "\nTHE PROBLEMATIC ELEMENTS OF ", data.name, " ARE: ", paste(unique(data[ ! (data %in% options)]), collapse = " "))**

**}**

**if(all.options.in.data == TRUE){**

**if( ! all(options %in% data)){**

**problem <- TRUE**

**if(text == ""){**

**text <- paste0("PROBLEM: THE ", data.name, " PARAMETER MUST BE SOME OF THESE OPTIONS: ", paste(options, collapse = " "), "\nTHE PROBLEMATIC ELEMENTS OF ", data.name, " ARE: ", unique(data[ ! (data %in% options)]))**

**}else{**

**text <- paste0(text, "\nPROBLEM: THE ", data.name, " PARAMETER MUST BE SOME OF THESE OPTIONS: ", paste(options, collapse = " "), "\nTHE PROBLEMATIC ELEMENTS OF ", data.name, " ARE: ", unique(data[ ! (data %in% options)]))**

**}**

**}**

**}**

**if( ! is.null(length)){**

**if(length(data) != length){**

**problem <- TRUE**

**if(text == ""){**

**text <- paste0("PROBLEM: THE LENGTH OF ", data.name, " MUST BE ", length, " AND NOT ", length(data))**

**}else{**

**text <- paste0(text, "\nPROBLEM: THE LENGTH OF ", data.name, " MUST BE ", length, " AND NOT ", length(data))**

**}**

**}**

**}**

**if(text == ""){**

**text <- paste0("NO PROBLEM DETECTED FOR THE ", data.name, " PARAMETER")**

**}**

**}**

**arg.names <- c("class", "typeof", "mode", "length")**

**if(is.null(options)){**

**for(i2 in 1:length(arg.names)){**

**if( ! is.null(get(arg.names[i2]))){**

*# script to execute*

**tempo.script <- '**

**problem <- TRUE ;**

**if(identical(text, paste0("NO PROBLEM DETECTED FOR THE ", data.name, " PARAMETER"))){**

**text <- paste0("PROBLEM: THE ", data.name, " PARAMETER MUST BE ") ;**

**}else{**

**text <- paste0(text, " AND ");**

**}**

**text <- paste0(text, toupper(arg.names[i2]), " ", get(arg.names[i2]))**

**'**

**if(typeof(data) == "double" & double.as.integer.allowed == TRUE & ((arg.names[i2] == "class" & get(arg.names[i2]) == "integer") | (arg.names[i2] == "typeof" & get(arg.names[i2]) == "integer"))){**

**if(! all(data%%1 == 0)){** *# to check integers (use %%, meaning the remaining of a division): see the precedent line*

**eval(parse(text = tempo.script))** *# execute tempo.script*

**}**

**}else if(eval(parse(text = paste0(arg.names[i2], "(data)"))) != get(arg.names[i2])){**

**eval(parse(text = tempo.script))** *# execute tempo.script*

**}**

**}**

**}**

**}**

**if( ! is.null(prop)){**

**if(prop == TRUE){**

**if(any(data < 0 | data > 1, na.rm = TRUE)){**

**problem <- TRUE**

**if(identical(text, paste0("NO PROBLEM DETECTED FOR THE ", data.name, " PARAMETER"))){**

**text <- paste0("PROBLEM: ")**

**}else{**

**text <- paste0(text, " AND ")**

**}**

**text <- paste0(text, "THE ", data.name, " PARAMETER MUST BE DECIMAL VALUES BETWEEN 0 AND 1")**

**}**

**}**

**}**

**if(na.contain == FALSE & any(is.na(data)) == TRUE){**

**problem <- TRUE**

**if(identical(text, paste0("NO PROBLEM DETECTED FOR THE ", data.name, " PARAMETER"))){**

**text <- paste0("PROBLEM: ")**

**}else{**

**text <- paste0(text, " AND ")**

**}**

**text <- paste0(text, "THE ", data.name, " PARAMETER CONTAINS NA WHILE NOT AUTHORIZED (na.contain ARGUMENT SET TO FALSE)")**

**}**

**if(neg.values == FALSE){**

**if(any(data < 0, na.rm = TRUE)){**

**problem <- TRUE**

**if(identical(text, paste0("NO PROBLEM DETECTED FOR THE ", data.name, " PARAMETER"))){**

**text <- paste0("PROBLEM: ")**

**}else{**

**text <- paste0(text, " AND ")**

**}**

**text <- paste0(text, "THE ", data.name, " PARAMETER MUST NON NEGATIVE NUMERIC VALUES")**

**}**

**}**

**if(print == TRUE & problem == TRUE){**

**cat(paste0("\n\n================\n\n", text, "\n\n================\n\n"))**

**}**

**output <- list(problem = problem, text = text, param.name = data.name)**

**return(output)**

**}**

######## fun\_object\_info() #### Recovering object information

**# Check OK: clear to go Apollo**

**fun\_object\_info <- function(data){**

*# AIM:*

*# provide info for a single object*

*# REQUIRED FUNCTIONS*

*# none*

*# ARGUMENTS*

*# data: object to test*

*# RETURN*

*# a list containing the info*

*# EXAMPLES*

*# fun\_object\_info(data = 1:3)*

*# DEBUGGING*

*# data = NULL # for function debugging*

*# data = 1:3 # for function debugging*

*# data = matrix(1:3) # for function debugging*

*# data = data.frame(a = 1:3) # for function debugging*

*# data = factor(1:3) # for function debugging*

*# data = list(1:3) # for function debugging*

*# argument checking*

*# source("C:/Users/Gael/Documents/Sources/debugging\_tools\_for\_r\_dev-v1.2/r\_debugging\_tools-v1.2.R") ; eval(parse(text = str\_basic\_arg\_check\_dev)) # to check arguments status and if they have been checked using fun\_param\_check()*

*# end argument checking*

**data.name <- deparse(substitute(data))**

**output <- list("FILE\_NAME" = data.name)**

**tempo <- list("FILE\_HEAD" = head(data))**

**output <- c(output, tempo)**

**if( ! is.null(data)){**

**tempo <- list("FILE\_TAIL" = tail(data))**

**output <- c(output, tempo)**

**if( ! is.null(dim(data))){**

**tempo <- list("FILE\_DIMENSION" = dim(data))**

**names(tempo[[1]]) <- c("NROW", "NCOL")**

**output <- c(output, tempo)**

**}**

**tempo <- list("SUMMARY" = summary(data))**

**output <- c(output, tempo)**

**}**

**if(class(data) == "data.frame" | class(data) == "matrix"){**

**tempo <- list("COLUM\_NAMES" = names(data))**

**output <- c(output, tempo)**

**}**

**if(class(data) == "data.frame"){**

**tempo <- list("STRUCTURE" = ls.str(data))**

**output <- c(output, tempo)**

**tempo <- list("COLUMN\_TYPE" = sapply(data, FUN = "typeof"))**

**output <- c(output, tempo)**

**}**

**if(class(data) == "list"){**

**tempo <- list("COMPARTMENT\_NAMES" = names(data))**

**output <- c(output, tempo)**

**}**

**return(output)**

**}**

######## fun\_1D\_comp() #### comparison of two 1D datasets (vectors, factors, 1D tables)

**# Check OK: clear to go Apollo**

**fun\_1D\_comp <- function(data1, data2){**

*# AIM:*

*# compare two 1D dataset (vector of factor or 1D table) of the same class or not. Check and report in a list if the 2 datasets have:*

*# same class*

*# same elements*

*# same element names*

*# REQUIRED FUNCTIONS*

*# none*

*# ARGUMENTS*

*# data1: vector or factor or 1D table*

*# data2: vector or factor or 1D table*

*# RETURN*

*# a list containing:*

*# $same.class: logical. Are class identical?*

*# $class: class of the 2 datasets (NULL otherwise)*

*# $same.length: logical. Are number of elements identical?*

*# $length: number of elements in the 2 datasets (NULL otherwise)*

*# $same.levels: logical. Are levels identical? (NULL if data1 and data2 are not factors)*

*# $levels: levels of the 2 datasets if identical (NULL otherwise or NULL if data1 and data2 are not factors)*

*# $any.id.levels: logical. Is there any identical levels? (NULL if data1 and data2 are not factors)*

*# $same.levels.pos1: position, in data1, of the levels identical in data2 (NULL if data1 and data2 are not factors)*

*# $same.levels.pos2: position, in data2, of the levels identical in data1 (NULL if data1 and data2 are not factors)*

*# $same.name: logical. Are element names identical ?*

*# $name: name of elements of the 2 datasets if identical (NULL otherwise)*

*# $any.id.name: logical. Is there any element names identical ?*

*# $same.name.pos1: position, in data1, of the element names identical in data2*

*# $same.name.pos2: position, in data2, of the elements names identical in data1*

*# $any.id.element: logical. is there any identical elements ?*

*# $same.element.pos1: position, in data1, of the elements identical in data2*

*# $same.element.pos2: position, in data2, of the elements identical in data1*

*# $identical.object: logical. Are objects identical (kind of object, element names and content)?*

*# $identical.content: logical. Are content objects identical (identical elements excluding kind of object and element names)?*

*# EXAMPLES*

*# obs1 = 1:5 ; obs2 = 1:5 ; names(obs1) <- LETTERS[1:5] ; names(obs2) <- LETTERS[1:5] ; fun\_1D\_comp(obs1, obs2)*

*# obs1 = 1:5 ; obs2 = 1:5 ; names(obs1) <- LETTERS[1:5] ; fun\_1D\_comp(obs1, obs2)*

*# obs1 = factor(LETTERS[1:5]) ; obs2 = factor(LETTERS[1:5]) ; fun\_1D\_comp(obs1, obs2)*

*# obs1 = factor(LETTERS[1:5]) ; obs2 = factor(LETTERS[10:11]) ; fun\_1D\_comp(obs1, obs2)*

*# obs1 = factor(LETTERS[1:5]) ; obs2 = factor(LETTERS[4:7]) ; fun\_1D\_comp(obs1, obs2)*

*# obs1 = 1:5 ; obs2 = factor(LETTERS[1:5]) ; fun\_1D\_comp(obs1, obs2)*

*# obs1 = 1:5 ; obs2 = 1.1:6.1 ; fun\_1D\_comp(obs1, obs2)*

*# obs1 = as.table(1:5); obs2 = as.table(1:5) ; fun\_1D\_comp(obs1, obs2)*

*# obs1 = as.table(1:5); obs2 = 1:5 ; fun\_1D\_comp(obs1, obs2)*

*# DEBUGGING*

*# data1 = 1:5 ; data2 = 1:5 ; names(data1) <- LETTERS[1:5] ; names(data2) <- LETTERS[1:5] # for function debugging*

*# argument checking*

*# source("C:/Users/Gael/Documents/Sources/debugging\_tools\_for\_r\_dev-v1.2/r\_debugging\_tools-v1.2.R") ; eval(parse(text = str\_basic\_arg\_check\_dev)) # to check arguments status and if they have been checked using fun\_param\_check()*

**if( ! any(class(data1) %in% c("logical", "integer", "numeric", "character", "factor", "table"))){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE data1 ARGUMENT MUST BE A NON NULL VECTOR, FACTOR OR 1D TABLE\n\n================\n\n")**

**stop(tempo.cat)**

**}else if(all(class(data1) %in% "table")){**

**if(length(dim(data1)) > 1){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE data1 ARGUMENT MUST BE A 1D TABLE\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}**

**if( ! any(class(data2) %in% c("logical", "integer", "numeric", "character", "factor", "table"))){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE data2 ARGUMENT MUST BE A NON NULL VECTOR, FACTOR OR 1D TABLE\n\n================\n\n")**

**stop(tempo.cat)**

**}else if(all(class(data2) %in% "table")){**

**if(length(dim(data2)) > 1){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE data2 ARGUMENT MUST BE A 1D TABLE\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}**

*# end argument checking*

**same.class <- NULL**

**class <- NULL**

**same.length <- NULL**

**length <- NULL**

**same.levels <- NULL**

**levels <- NULL**

**any.id.levels <- NULL**

**same.levels.pos1 <- NULL**

**same.levels.pos2 <- NULL**

**same.name <- NULL**

**name <- NULL**

**any.id.name <- NULL**

**same.name.pos1 <- NULL**

**same.name.pos2 <- NULL**

**any.id.element <- NULL**

**same.element.pos1 <- NULL**

**same.element.pos2 <- NULL**

**identical.object <- NULL**

**identical.content <- NULL**

**if(identical(data1, data2)){**

**same.class <- TRUE**

**class <- class(data1)**

**same.length <- TRUE**

**length <- length(data1)**

**if(any(class(data1) %in% "factor")){**

**same.levels <- TRUE**

**levels <- levels(data1)**

**any.id.levels <- TRUE**

**same.levels.pos1 <- 1:length(levels(data1))**

**same.levels.pos2 <- 1:length(levels(data2))**

**}**

**if( ! is.null(names(data1))){**

**same.name <- TRUE**

**name <- names(data1)**

**any.id.name <- TRUE**

**same.name.pos1 <- 1:length(data1)**

**same.name.pos2 <- 1:length(data2)**

**}**

**any.id.element <- TRUE**

**same.element.pos1 <- 1:length(data1)**

**same.element.pos2 <- 1:length(data2)**

**identical.object <- TRUE**

**identical.content <- TRUE**

**}else{**

**identical.object <- FALSE**

**if( ! identical(class(data1), class(data2))){**

**same.class <- FALSE**

**}else{**

**same.class <- TRUE**

**class <- class(data1)**

**}**

**if( ! identical(length(data1), length(data2))){**

**same.length<- FALSE**

**}else{**

**same.length<- TRUE**

**length <- length(data1)**

**}**

**if(any(class(data1) %in% "factor") & any(class(data2) %in% "factor")){**

**if( ! identical(levels(data1), levels(data2))){**

**same.levels <- FALSE**

**}else{**

**same.levels <- TRUE**

**levels <- levels(data1)**

**}**

**any.id.levels <- FALSE**

**if(any(levels(data1) %in% levels(data2))){**

**any.id.levels <- TRUE**

**same.levels.pos1 <- which(levels(data1) %in% levels(data2))**

**}**

**if(any(levels(data2) %in% levels(data1))){**

**any.id.levels <- TRUE**

**same.levels.pos2 <- which(levels(data2) %in% levels(data1))**

**}**

**}**

**if(any(class(data1) %in% "factor")){** *# to compare content*

**data1 <- as.character(data1)**

**}**

**if(any(class(data2) %in% "factor")){** *# to compare content*

**data2 <- as.character(data2)**

**}**

**if( ! (is.null(names(data1)) & is.null(names(data2)))){**

**if( ! identical(names(data1), names(data2))){**

**same.name <- FALSE**

**}else{**

**same.name <- TRUE**

**name <- names(data1)**

**}**

**any.id.name <- FALSE**

**if(any(names(data1) %in% names(data2))){**

**any.id.name <- TRUE**

**same.name.pos1 <- which(names(data1) %in% names(data2))**

**}**

**if(any(names(data2) %in% names(data1))){**

**any.id.name <- TRUE**

**same.name.pos2 <- which(names(data2) %in% names(data1))**

**}**

**}**

**any.id.element <- FALSE**

**if(any(data1 %in% data2)){**

**any.id.element <- TRUE**

**same.element.pos1 <- which(data1 %in% data2)**

**}**

**if(any(data2 %in% data1)){**

**any.id.element <- TRUE**

**same.element.pos2 <- which(data2 %in% data1)**

**}**

**if(same.length == TRUE & ! all(is.null(same.element.pos1), is.null(same.element.pos2))){**

**names(same.element.pos1) <- NULL**

**names(same.element.pos2) <- NULL**

**if(identical(same.element.pos1, same.element.pos2)){**

**identical.content <- TRUE**

**}else{**

**identical.content <- FALSE**

**}**

**}else{**

**identical.content <- FALSE**

**}**

**}**

**output <- list(same.class = same.class, class = class, same.length = same.length, length = length, same.levels = same.levels, levels = levels, any.id.levels = any.id.levels, same.levels.pos1 = same.levels.pos1, same.levels.pos2 = same.levels.pos2, same.name = same.name, name = name, any.id.name = any.id.name, same.name.pos1 = same.name.pos1, same.name.pos2 = same.name.pos2, any.id.element = any.id.element, same.element.pos1 = same.element.pos1, same.element.pos2 = same.element.pos2, identical.object = identical.object, identical.content = identical.content)**

**return(output)**

**}**

######## fun\_2D\_comp() #### comparison of two 2D datasets (row & col names, dimensions, etc.)

**# Check OK: clear to go Apollo**

**fun\_2D\_comp <- function(data1, data2){**

*# AIM:*

*# compare two 2D dataset of the same class or not. Check and report in a list if the 2 datasets have:*

*# same row names*

*# same column names*

*# same row number*

*# same column number*

*# potential identical rows between the 2 datasets*

*# potential identical columns between the 2 datasets*

*# REQUIRED FUNCTIONS*

*# none*

*# ARGUMENTS*

*# data1: matrix, data frame or table*

*# data2: matrix, data frame or table*

*# RETURN*

*# a list containing:*

*# $same.class: logical. Are class identical ?*

*# $class: classes of the 2 datasets (NULL otherwise)*

*# $same.dim: logical. Are dimension identical ?*

*# $dim: dimension of the 2 datasets (NULL otherwise)*

*# $same.row.nb: logical. Are number of rows identical ?*

*# $row.nb: nb of rows of the 2 datasets if identical (NULL otherwise)*

*# $same.col.nb: logical. Are number of columns identical ?*

*# $col.nb: nb of columns of the 2 datasets if identical (NULL otherwise)*

*# $same.row.name: logical. Are row names identical ?*

*# $row.name: name of rows of the 2 datasets if identical (NULL otherwise)*

*# $same.col.name: logical. Are column names identical ?*

*# $col.name: name of columns of the 2 datasets if identical (NULL otherwise)*

*# $any.id.row: logical. is there identical rows ?*

*# $same.row.pos1: position, in data1, of the rows identical in data2*

*# $same.row.pos2: position, in data2, of the rows identical in data1*

*# $any.id.col: logical. is there identical columns ?*

*# $same.col.pos1: position in data1 of the cols identical in data2*

*# $same.col.pos2: position in data2 of the cols identical in data1*

*# $identical.object: logical. Are objects identical (including row & column names)?*

*# $identical.content: logical. Are content objects identical (identical excluding row & column names)?*

*# EXAMPLES*

*# obs1 = matrix(1:10, ncol = 5, dimnames = list(letters[1:2], LETTERS[1:5])) ; obs2 = as.data.frame(matrix(1:10, ncol = 5, dimnames = list(letters[1:2], LETTERS[1:5]))) ; fun\_2D\_comp(obs1, obs2)*

*# DEBUGGING*

*# data1 = matrix(1:10, ncol = 5) ; data2 = matrix(1:10, ncol = 5) # for function debugging*

*# data1 = matrix(1:10, ncol = 5, dimnames = list(letters[1:2], LETTERS[1:5])) ; data2 = matrix(1:10, ncol = 5, dimnames = list(letters[1:2], LETTERS[1:5])) # for function debugging*

*# data1 = matrix(1:15, byrow = TRUE, ncol = 5, dimnames = list(letters[1:3], LETTERS[1:5])) ; data2 = matrix(1:10, byrow = TRUE, ncol = 5, dimnames = list(letters[1:2], LETTERS[1:5])) # for function debugging*

*# data1 = matrix(1:15, ncol = 5, dimnames = list(letters[1:3], LETTERS[1:5])) ; data2 = matrix(1:10, ncol = 5, dimnames = list(letters[1:2], LETTERS[1:5])) # for function debugging*

*# data1 = matrix(1:15, ncol = 5, dimnames = list(paste0("A", letters[1:3]), LETTERS[1:5])) ; data2 = matrix(1:10, ncol = 5, dimnames = list(letters[1:2], LETTERS[1:5])) # for function debugging*

*# data1 = matrix(1:15, ncol = 5, dimnames = list(letters[1:3], LETTERS[1:5])) ; data2 = matrix(1:12, ncol = 4, dimnames = list(letters[1:3], LETTERS[1:4])) # for function debugging*

*# data1 = matrix(1:10, ncol = 5, dimnames = list(letters[1:2], LETTERS[1:5])) ; data2 = matrix(101:110, ncol = 5, dimnames = list(letters[1:2], LETTERS[1:5])) # for function debugging*

*# data1 = data.frame(a = 1:3, b= letters[1:3], row.names = LETTERS[1:3]) ; data2 = data.frame(A = 1:3, B= letters[1:3]) # for function debugging*

*# data1 = matrix(1:10, ncol = 5, dimnames = list(letters[1:2], LETTERS[1:5])) ; data2 = as.data.frame(matrix(1:10, ncol = 5, dimnames = list(letters[1:2], LETTERS[1:5]))) # for function debugging*

*# data1 = table(Exp1 = c("A", "A", "A", "B", "B", "B"), Exp2 = c("A1", "B1", "A1", "C1", "C1", "B1")) ; data2 = data.frame(A = 1:3, B= letters[1:3]) # for function debugging*

*# argument checking*

*# source("C:/Users/Gael/Documents/Sources/debugging\_tools\_for\_r\_dev-v1.2/r\_debugging\_tools-v1.2.R") ; eval(parse(text = str\_basic\_arg\_check\_dev)) # to check arguments status and if they have been checked using fun\_param\_check()*

**if( ! any(class(data1) %in% c("matrix", "data.frame", "table"))){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE data1 ARGUMENT MUST BE A MATRIX, DATA FRAME OR TABLE\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**if( ! any(class(data2) %in% c("matrix", "data.frame", "table"))){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE data2 ARGUMENT MUST BE A MATRIX, DATA FRAME OR TABLE\n\n================\n\n")**

**stop(tempo.cat)**

**}**

*# end argument checking*

**same.class <- NULL**

**class <- NULL**

**same.dim <- NULL**

**dim <- NULL**

**same.row.nb <- NULL**

**row.nb <- NULL**

**same.col.nb <- NULL**

**col.nb <- NULL**

**same.row.name <- NULL**

**row.name <- NULL**

**same.col.name <- NULL**

**col.name <- NULL**

**any.id.row <- NULL**

**same.row.pos1 <- NULL**

**same.row.pos2 <- NULL**

**any.id.col <- NULL**

**same.col.pos1 <- NULL**

**same.col.pos2 <- NULL**

**identical.object <- NULL**

**identical.content <- NULL**

**if(identical(data1, data2) & any(class(data1) %in% c("matrix", "data.frame", "table"))){**

**same.class <- TRUE**

**class <- class(data1)**

**same.dim <- TRUE**

**dim <- dim(data1)**

**same.row.nb <- TRUE**

**row.nb <- nrow(data1)**

**same.col.nb <- TRUE**

**col.nb <- ncol(data1)**

**same.row.name <- TRUE**

**row.name <- dimnames(data1)[[1]]**

**same.col.name <- TRUE**

**col.name <- dimnames(data1)[[2]]**

**any.id.row <- TRUE**

**same.row.pos1 <- 1:row.nb**

**same.row.pos2 <- 1:row.nb**

**any.id.col <- TRUE**

**same.col.pos1 <- 1:col.nb**

**same.col.pos2 <- 1:col.nb**

**identical.object <- TRUE**

**identical.content <- TRUE**

**}else{**

**identical.object <- FALSE**

**if(all(class(data1) == "table") & length(dim(data1)) == 1){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE data1 ARGUMENT IS A 1D TABLE. USE THE info\_1D\_dataset\_fun FUNCTION\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**if(all(class(data2) == "table") & length(dim(data2)) == 1){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE data2 ARGUMENT IS A 1D TABLE. USE THE info\_1D\_dataset\_fun FUNCTION\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**if( ! identical(class(data1), class(data2))){**

**same.class <- FALSE**

**}else if( ! any(class(data1) %in% c("matrix", "data.frame", "table"))){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE data1 AND data2 ARGUMENTS MUST BE EITHER MATRIX, DATA FRAME OR TABLE\n\n================\n\n")**

**stop(tempo.cat)**

**}else{**

**same.class <- TRUE**

**class <- class(data1)**

**}**

**if( ! identical(dim(data1), dim(data2))){**

**same.dim <- FALSE**

**}else{**

**same.dim <- TRUE**

**dim <- dim(data1)**

**}**

**if( ! identical(nrow(data1), nrow(data2))){**

**same.row.nb <- FALSE**

**}else{**

**same.row.nb <- TRUE**

**row.nb <- nrow(data1)**

**}**

**if( ! identical(ncol(data1), ncol(data2))){**

**same.col.nb <- FALSE**

**}else{**

**same.col.nb <- TRUE**

**col.nb <- ncol(data1)**

**}**

*# row and col names*

**if(is.null(dimnames(data1)) & is.null(dimnames(data2))){**

**same.row.name <- TRUE**

**same.col.name <- TRUE**

*# row and col names remain NULL*

**}else if((is.null(dimnames(data1)) & ! is.null(dimnames(data2))) | ( ! is.null(dimnames(data1)) & is.null(dimnames(data2)))){**

**same.row.name <- FALSE**

**same.col.name <- FALSE**

*# row and col names remain NULL*

**}else{**

**if( ! identical(dimnames(data1)[[1]], dimnames(data2)[[1]])){**

**same.row.name <- FALSE**

*# row names remain NULL*

**}else{**

**same.row.name <- TRUE**

**row.name <- dimnames(data1)[[1]]**

**}**

**if( ! identical(dimnames(data1)[[2]], dimnames(data2)[[2]])){**

**same.col.name <- FALSE**

*# col names remain NULL*

**}else{**

**same.col.name <- TRUE**

**col.name <- dimnames(data1)[[2]]**

**}**

**}**

*# identical row and col content*

**if(all(class(data1) == "table")){**

**as.data.frame(matrix(data1, ncol = ncol(data1)), stringsAsFactors = FALSE)**

**}else if(all(class(data1) == "matrix")){**

**data1 <- as.data.frame(data1, stringsAsFactors = FALSE)**

**}else if(all(class(data1) == "data.frame")){**

**data1 <- data.frame(lapply(data1, as.character), stringsAsFactors=FALSE)**

**}**

**if(all(class(data2) == "table")){**

**as.data.frame(matrix(data2, ncol = ncol(data2)), stringsAsFactors = FALSE)**

**}else if(all(class(data2) == "matrix")){**

**data2 <- as.data.frame(data2, stringsAsFactors = FALSE)**

**}else if(all(class(data2) == "data.frame")){**

**data2 <- data.frame(lapply(data2, as.character), stringsAsFactors=FALSE)**

**}**

**row.names(data1) <- paste0("A", 1:nrow(data1))**

**row.names(data2) <- paste0("A", 1:nrow(data2))**

**if(same.col.nb == TRUE){** *# because if not the same col nb, the row cannot be identical*

**same.row.pos1 <- suppressWarnings(which(mapply(FUN = identical, c(as.data.frame(t(data1), stringsAsFactors = FALSE)), c(as.data.frame(t(data2), stringsAsFactors = FALSE)))))**

**same.row.pos2 <- suppressWarnings(which(mapply(FUN = identical, c(as.data.frame(t(data2), stringsAsFactors = FALSE)), c(as.data.frame(t(data1), stringsAsFactors = FALSE)))))**

**names(same.row.pos1) <- NULL**

**names(same.row.pos2) <- NULL**

**if(all(is.na(same.row.pos1))){**

**same.row.pos1 <- NULL**

**}else{**

**same.row.pos1 <- same.row.pos1[ ! is.na(same.row.pos1)]**

**any.id.row <- TRUE**

**}**

**if(all(is.na(same.row.pos2))){**

**same.row.pos2 <- NULL**

**}else{**

**same.row.pos2 <- same.row.pos2[ ! is.na(same.row.pos2)]**

**any.id.row <- TRUE**

**}**

**if(is.null(same.row.pos1) & is.null(same.row.pos2)){**

**any.id.row <- FALSE**

**}**

**}else{**

**any.id.row <- FALSE**

*# same.row.pos1 and 2 remain NULL*

**}**

**if(same.row.nb == TRUE){** *# because if not the same row nb, the col cannot be identical*

**same.col.pos1 <- suppressWarnings(which(mapply(FUN = identical, c(data1), c(data2))))**

**same.col.pos2 <- suppressWarnings(which(mapply(FUN = identical, c(data2), c(data1))))**

**names(same.col.pos1) <- NULL**

**names(same.col.pos2) <- NULL**

**if(all(is.na(same.col.pos1))){**

**same.col.pos1 <- NULL**

**}else{**

**same.col.pos1 <- same.col.pos1[ ! is.na(same.col.pos1)]**

**any.id.col <- TRUE**

**}**

**if(all(is.na(same.col.pos2))){**

**same.col.pos2 <- NULL**

**}else{**

**same.col.pos2 <- same.col.pos2[ ! is.na(same.col.pos2)]**

**any.id.col <- TRUE**

**}**

**if(is.null(same.col.pos1) & is.null(same.col.pos2)){**

**any.id.col <- FALSE**

**}**

**}else{**

**any.id.col <- FALSE**

*# same.col.pos1 and 2 remain NULL*

**}**

**if(same.dim == TRUE & ! all(is.null(same.row.pos1), is.null(same.row.pos2), is.null(same.col.pos1), is.null(same.col.pos2))){** *# same.dim == TRUE means that same.row.nb == TRUE and same.col.nb == TRUE, meaning that row.nb != NULL and col.nb != NULL. Thus, no need to include these checkings*

**if(identical(same.row.pos1, 1:row.nb) & identical(same.row.pos2, 1:row.nb) & identical(same.col.pos1, 1:col.nb) & identical(same.col.pos2, 1:col.nb)){**

**identical.content <- TRUE**

**}**

**}else{**

**identical.content <- FALSE**

**}**

**}**

**output <- list(same.class = same.class, class = class, same.dim = same.dim, dim = dim, same.row.nb = same.row.nb, row.nb = row.nb, same.col.nb = same.col.nb , col.nb = col.nb, same.row.name = same.row.name, row.name = row.name, same.col.name = same.col.name, col.name = col.name, any.id.row = any.id.row, same.row.pos1 = same.row.pos1, same.row.pos2 = same.row.pos2, any.id.col = any.id.col, same.col.pos1 = same.col.pos1, same.col.pos2 = same.col.pos2, identical.object = identical.object, identical.content = identical.content)**

**return(output)**

**}**

######## fun\_list\_comp() #### comparison of two lists

**# Check OK: clear to go Apollo**

**fun\_list\_comp <- function(data1, data2){**

*# AIM:*

*# compare two lists. Check and report in a list if the 2 datasets have:*

*#same length*

*# same names*

*# same compartments*

*# REQUIRED FUNCTIONS*

*# none*

*# ARGUMENTS*

*# data1: list*

*# data2: list*

*# RETURN*

*# a list containing:*

*# $same.length: logical. Are number of elements identical?*

*# $length: number of elements in the 2 datasets (NULL otherwise)*

*# $same.name: logical. Are element names identical ?*

*# $name: name of elements of the 2 datasets if identical (NULL otherwise)*

*# $any.id.name: logical. Is there any element names identical ?*

*# $same.name.pos1: position, in data1, of the element names identical in data2*

*# $same.name.pos2: position, in data2, of the compartment names identical in data1*

*# $any.id.compartment: logical. is there any identical compartments ?*

*# $same.compartment.pos1: position, in data1, of the compartments identical in data2*

*# $same.compartment.pos2: position, in data2, of the compartments identical in data1*

*# $identical.object: logical. Are objects identical (kind of object, compartment names and content)?*

*# $identical.content: logical. Are content objects identical (identical compartments excluding compartment names)?*

*# EXAMPLES*

*# obs1 = list(a = 1:5, b = LETTERS[1:2], d = matrix(1:6)) ; obs2 = list(a = 1:5, b = LETTERS[1:2], d = matrix(1:6)) ; fun\_list\_comp(obs1, obs2)*

*# obs1 = list(1:5, LETTERS[1:2]) ; obs2 = list(a = 1:5, b = LETTERS[1:2]) ; fun\_list\_comp(obs1, obs2)*

*# obs1 = list(b = 1:5, c = LETTERS[1:2]) ; obs2 = list(a = 1:5, b = LETTERS[1:2], d = matrix(1:6)) ; fun\_list\_comp(obs1, obs2)*

*# obs1 = list(b = 1:5, c = LETTERS[1:2]) ; obs2 = list(LETTERS[5:9], matrix(1:6), 1:5) ; fun\_list\_comp(obs1, obs2)*

*# DEBUGGING*

*# data1 = list(a = 1:5, b = LETTERS[1:2], d = matrix(1:6)) ; data2 = list(a = 1:5, b = LETTERS[1:2], d = matrix(1:6)) # for function debugging*

*# data1 = list(a = 1:5, b = LETTERS[1:2]) ; data2 = list(a = 1:5, b = LETTERS[1:2], d = matrix(1:6)) # for function debugging*

*# argument checking*

*# source("C:/Users/Gael/Documents/Sources/debugging\_tools\_for\_r\_dev-v1.2/r\_debugging\_tools-v1.2.R") ; eval(parse(text = str\_basic\_arg\_check\_dev)) # to check arguments status and if they have been checked using fun\_param\_check()*

**if( ! any(class(data1) %in% "list")){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE data1 ARGUMENT MUST BE A LIST\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**if( ! any(class(data2) %in% "list")){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE data2 ARGUMENT MUST BE A LIST\n\n================\n\n")**

**stop(tempo.cat)**

**}**

*# end argument checking*

**same.length <- NULL**

**length <- NULL**

**same.name <- NULL**

**name <- NULL**

**any.id.name <- NULL**

**same.name.pos1 <- NULL**

**same.name.pos2 <- NULL**

**any.id.compartment <- NULL**

**same.compartment.pos1 <- NULL**

**same.compartment.pos2 <- NULL**

**identical.object <- NULL**

**identical.content <- NULL**

**if(identical(data1, data2)){**

**same.length <- TRUE**

**length <- length(data1)**

**if( ! is.null(names(data1))){**

**same.name <- TRUE**

**name <- names(data1)**

**any.id.name <- TRUE**

**same.name.pos1 <- 1:length(data1)**

**same.name.pos2 <- 1:length(data2)**

**}**

**any.id.compartment <- TRUE**

**same.compartment.pos1 <- 1:length(data1)**

**same.compartment.pos2 <- 1:length(data2)**

**identical.object <- TRUE**

**identical.content <- TRUE**

**}else{**

**identical.object <- FALSE**

**if( ! identical(length(data1), length(data2))){**

**same.length<- FALSE**

**}else{**

**same.length<- TRUE**

**length <- length(data1)**

**}**

**if( ! (is.null(names(data1)) & is.null(names(data2)))){**

**if( ! identical(names(data1), names(data2))){**

**same.name <- FALSE**

**}else{**

**same.name <- TRUE**

**name <- names(data1)**

**}**

**any.id.name <- FALSE**

**if(any(names(data1) %in% names(data2))){**

**any.id.name <- TRUE**

**same.name.pos1 <- which(names(data1) %in% names(data2))**

**}**

**if(any(names(data2) %in% names(data1))){**

**any.id.name <- TRUE**

**same.name.pos2 <- which(names(data2) %in% names(data1))**

**}**

**}**

**names(data1) <- NULL**

**names(data2) <- NULL**

**any.id.compartment <- FALSE**

**if(any(data1 %in% data2)){**

**any.id.compartment <- TRUE**

**same.compartment.pos1 <- which(data1 %in% data2)**

**}**

**if(any(data2 %in% data1)){**

**any.id.compartment <- TRUE**

**same.compartment.pos2 <- which(data2 %in% data1)**

**}**

**if(same.length == TRUE & ! all(is.null(same.compartment.pos1), is.null(same.compartment.pos2))){**

**if(identical(same.compartment.pos1, same.compartment.pos2)){**

**identical.content <- TRUE**

**}else{**

**identical.content <- FALSE**

**}**

**}else{**

**identical.content <- FALSE**

**}**

**}**

**output <- list(same.length = same.length, length = length, same.name = same.name, name = name, any.id.name = any.id.name, same.name.pos1 = same.name.pos1, same.name.pos2 = same.name.pos2, any.id.compartment = any.id.compartment, same.compartment.pos1 = same.compartment.pos1, same.compartment.pos2 = same.compartment.pos2, identical.object = identical.object, identical.content = identical.content)**

**return(output)**

**}**

################ Object modification

######## fun\_dataframe\_flipping() #### flipping a data frame to have column name as a qualitative column and vice-versa

**# Check OK: clear to go Apollo**

**fun\_dataframe\_flipping <- function(data, quanti.col.name = "quanti", quali.col.name = "quali"){**

*# AIM:*

*# if the data frame is made of numeric columns, a new data frame is created, with the 1st column gathering all the numeric values, and the 2nd column being the name of the columns of the initial data frame. If the data frame is made of one numeric column and one character or factor column, a new data frame is created, with the new columns corresponding to the split numeric values (according to the character column). NA are added a the end of each column to have the same number of rows*

*# REQUIRED FUNCTIONS*

*# fun\_param\_check()*

*# ARGUMENTS*

*# data: data frame to convert*

*# quanti.col.name: optional name for the quanti column of the new data frame*

*# quali.col.name: optional name for the quali column of the new data frame*

*# RETURN*

*# the modified data frame*

*# EXAMPLES*

*# obs <- data.frame(a = 1:3, b = c("A", "B", "A")) ; fun\_dataframe\_flipping(obs)*

*# obs <- data.frame(a = 1:3, b = c("A", "B", "A")) ; fun\_dataframe\_flipping(obs, quanti.col.name = "quanti", quali.col.name = "quali")*

*# obs <- data.frame(a = 1:3, b = 4:6) ; fun\_dataframe\_flipping(obs)*

*# DEBUGGING*

*# data = data.frame(a = 1:3, b = 4:6) ; quanti.col.name = "quanti" ; quali.col.name = "quali" # for function debugging*

*# data = data.frame(a = 1:3, b = 4:6, c = 11:13) ; quanti.col.name = "quanti" ; quali.col.name = "quali" # for function debugging*

*# data = data.frame(a = 1:3, b = letters[1:3]) ; quanti.col.name = "quanti" ; quali.col.name = "quali" # for function debugging*

*# data = data.frame(a = 1:3, b = letters[1:3]) ; quanti.col.name = "TEST" ; quali.col.name = "quali" # for function debugging*

*# data = data.frame(b = letters[1:3], a = 1:3) ; quanti.col.name = "quanti" ; quali.col.name = "quali" # for function debugging*

*# data = data.frame(b = c("e", "e", "h"), a = 1:3) ; quanti.col.name = "quanti" ; quali.col.name = "quali" # for function debugging*

*# argument checking*

**arg.check <- NULL** *# for function debbuging*

**checked.arg.names <- NULL** *# for function debbuging*

**ee <- expression(arg.check <- c(arg.check, tempo$problem) , checked.arg.names <- c(checked.arg.names, tempo$param.name))**

**tempo <- fun\_param\_check(data = quanti.col.name, class = "character", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = quali.col.name, class = "character", length = 1) ; eval(ee)**

**if(any(arg.check) == TRUE){**

**stop()**

**}**

*# source("C:/Users/Gael/Documents/Sources/debugging\_tools\_for\_r\_dev-v1.2/r\_debugging\_tools-v1.2.R") ; eval(parse(text = str\_basic\_arg\_check\_dev)) ; eval(parse(text = str\_arg\_check\_with\_fun\_param\_check\_dev)) # to check arguments status and if they have been checked using fun\_param\_check()*

**if( ! any(class(data) %in% "data.frame")){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE data ARGUMENT MUST BE A DATA FRAME\n\n================\n\n")**

**stop(tempo.cat)**

**}**

*# end argument checking*

**tempo.factor <- unlist(lapply(data, class))**

**for(i in 1:length(tempo.factor)){** *# convert factor columns as character*

**if(all(tempo.factor[i] == "factor")){**

**data[, i] <- as.character(data[, i])**

**}**

**}**

**tempo.factor <- unlist(lapply(data, mode))**

**if(length(data) == 2){**

**if( ! ((mode(data[, 1]) == "character" & mode(data[, 2]) == "numeric") | mode(data[, 2]) == "character" & mode(data[, 1]) == "numeric" | mode(data[, 2]) == "numeric" & mode(data[, 1]) == "numeric") ){**

**tempo.cat <- paste0("\n\n================\n\nERROR: IF data ARGUMENT IS A DATA FRAME MADE OF 2 COLUMNS, EITHER A COLUMN MUST BE NUMERIC AND THE OTHER CHARACTER, OR THE TWO COLUMNS MUST BE NUMERIC\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**if((mode(data[, 1]) == "character" | mode(data[, 2]) == "character") & (quanti.col.name != "quanti" | quali.col.name != "quali")){**

**tempo.cat <- paste0("\n\n================\n\nERROR: IMPROPER quanti.col.name OR quali.col.name RESETTINGS. THESE ARGUMENTS ARE RESERVED FOR DATA FRAMES MADE OF n NUMERIC COLUMNS ONLY\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}else{**

**if( ! all(tempo.factor %in% "numeric")){**

**tempo.cat <- paste0("\n\n================\n\nERROR: IF data ARGUMENT IS A DATA FRAME MADE OF ONE COLUMN, OR MORE THAN 2 COLUMNS, THESE COLUMNS MUST BE NUMERIC\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}**

**if(( ! any(tempo.factor %in% "character")) & is.null(names(data))){**

**tempo.cat <- paste0("\n\n================\n\nERROR: NUMERIC DATA FRAME in the data ARGUMENT MUST HAVE COLUMN NAMES\n\n================\n\n")**

**stop()**

**}**

**if(all(tempo.factor %in% "numeric")){**

**quanti <- NULL**

**for(i in 1:length(data)){**

**quanti <-c(quanti, data[, i])**

**}**

**quali <- rep(names(data), each = nrow(data))**

**output.data <- data.frame(quanti, quali)**

**names(output.data) <- c(quanti.col.name, quali.col.name)**

**}else{**

**if(class(data[, 1]) == "character"){**

**data <- cbind(data[2], data[1])**

**}**

**nc.max <- max(table(data[, 2]))** *# effectif maximum des classes*

**nb.na <- nc.max - table(data[,2])** *# nombre de NA à ajouter pour réaliser la data frame*

**tempo<-split(data[, 1], data[, 2])**

**for(i in 1:length(tempo)){tempo[[i]] <- append(tempo[[i]], rep(NA, nb.na[i]))}** *# des NA doivent être ajoutés lorsque les effectifs sont différents entre les classes. C'est uniquement pour que chaque colonne ait le même nombre de lignes*

**output.data<-data.frame(tempo)**

**}**

**return(output.data)**

**}**

######## fun\_refactorization() #### remove classes that are not anymore present in factors or factor columns in data frames

**# Check OK: clear to go Apollo**

**fun\_refactorization <- function(data, also.ordered = TRUE){**

*# AIM:*

*# refactorize a factor or the factor columns of a data frame, such as only the class present are in the levels (no empty levels). The class order in levels is kept*

*# useful to remove the empty classes after row removing for instance*

*# REQUIRED FUNCTIONS*

*# fun\_param\_check()*

*# ARGUMENTS*

*# data: factor (ordered or not) or data frame*

*# also.ordered: refactorize also ordered factors? This to deal with ordered factors that have class "ordered" "factor"*

*# RETURN*

*# a list containing:*

*# $data: the modified object*

*# $removed: the removed classes for a factor and a list of the removed classes for each factor class of the data frame*

*# EXAMPLES*

*# obs <- data.frame(a = LETTERS[1:6], b = paste0(letters[1.6], c(1,1,2,2,3,3)), c = ordered(LETTERS[7:12]), d = 1:6)[-c(1:2),] ; fun\_refactorization(obs, FALSE)*

*# obs <- data.frame(a = LETTERS[1:6], b = paste0(letters[1.6], c(1,1,2,2,3,3)), c = ordered(LETTERS[7:12]), d = 1:6)[-c(1:2),] ; fun\_refactorization(obs, TRUE)*

*# obs <- factor(LETTERS[1:6])[-c(1:2)] ; fun\_refactorization(obs, TRUE)*

*# obs <- ordered(LETTERS[1:6])[-c(1:2)] ; fun\_refactorization(obs, TRUE)*

*# obs <- factor(LETTERS[1:6], levels = rev(LETTERS[1:6]))[-c(1:2)] ; fun\_refactorization(obs, FALSE)*

*# DEBUGGING*

*# data <- data.frame(a = LETTERS[1:6], b = paste0(letters[1.6], c(1,1,2,2,3,3)), c = ordered(LETTERS[7:12]), d = 1:6) ; data <- data[-c(1:2),] ; also.ordered <- TRUE # for function debugging*

*# data <- factor(LETTERS[1:6])[-c(1:2)] ; also.ordered <- TRUE # for function debugging*

*# data <- ordered(LETTERS[1:6])[-c(1:2)] ; also.ordered <- TRUE # for function debugging*

*# argument checking*

**arg.check <- NULL** *# for function debbuging*

**checked.arg.names <- NULL** *# for function debbuging*

**ee <- expression(arg.check <- c(arg.check, tempo$problem) , checked.arg.names <- c(checked.arg.names, tempo$param.name))**

**tempo <- fun\_param\_check(data = also.ordered, class = "logical", length = 1) ; eval(ee)**

**if(any(arg.check) == TRUE){**

**stop()**

**}**

*# source("C:/Users/Gael/Documents/Sources/debugging\_tools\_for\_r\_dev-v1.2/r\_debugging\_tools-v1.2.R") ; eval(parse(text = str\_basic\_arg\_check\_dev)) ; eval(parse(text = str\_arg\_check\_with\_fun\_param\_check\_dev)) # to check arguments status and if they have been checked using fun\_param\_check()*

**if(also.ordered == FALSE){**

**if( ! (all(class(data) == "data.frame") | all(class(data) == "factor"))){**

**tempo.cat <- paste0("\n\n================\n\nERROR: data ARGUMENT MUST BE A FACTOR (NON ORDERED BECAUSE THE also.ordered ARGUMENT IS SET TO FALSE) OR A DATA FRAME\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}**

**if(also.ordered == TRUE){**

**if( ! (all(class(data) == "data.frame") | all(class(data) == "factor") | all(class(data) %in% c("ordered", "factor")))){**

**tempo.cat <- paste0("\n\n================\n\nERROR: data ARGUMENT MUST BE A FACTOR OR A DATA FRAME\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}**

*# end argument checking*

**text <- NULL**

**if(all(class(data) == "factor")){**

**tempo.keep.log <- levels(data) %in% unique(data)**

**if(any(tempo.factor.col == TRUE)){**

**text <- levels(data)[ ! tempo.keep.log]**

**data <- factor(data, levels = levels(data)[tempo.keep.log])**

**}**

**}else if(all(class(data) %in% c("ordered", "factor"))){**

**tempo.keep.log <- levels(data) %in% unique(data)**

**if(any(tempo.factor.col == TRUE)){**

**text <- levels(data)[ ! tempo.keep.log]**

**data <- ordered(data, levels = levels(data)[tempo.keep.log])**

**}**

**}else if(all(class(data) == "data.frame")){**

**text <- vector("list", length(data))**

**names(text) <- names(data)**

**tempo.factor.col <- sapply(sapply(lapply(data, class), FUN = "%in%", "factor"), FUN = "all")** *# get the factor column (logical)*

**for(i in 1:length(tempo.factor.col)){**

**if(tempo.factor.col[i] == TRUE){**

**tempo.keep.log <- levels(data[[i]]) %in% unique(data[[i]])**

**text[[i]] <- levels(data[[i]])[ ! tempo.keep.log]**

**data[[i]] <- factor(data[[i]], levels = levels(data[[i]])[tempo.keep.log])**

**}**

**}**

**if(also.ordered == TRUE){**

**tempo.ordered.col <- sapply(sapply(lapply(data, class), FUN = "%in%", "ordered"), FUN = "any")** *# get the ordered factor column (logical)*

**for(i in 1:length(tempo.ordered.col)){**

**if(tempo.ordered.col[i] == TRUE){**

**tempo.keep.log <- levels(data[[i]]) %in% unique(data[[i]])**

**text[[i]] <- levels(data[[i]])[ ! tempo.keep.log]**

**data[[i]] <- ordered(data[[i]], levels = levels(data[[i]])[tempo.keep.log])**

**}**

**}**

**}**

**}**

**output <- list(data = data, removed = text)**

**return(output)**

**}**

######## fun\_rounding() #### Rounding number if decimal present

**# Check OK: clear to go Apollo**

**fun\_rounding <- function(data, dec.nb = 2, after.lead.zero = TRUE){**

*# AIM:*

*# round a vector of value, if decimal, with the desired number of decimal digits after the decimal leading zeros*

*# BEWARE*

*# Work well with numbers as character strings, but not always with numerical numbers because of the floating point*

*# Numeric values are really truncated from a part of their decimal digits, whatever options(digits) settings*

*# ARGUMENTS*

*# data: a vector of numbers (numeric or character mode)*

*# dec.nb: number of required decimal digits*

*# after.lead.zero: logical. If FALSE, rounding is performed for all the decimal numbers, whatever the leading zeros (e.g., 0.123 -> 0.12 and 0.00128 -> 0.00). If TRUE, dec.nb are taken after the leading zeros (e.g., 0.123 -> 0.12 and 0.00128 -> 0.0013)*

*# REQUIRED FUNCTIONS*

*# fun\_param\_check()*

*# RETURN*

*# the modified vector*

*# EXAMPLES*

*# fun\_rounding(data = c(10, 100.001, 333.0001254, 12312.1235), dec.nb = 2, after.lead.zero = FALSE)*

*# fun\_rounding(data = c("10", "100.001", "333.0001254", "12312.1235"), dec.nb = 2, after.lead.zero = FALSE)*

*# DEBUGGING*

*# data = data = c(10, 100.001, 333.0001254, 12312.1235) ; dec.nb = 2 ; after.lead.zero = FALSE # # for function debugging*

*# data = data = c("10", "100.001", "333.0001254", "12312.1235") ; dec.nb = 2 ; after.lead.zero = TRUE # # for function debugging*

*# argument checking*

**if( ! (all(typeof(data) == "character") | all(typeof(data) == "double") | all(typeof(data) == "integer"))){**

**tempo.cat <- paste0("\n\n================\n\nERROR: data ARGUMENT MUST BE A VECTOR OF NUMBERS (IN NUMERIC OR CHARACTER MODE)\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**arg.check <- NULL** *# for function debbuging*

**checked.arg.names <- NULL** *# for function debbuging*

**ee <- expression(arg.check <- c(arg.check, tempo$problem) , checked.arg.names <- c(checked.arg.names, tempo$param.name))**

**tempo <- fun\_param\_check(data = dec.nb, typeof = "integer", length = 1, double.as.integer.allowed = TRUE, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = after.lead.zero, class = "logical", length = 1) ; eval(ee)**

**if(any(arg.check) == TRUE){**

**stop()**

**}**

*# source("C:/Users/Gael/Documents/Sources/debugging\_tools\_for\_r\_dev-v1.2/r\_debugging\_tools-v1.2.R") ; eval(parse(text = str\_basic\_arg\_check\_dev)) ; eval(parse(text = str\_arg\_check\_with\_fun\_param\_check\_dev)) # to check arguments status and if they have been checked using fun\_param\_check()*

*# end argument checking*

**tempo <- grepl(x = data, pattern = "\\.")** *# detection of decimal numbers*

**ini.mode <- mode(data)**

**data <- as.character(data)** *# to really truncate decimal digits*

**for(i in 1:length(data)){** *# scan all the numbers of the vector*

**if(tempo[i] == TRUE){** *# means decimal number*

**if(after.lead.zero == TRUE){**

**zero.pos <- unlist(gregexpr(text=data[i], pattern = 0))** *# recover all the position of the zeros in the number. -1 if no zeros (do not record the leading and trailing zeros)*

**}else{**

**zero.pos <- -1** *# -1 as if no zero*

**}**

**dot.pos <- unlist(gregexpr(text=data[i], pattern = "\\."))** *# recover all the position of the zeros in the number*

**digit.pos <- unlist(gregexpr(text=data[i], pattern = "[[:digit:]]"))** *# recover all the position of the digits in the number*

**dec.pos <- digit.pos[digit.pos > dot.pos]**

**count <- 0**

**while((dot.pos + count + 1) %in% zero.pos & (dot.pos + count + 1) <= max(dec.pos) & (count + dec.nb) < length(dec.pos)){** *# count the number of leading zeros in the decimal part*

**count <- count + 1**

**}**

**data[i] <- formatC(as.numeric(data[i]), digits = (count + dec.nb), format = "f")**

**}**

**}**

**if(ini.mode != "character"){**

**data <- as.numeric(data)**

**}**

**return(data)**

**}**

######## fun\_90clock\_matrix\_rot() #### 90° clockwise matrix rotation

**# Check OK: clear to go Apollo**

**fun\_90clock\_matrix\_rot <- function(data){**

*# AIM:*

*# 90° clockwise matrix rotation*

*# applied twice, the function provide the mirror matrix, according to vertical and horizontal symmetry*

*# REQUIRED FUNCTIONS*

*# fun\_param\_check()*

*# ARGUMENTS*

*# data: matrix (matrix class)*

*# RETURN*

*# the modified matrix*

*# EXAMPLES*

*# obs <- matrix(1:10, ncol = 1) ; obs ; fun\_90clock\_matrix\_rot(obs)*

*# obs <- matrix(LETTERS[1:10], ncol = 5) ; obs ; fun\_90clock\_matrix\_rot(obs)*

*# DEBUGGING*

*# data = matrix(1:10, ncol = 1)*

*# argument checking*

*# source("C:/Users/Gael/Documents/Sources/debugging\_tools\_for\_r\_dev-v1.2/r\_debugging\_tools-v1.2.R") ; eval(parse(text = str\_basic\_arg\_check\_dev)) # to check arguments status and if they have been checked using fun\_param\_check()*

**if( ! any(class(data) %in% "matrix")){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE data ARGUMENT MUST BE A MATRIX\n\n================\n\n")**

**stop(tempo.cat)**

**}**

*# end argument checking*

**for (i in 1:ncol(data)){data[,i] <- rev(data[,i])}**

**data <- t(data)**

**return(data)**

**}**

######## fun\_hexa\_hsv\_color\_matrix() #### Conversion of a numeric matrix into hexadecimal color matrix

**# Check OK: clear to go Apollo**

**fun\_hexa\_hsv\_color\_matrix <- function(mat1, mat.hsv.h = TRUE, notch = 1, s = 1, v = 1, forced.color = NULL){**

*# AIM:*

*# convert a matrix made of number into a hexadecimal matrix for rgb colorization*

*# REQUIRED FUNCTIONS*

*# fun\_param\_check()*

*# ARGUMENTS:*

*# mat1: matrix 1 of non negative numerical values that has to be colored (matrix class). NA allowed*

*# mat.hsv.h: logical. Is mat1 the h of hsv colors ? (if TRUE, mat1 must be between zero and 1)*

*# notch: single value between 0 and 1 to shift the successive colors on the hsv circle by + notch*

*# s: s argument of hsv(). Must be between 0 and 1*

*# v: v argument of hsv(). Must be between 0 and 1*

*# forced.color: Must be NULL or hexadecimal color code or name given by colors(). The first minimal values of mat1 will be these colors. All the color of mat1 can be forced using this argument*

*# RETURN*

*# a list containing:*

*# $mat1.name: name of mat1*

*# $colored.mat: colors of mat1 in hexa*

*# $problem: logical. Is any colors of forced.color overlap the colors designed by the function. NULL if forced.color = NULL*

*# $text.problem: text when overlapping colors. NULL if forced.color = NULL or problem == FALSE*

*# EXAMPLES*

*# mat1 = matrix(c(1,1,1,2,1,5,9,NA), ncol = 2) ; dimnames(mat1) <- list(LETTERS[1:4], letters[1:2]) ; fun\_hexa\_hsv\_color\_matrix(mat1, mat.hsv.h = FALSE, notch = 1, s = 1, v = 1, forced.color = NULL)*

*# DEBUGGING*

*# mat1 = matrix(c(1,1,1,2,1,5,9,NA), ncol = 2) ; dimnames(mat1) <- list(LETTERS[1:4], letters[1:2]); mat.hsv.h = FALSE ; notch = 1 ; s = 1 ; v = 1 ; forced.color = c(hsv(1,1,1), hsv(0,0,0)) # for function debugging*

*# argument checking*

**arg.check <- NULL** *# for function debbuging*

**checked.arg.names <- NULL** *# for function debbuging*

**ee <- expression(arg.check <- c(arg.check, tempo$problem) , checked.arg.names <- c(checked.arg.names, tempo$param.name))**

**tempo <- fun\_param\_check(data = mat1, mode = "numeric", class = "matrix", na.contain = TRUE, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = mat.hsv.h, class = "logical", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = notch, mode = "numeric", length = 1, prop = TRUE) ; eval(ee)**

**tempo <- fun\_param\_check(data = s, mode = "numeric", length = 1, prop = TRUE) ; eval(ee)**

**tempo <- fun\_param\_check(data = v, mode = "numeric", length = 1, prop = TRUE) ; eval(ee)**

**if(any(arg.check) == TRUE){**

**stop()**

**}**

*# source("C:/Users/Gael/Documents/Sources/debugging\_tools\_for\_r\_dev-v1.2/r\_debugging\_tools-v1.2.R") ; eval(parse(text = str\_basic\_arg\_check\_dev)) ; eval(parse(text = str\_arg\_check\_with\_fun\_param\_check\_dev)) # to check arguments status and if they have been checked using fun\_param\_check()*

**if(mat.hsv.h == TRUE & fun\_param\_check(data = mat1, mode = "numeric", prop = TRUE, print = FALSE)$problem == TRUE){**

**tempo.cat <- paste0("\n\n================\n\nERROR: mat1 ARGUMENT MUST BE A MATRIX OF PROPORTIONS SINCE THE mat.hsv.h ARGUMENT IS SET TO TRUE\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**if( ! is.null(forced.color)){**

**tempo <- fun\_param\_check(data = forced.color, class = "character")**

**if(tempo$problem == TRUE){**

**stop()**

**}**

**if( ! all(forced.color %in% colors() | grepl(pattern = "^#", forced.color))){** *# check that all strings of forced.color start by #*

**tempo.cat <- paste0("\n\n================\n\nERROR: forced.color ARGUMENT MUST BE A HEXADECIMAL COLOR VECTOR STARTING BY # AND/OR COLOR NAMES GIVEN BY colors()\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}**

*# end argument checking*

**problem <- NULL**

**text.problem <- NULL**

**mat1.name <- deparse(substitute(mat1))**

*# change the scale of the plotted matrix*

**if(mat.hsv.h == TRUE){**

**if(any(min(mat1, na.rm = TRUE) < 0 | max(mat1, na.rm = TRUE) > 1, na.rm = TRUE)){**

**tempo.cat <- paste0("\n\n================\n\nERROR: mat1 MUST BE MADE OF VALUES BETWEEN 0 AND 1 BECAUSE mat.hsv.h ARGUMENT SET TO TRUE\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}else{**

**if(any(mat1 - floor(mat1) > 0, na.rm = TRUE) | any(mat1 == 0, na.rm = TRUE)){**

**tempo.cat <- paste0("\n\n================\n\nERROR: mat1 MUST BE MADE OF INTEGER VALUES WITHOUT 0 BECAUSE mat.hsv.h ARGUMENT SET TO FALSE\n\n================\n\n")**

**stop(tempo.cat)**

**}else{**

**mat1 <- mat1 / max(mat1, na.rm = TRUE)**

**}**

**}**

**if(notch != 1){**

**different.color <- unique(as.vector(mat1))**

**different.color <- different.color[ ! is.na(different.color)]**

**tempo.different.color <- different.color + c(0, cumsum(rep(notch, length(different.color) - 1)))**

**tempo.different.color <- tempo.different.color - floor(tempo.different.color)**

**if(any(duplicated(tempo.different.color) == TRUE)){**

**tempo.cat <- paste0("\n\n================\n\nERROR: DUPLICATED VALUES AFTER USING notch (", paste(tempo.different.color[duplicated(tempo.different.color)], collapse = " "), "). TRY ANOTHER notch VALUE\n\n================\n\n")**

**stop(tempo.cat)**

**}else if(length(different.color) != length(tempo.different.color)){**

**tempo.cat <- paste0("\n\n================\n\nERROR: LENGTH OF different.color (", paste(different.color, collapse = " "), ") DIFFERENT FROM LENGTH OF tempo.different.color (", paste(tempo.different.color, collapse = " "), ")\n\n================\n\n")**

**stop(tempo.cat)**

**}else{**

**for(i in 1:length(different.color)){**

**mat1[mat1 == different.color[i]] <- tempo.different.color[i]**

**}**

**}**

**}**

**if( ! is.null(forced.color)){**

**hexa.values.to.change <- hsv(unique(sort(mat1))[1:length(forced.color)], s, v)**

**}**

**mat1[ ! is.na(mat1)] <- hsv(mat1[ ! is.na(mat1)], s, v)**

**if( ! is.null(forced.color)){**

**if(any(forced.color %in% mat1, na.rm = TRUE)){**

**problem <- TRUE**

**text.problem <- paste0("THE FOLLOWING COLORS WHERE INTRODUCED USING forced.color BUT WHERE ALREADY PRESENT IN THE COLORED MATRIX :", paste(forced.color[forced.color %in% mat1], collapse = " "))**

**}else{**

**problem <- FALSE**

**}**

**for(i in 1:length(hexa.values.to.change)){**

**if( ! any(mat1 == hexa.values.to.change[i], na.rm = TRUE)){**

**tempo.cat <- paste0("\n\n================\n\nERROR: THE ", hexa.values.to.change[i], " VALUE FROM hexa.values.to.change IS NOT REPRESENTED IN mat1 : ", paste(unique(as.vector(mat1)), collapse = " "), "\n\n================\n\n")**

**stop(tempo.cat)**

**}else{**

**mat1[which(mat1 == hexa.values.to.change[i])] <- forced.color[i]**

**}**

**}**

**}**

**output <- list(mat1.name = mat1.name, colored.mat = mat1, problem = problem, text.problem = text.problem)**

**return(output)**

**}**

################ Graphics

*# this order can be used:*

*# fun\_window\_width\_resizing()*

*# fun\_open\_window()*

*# fun\_graph\_param\_prior\_plot() # not for ggplot2*

*# plot() or any other plotting*

*# fun\_feature\_post\_plot() if fun\_graph\_param\_prior\_plot() has been used # not for ggplot2*

*# fun\_close\_specif\_window()*

######## fun\_window\_width\_resizing() #### window width depending on classes to plot

**# Check OK: clear to go Apollo**

**fun\_window\_width\_resizing <- function(class.nb, inches.per.class.nb = 1, ini.window.width = 7, inch.left.space, inch.right.space, boundarie.space = 0.5){**

*# AIM:*

*# rescale the width of a window to open depending on the number of classes to plot*

*# can be used for height, considering that it is as if it was a width*

*# this order can be used:*

*# fun\_window\_width\_resizing()*

*# fun\_open\_window()*

*# fun\_graph\_param\_prior\_plot() # not for ggplot2*

*# plot() or any other plotting*

*# fun\_feature\_post\_plot() if fun\_graph\_param\_prior\_plot() has been used # not for ggplot2*

*# fun\_close\_specif\_window()*

*# REQUIRED FUNCTIONS*

*# fun\_param\_check()*

*# ARGUMENTS*

*# class.nb: number of class to plot*

*# inches.per.class.nb: number of inches per unit of class.nb. 2 means 2 inches for each boxplot for instance*

*# ini.window.width:initial window width in inches*

*# inch.left.space: left horizontal margin of the figure region (in inches)*

*# inch.right.space: right horizontal margin of the figure region (in inches)*

*# boundarie.space: space between the right and left limits of the plotting region and the plot (0.5 means half a class width)*

*# RETURN*

*# the new window width in inches*

*# EXAMPLES*

*# fun\_window\_width\_resizing(class.nb = 10, inches.per.class.nb = 0.2, ini.window.width = 7, inch.left.space = 1, inch.right.space = 1, boundarie.space = 0.5)*

*# DEBUGGING*

*# class.nb = 10 ; inches.per.class.nb = 0.2 ; ini.window.width = 7 ; inch.left.space = 1 ; inch.right.space = 1 ; boundarie.space = 0.5 # for function debugging*

*# argument checking*

**arg.check <- NULL** *# for function debbuging*

**checked.arg.names <- NULL** *# for function debbuging*

**ee <- expression(arg.check <- c(arg.check, tempo$problem) , checked.arg.names <- c(checked.arg.names, tempo$param.name))**

**tempo <- fun\_param\_check(data = class.nb, typeof = "integer", length = 1, double.as.integer.allowed = TRUE, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = inches.per.class.nb, mode = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = ini.window.width, mode = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = inch.left.space, mode = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = inch.right.space, mode = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = boundarie.space, mode = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**if(any(arg.check) == TRUE){**

**stop()**

**}**

*# source("C:/Users/Gael/Documents/Sources/debugging\_tools\_for\_r\_dev-v1.2/r\_debugging\_tools-v1.2.R") ; eval(parse(text = str\_basic\_arg\_check\_dev)) ; eval(parse(text = str\_arg\_check\_with\_fun\_param\_check\_dev)) # to check arguments status and if they have been checked using fun\_param\_check()*

*# end argument checking*

**range.max <- class.nb + boundarie.space** *# the max range of the future plot*

**range.min <- boundarie.space** *# the min range of the future plot*

**window.width <- inch.left.space + inch.right.space + inches.per.class.nb \* (range.max - range.min)**

**return(window.width)**

**}**

######## fun\_open\_window() #### Open a GUI or pdf graphic window

**# Check OK: clear to go Apollo**

**fun\_open\_window <- function(pdf.disp = TRUE, path.fun = "working.dir", pdf.name.file = "graph", width.fun = 7, height.fun = 7, paper = "special", no.pdf.overwrite = TRUE, return.output = FALSE){**

*# AIM:*

*# open a pdf or screen (GUI) graphic window*

*# BEWARE: on Linux, use pdf.disp = TRUE, if (GUI) graphic window is not always available, meaning that X is not installed (clusters for instance). Use X11() in R to test if available*

*# this order can be used:*

*# fun\_window\_width\_resizing()*

*# fun\_open\_window()*

*# fun\_graph\_param\_prior\_plot() # not for ggplot2*

*# plot() or any other plotting*

*# fun\_feature\_post\_plot() if fun\_graph\_param\_prior\_plot() has been used # not for ggplot2*

*# fun\_close\_specif\_window()*

*# REQUIRED FUNCTIONS*

*# fun\_param\_check()*

*# ARGUMENTS:*

*# pdf.disp: use pdf or not*

*# path.fun: where the pdf is saved. Must not finish by a path separator. Write "working.dir" if working directory is required (default)*

*# pdf.name.file: name of the pdf file containing the graphs (the .pdf extension is added by the function)*

*# width.fun: width of the windows (in inches)*

*# height.fun: height of the windows (in inches)*

*# paper: paper argument of the pdf function (paper format). Only used for pdf(). Either "a4", "letter", "legal", "us", "executive", "a4r", "USr" or "special". If "special", means that width.fun and height.fun specify the paper size*

*# no.pdf.overwrite: existing pdf can be overwritten ? Only used if pdf.disp = TRUE*

*# return.output: return output ? If TRUE but function not assigned, the output list is displayed*

*# RETURN*

*# a list containing:*

*# $pdf.loc: path of the pdf created*

*# $ini.par: initial par() parameters (to reset in a new graph)*

*# $zone.ini: initial window spliting (to reset in a new graph)*

*# EXAMPLES*

*# fun\_open\_window(pdf.disp = FALSE, path.fun = "C:/Users/Gael/Desktop", pdf.name.file = "graph", width.fun = 7, height.fun = 7, paper = "special", no.pdf.overwrite = TRUE, return.output = TRUE)*

*# DEBUGGING*

*# pdf.disp = TRUE ; path.fun = "C:/Users/Gael/Desktop" ; pdf.name.file = "graphs" ; width.fun = 7 ; height.fun = 7 ; paper = "special" ; no.pdf.overwrite = TRUE ; return.output = TRUE # for function debugging*

*# argument checking*

**arg.check <- NULL** *# for function debbuging*

**checked.arg.names <- NULL** *# for function debbuging*

**ee <- expression(arg.check <- c(arg.check, tempo$problem) , checked.arg.names <- c(checked.arg.names, tempo$param.name))**

**tempo <- fun\_param\_check(data = pdf.disp, class = "logical", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = path.fun, class = "character", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = pdf.name.file, class = "character", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = width.fun, mode = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = height.fun, mode = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = path.fun, class = "character", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = paper, options = c("a4", "letter", "legal", "us", "executive", "a4r", "USr", "special", "A4", "LETTER", "LEGAL", "US"), length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data =no.pdf.overwrite, class = "logical", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = return.output, class = "logical", length = 1) ; eval(ee)**

**if(any(arg.check) == TRUE){**

**stop()**

**}**

*# source("C:/Users/Gael/Documents/Sources/debugging\_tools\_for\_r\_dev-v1.2/r\_debugging\_tools-v1.2.R") ; eval(parse(text = str\_basic\_arg\_check\_dev)) ; eval(parse(text = str\_arg\_check\_with\_fun\_param\_check\_dev)) # to check arguments status and if they have been checked using fun\_param\_check()*

*# end argument checking*

**if(path.fun == "working.dir"){**

**path.fun <- getwd()**

**}else{**

**if(dir.exists(path.fun) == FALSE){**

**tempo.cat <- paste0("\n\n================\n\nERROR: path.fun ARGUMENT DOES NOT CORRESPOND TO EXISTING DIRECTORY\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}**

**if(Sys.info()["sysname"] == "Windows"){** *# Note that .Platform$OS.type() only says "unix" for macOS and Linux and "Windows" for Windows*

**tempo <- NULL**

**windows()**

**ini.par <- par(no.readonly = TRUE)** *# to recover the initial graphical parameters if required (reset)*

**invisible(dev.off())** *# close the new window*

**}else if(Sys.info()["sysname"] == "Linux"){**

**if( ! file.exists(paste0(getwd(), "/Rplots.pdf"))){**

**tempo <- suppressWarnings(try(X11(), silent = TRUE))[]** *# open a X11 window or a pdf. So no need to use again X11(). tempo == NULL if no problem, meaning that the X11 window is opened. If tempo != NULL, a pdf is open here paste0(getwd(), "/Rplots.pdf")*

**ini.par <- par(no.readonly = TRUE)** *# recover the initial graphical parameters. Works even if X11 is not working as R opens a pdf*

**invisible(dev.off())** *# can be used here to close the pdf windows if tempo != NULL and to close the X11 window if tempo == NULL*

**}else{**

**tempo.cat <- paste0("\n\n================\n\nPROBLEM: THIS FUNCTION CANNOT OPEN GUI ON LINUX OR NON MACOS UNIX SYSTEM (X GRAPHIC INTERFACE HAS TO BE SET.\nTO OVERCOME THIS, PLEASE SET pdf.disp ARGUMENT TO TRUE AND RERUN\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}else{**

**tempo <- NULL**

**quartz()**

**ini.par <- par(no.readonly = TRUE)** *# to recover the initial graphical parameters if required (reset)*

**invisible(dev.off())** *# close the new window*

**}**

**zone.ini <- matrix(1, ncol=1)** *# to recover the initial parameters for next figure region when device region split into several figure regions*

**if(pdf.disp == TRUE){**

**pdf.loc <- paste0(path.fun, "/", pdf.name.file, ".pdf")**

**if(file.exists(pdf.loc) == TRUE & no.pdf.overwrite == TRUE){**

**tempo.cat <- paste0("\n\n================\n\nERROR: pdf.loc FILE ALREADY EXISTS AND CANNOT BE OVERWRITTEN DUE TO no.pdf.overwrite ARGUMENT SET TO TRUE\n\n================\n\n")**

**stop(tempo.cat)**

**}else{**

**pdf(width = width.fun, height = height.fun, file=pdf.loc, paper = paper)**

**}**

**}else if(pdf.disp == FALSE){**

**pdf.loc <- NULL**

**if(Sys.info()["sysname"] == "Windows"){** *# .Platform$OS.type() only says "unix" for macOS and Linux and "Windows" for Windows*

**windows(width = width.fun, height = height.fun, rescale="fixed")**

**}else if(Sys.info()["sysname"] == "Linux"){**

**if( ! is.null(tempo)){**

**stop("PROBLEM: THIS FUNCTION CANNOT OPEN GUI ON LINUX OR NON MACOS UNIX SYSTEM (X GRAPHIC INTERFACE HAS TO BE SET.\nTO OVERCOME THIS, PLEASE SET pdf.disp ARGUMENT TO TRUE AND RERUN")**

**}else{**

**X11(width = width.fun, height = height.fun)**

**}**

**}else{**

**quartz(width = width.fun, height = height.fun)**

**}**

**}**

**if(return.output == TRUE){**

**output <- list(pdf.loc = pdf.loc, ini.par = ini.par, zone.ini = zone.ini)**

**return(output)**

**}**

**}**

######## fun\_graph\_param\_prior\_plot() #### Graph param before plotting

**# Check OK: clear to go Apollo**

**fun\_graph\_param\_prior\_plot <- function(param.reinitial = FALSE, xlog.scale = FALSE, ylog.scale = FALSE, remove.label = TRUE, remove.x.axis = TRUE, remove.y.axis = TRUE, std.x.range = TRUE, std.y.range = TRUE, down.space = 1, left.space = 1, up.space = 1, right.space = 1, orient = 1, dist.legend = 3.5, tick.length = 0.5, box.type = "n", amplif.label = 1, amplif.axis = 1, display.extend = FALSE, return.par = FALSE){**

*# AIM:*

*# reinitialize and set the graphic parameters before plotting*

*# very convenient to erase the axes for post plot axis redrawing using fun\_feature\_post\_plot()*

*# REQUIRED FUNCTIONS*

*# fun\_param\_check()*

*# ARGUMENTS*

*# param.reinitial: reinitialize graphic parameters before applying the new ones, as defined by the other arguments? Either TRUE or FALSE*

*# xlog.scale: Log scale for the x-axis? Either TRUE or FALSE. If TRUE, erases the x-axis, except legend, for further drawing by fun\_feature\_post\_plot()(xlog argument of par())*

*# ylog.scale: Log scale for the y-axis? Either TRUE or FALSE. If TRUE, erases the y-axis, except legend, for further drawing by fun\_feature\_post\_plot()(ylog argument of par())*

*# remove.label: remove labels (axis legend) of the two axes? Either TRUE or FALSE (ann argument of par())*

*# remove.x.axis: remove x-axis except legend? Either TRUE or FALSE (control the xaxt argument of par()). Automately set to TRUE if xlog.scale == TRUE*

*# remove.y.axis: remove y-axis except legend? Either TRUE or FALSE (control the yaxt argument of par()). Automately set to TRUE if ylog.scale == TRUE*

*# std.x.range: standard range on the x-axis? TRUE (no range extend) or FALSE (4% range extend). Controls xaxs argument of par() (TRUE is xaxs = "i", FALSE is xaxs = "r")*

*# std.y.range: standard range on the y-axis? TRUE (no range extend) or FALSE (4% range extend). Controls yaxs argument of par() (TRUE is yaxs = "i", FALSE is yaxs = "r")*

*# down.space: lower vertical margin (in inches, mai argument of par())*

*# left.space: left horizontal margin (in inches, mai argument of par())*

*# up.space: upper vertical margin between plot region and grapical window (in inches, mai argument of par())*

*# right.space: right horizontal margin (in inches, mai argument of par())*

*# orient: scale number orientation (las argument of par()). 0, always parallel to the axis; 1, always horizontal; 2, always perpendicular to the axis; 3, always vertical*

*# dist.legend: numeric value that moves axis legends away in inches (first number of mgp argument of par() but in inches thus / 0.2)*

*# tick.length: length of the ticks (1 means complete the distance between the plot region and the axis numbers, 0.5 means half the length, etc. 0 means no tick*

*# box.type: bty argument of par(). Either "o", "l", "7", "c", "u", "]", the resulting box resembles the corresponding upper case letter. A value of "n" suppresses the box*

*# amplif.label: increase or decrease the size of the text in legends*

*# amplif.axis: increase or decrease the size of the scale numbers in axis*

*# display.extend: extend display beyond plotting region? Either TRUE or FALSE (xpd argument of par() without NA)*

*# return.par: return graphic parameter modification?*

*# RETURN*

*# return graphic parameter modification*

*# EXAMPLES*

*# fun\_graph\_param\_prior\_plot(param.reinitial = FALSE, xlog.scale = FALSE, ylog.scale = FALSE, remove.label = TRUE, remove.x.axis = TRUE, remove.y.axis = TRUE, down.space = 1, left.space = 1, up.space = 1, right.space = 1, orient = 1, dist.legend = 4.5, tick.length = 0.5, box.type = "n", amplif.label = 1, amplif.axis = 1, display.extend = FALSE, return.par = FALSE)*

*# DEBUGGING*

*# param.reinitial = FALSE ; xlog.scale = FALSE ; ylog.scale = FALSE ; remove.label = TRUE ; remove.x.axis = TRUE ; remove.y.axis = TRUE ; down.space = 1 ; left.space = 1 ; up.space = 1 ; right.space = 1 ; orient = 1 ; dist.legend = 4.5 ; tick.length = 0.5 ; box.type = "n" ; amplif.label = 1 ; amplif.axis = 1 ; display.extend = FALSE ; return.par = FALSE # for function debugging*

*# argument checking*

**arg.check <- NULL** *# for function debbuging*

**checked.arg.names <- NULL** *# for function debbuging*

**ee <- expression(arg.check <- c(arg.check, tempo$problem) , checked.arg.names <- c(checked.arg.names, tempo$param.name))**

**tempo <- fun\_param\_check(data = param.reinitial, class = "logical", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = xlog.scale, class = "logical", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = ylog.scale, class = "logical", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = remove.label, class = "logical", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = remove.x.axis, class = "logical", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = remove.y.axis, class = "logical", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = std.x.range, class = "logical", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = std.y.range, class = "logical", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = down.space, class = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = left.space, class = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = up.space, class = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = right.space, class = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = orient, class = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = dist.legend, class = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = tick.length, class = "numeric", length = 1, prop = TRUE) ; eval(ee)**

**tempo <- fun\_param\_check(data = box.type, options = c("o", "l", "7", "c", "u", "]", "n"), length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = amplif.label, class = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = amplif.axis, class = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = display.extend, class = "logical", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = return.par, class = "logical", length = 1) ; eval(ee)**

**if(any(arg.check) == TRUE){**

**stop()**

**}**

*# source("C:/Users/Gael/Documents/Sources/debugging\_tools\_for\_r\_dev-v1.2/r\_debugging\_tools-v1.2.R") ; eval(parse(text = str\_basic\_arg\_check\_dev)) ; eval(parse(text = str\_arg\_check\_with\_fun\_param\_check\_dev)) # to check arguments status and if they have been checked using fun\_param\_check()*

*# end argument checking*

**if(param.reinitial == TRUE){**

**if(Sys.info()["sysname"] == "Windows"){** *# Note that .Platform$OS.type() only says "unix" for macOS and Linux and "Windows" for Windows*

**windows()**

**ini.par <- par(no.readonly = TRUE)** *# to recover the initial graphical parameters if required (reset)*

**invisible(dev.off())** *# close the new window*

**par(ini.par)** *# apply the initial par to current window*

**}else if(Sys.info()["sysname"] == "Linux"){**

**if( ! file.exists(paste0(getwd(), "/Rplots.pdf"))){**

**tempo <- suppressWarnings(try(X11(), silent = TRUE))[]** *# open a X11 window or a pdf. So no need to use again X11(). tempo == NULL if no problem, meaning that the X11 window is opened. If tempo != NULL, a pdf is open here paste0(getwd(), "/Rplots.pdf")*

**ini.par <- par(no.readonly = TRUE)** *# recover the initial graphical parameters. Works even if X11 is not working as R opens a pdf*

**invisible(dev.off())** *# can be used here to close the pdf windows if tempo != NULL and to close the X11 window if tempo == NULL*

**par(ini.par)** *# apply the initial par to current window*

**}else{**

**tempo.cat <- paste0("\n\n================\n\nPROBLEM: THIS FUNCTION CANNOT OPEN GUI ON LINUX OR NON MACOS UNIX SYSTEM (X GRAPHIC INTERFACE HAS TO BE SET.\nTO OVERCOME THIS, PLEASE SET pdf.disp ARGUMENT TO TRUE AND RERUN\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}else{** *# macOS*

**quartz()**

**ini.par <- par(no.readonly = TRUE)** *# to recover the initial graphical parameters if required (reset)*

**invisible(dev.off())** *# close the new window*

**par(ini.par)** *# apply the initial par to current window*

**}**

**}**

**if(remove.x.axis == TRUE){**

**par(xaxt = "n")** *# suppress the y-axis label*

**}else{**

**par(xaxt = "s")**

**}**

**if(remove.y.axis == TRUE){**

**par(yaxt = "n")** *# suppress the y-axis label*

**}else{**

**par(yaxt = "s")**

**}**

**if(std.x.range == TRUE){**

**par(xaxs = "i")**

**}else{**

**par(xaxs = "r")**

**}**

**if(std.y.range == TRUE){**

**par(yaxs = "i")**

**}else{**

**par(yaxs = "r")**

**}**

**par(mai = c(down.space, left.space, up.space, right.space), ann = ! remove.label, las = orient, mgp = c(dist.legend/0.2, 1, 0), xpd = display.extend, bty= box.type, cex.lab = amplif.label, cex.axis = amplif.axis)**

**par(tcl = -par()$mgp[2] \* tick.length)** *# tcl gives the length of the ticks as proportion of line text, knowing that mgp is in text lines. So the main ticks are a 0.5 of the distance of the axis numbers by default. The sign provides the side of the tick (negative for outside of the plot region)*

**if(xlog.scale == TRUE){**

**par(xaxt = "n", xlog = TRUE)** *# suppress the x-axis label*

**}else{**

**par(xlog = FALSE)**

**}**

**if(ylog.scale == TRUE){**

**par(yaxt = "n", ylog = TRUE)** *# suppress the y-axis label*

**}else{**

**par(ylog = FALSE)**

**}**

**if(return.par == TRUE){**

**tempo.par <- par()**

**return(tempo.par)**

**}**

**}**

######## fun\_feature\_post\_plot() #### Graph param before plotting

**# Check OK: clear to go Apollo**

**fun\_feature\_post\_plot <- function(x.side = 0, x.categ = NULL, x.categ.pos = NULL, x.lab = "", x.dist.legend = 0.5, x.log.scale = FALSE, x.nb.inter.tick = 1, y.side = 0, y.categ = NULL, y.categ.pos = NULL, y.lab = "", y.dist.legend = 0.5, y.log.scale = FALSE, y.nb.inter.tick = 1, text.angle = 90, tick.length = 0.5, sec.tick.length = 0.3, bg.color = NULL, grid.lwd = NULL, grid.col = "white", corner.text = "", magnific = 1.5, magnific.corner.text = 1, just.label.add = FALSE, par.reset = FALSE, custom.par = NULL){**

*# AIM:*

*# redesign axis. If x.side = 0, y.side = 0, the function just adds text at topright of the graph and reset par() for next graphics and provides outputs (see below)*

*# provide also positions for legend or additional text on the graph*

*# use fun\_graph\_param\_prior\_plot() before this function for initial inactivation of the axis drawings*

*# REQUIRED FUNCTIONS*

*# fun\_param\_check()*

*# fun\_open\_window() to reinitialize graph parameters if par.reset = TRUE and custom.par = NULL*

*# ARGUMENTS*

*# x.side: axis at the bottom (1) or top (3) of the region figure. Write 0 for no change*

*# x.categ: character vector representing the classes (levels()) to specify when the x-axis is qualititative(stripchart, boxplot)*

*# x.categ.pos: position of the classes names (numeric vector of identical length than x.categ). If left NULL, this will be 1:length(levels())*

*# x.lab: label of the x-axis. If x.side == 0 and x.lab != "", then x.lab is printed*

*# x.dist.legend: increase the number to move x-axis legends away in inches (first number of mgp argument of par() but in inches)*

*# x.log.scale: Log scale for the x-axis? Either TRUE or FALSE*

*# x.nb.inter.tick: number of secondary ticks between main ticks on x-axis (only if not log scale). 0 means no secondary ticks*

*# y.side: axis at the left (2) or right (4) of the region figure. Write 0 for no change*

*# y.categ: classes (levels()) to specify when the y-axis is qualititative(stripchart, boxplot)*

*# y.categ.pos: position of the classes names (numeric vector of identical length than y.categ). If left NULL, this will be 1:length(levels())*

*# y.lab: label of the y-axis. If y.side == 0 and y.lab != "", then y.lab is printed*

*# y.dist.legend: increase the number to move y-axis legends away in inches (first number of mgp argument of par() but in inches)*

*# y.log.scale: Log scale for the y-axis? Either TRUE or FALSE*

*# y.nb.inter.tick: number of secondary ticks between main ticks on y-axis (only if not log scale). 0 means non secondary ticks*

*# text.angle: angle of the text when axis is qualitative*

*# tick.length: length of the main ticks (1 means complete the distance between the plot region and the axis numbers, 0.5 means half the length, etc., 0 for no ticks)*

*# sec.tick.length: length of the secondary ticks (1 means complete the distance between the plot region and the axis numbers, 0.5 means half the length, etc., 0 for no ticks)*

*# bg.color: background color of the plot region. NULL for no color. BEWARE: cover/hide an existing plot !*

*# grid.lwd: if non NULL, activate the grid line (specify the line width)*

*# grid.col: grid line color (only if grid.lwd non NULL)*

*# corner.text: text to add at the top right corner of the window*

*# magnific: increase or decrease the value to increase or decrease the size of the axis (numbers and legends)*

*# magnific.corner.text: increase or decrease the size of the text*

*# par.reset: to reset all the graphics parameters. BEWARE: TRUE can generate display problems, mainly in graphic devices with multiple figure regions*

*# just.label.add: just add axis labels (legend)? Either TRUE or FALSE. If TRUE, at least (x.side == 0 & x.lab != "") or (y.side == 0 & y.lab != "") must be set to display the corresponding x.lab or y.lab*

*# custom.par: list that provides the parameters that reset all the graphics parameters. BEWARE: if NULL and par.rest == TRUE, the default par() parameters are used*

*# RETURN*

*# a list containing::*

*# $x.mid.left.fig.region: middle of the left margin of the figure region, in coordinates of the x-axis*

*# $x.left.fig.region: left side of the left margin, in coordinates of the x-axis*

*# $x.mid.right.fig.region: middle of the right margin of the figure region, in coordinates of the x-axis*

*# $x.right.fig.region: right side of the right margin, in coordinates of the x-axis*

*# $x.left.plot.region: left side of the plot region, in coordinates of the x-axis*

*# $x.right.plot.region: right side of the plot region, in coordinates of the x-axis*

*# $x.mid.plot.region: middle of the plot region, in coordinates of the x-axis*

*# $y.mid.bottom.fig.region: middle of the bottom margin of the figure region, in coordinates of the y-axis*

*# $y.bottom.fig.region: bottom of the bottom margin of the figure region, in coordinates of the y-axis*

*# $y.mid.top.fig.region: middle of the top margin of the figure region, in coordinates of the y-axis*

*# $y.top.fig.region: top of the top margin of the figure region, in coordinates of the y-axis*

*# $y.top.plot.region: top of the plot region, in coordinates of the y-axis*

*# $y.bottom.plot.region: bottom of the plot region, in coordinates of the y-axis*

*# $y.mid.plot.region: middle of the plot region, in coordinates of the y-axis*

*# $text: warning text*

*# EXAMPLES*

*# Example of log axis with log y-axis and unmodified x-axis:*

*# prior.par <- fun\_graph\_param\_prior\_plot(param.reinitial = TRUE, xlog.scale = FALSE, ylog.scale = TRUE, remove.label = TRUE, remove.x.axis = FALSE, remove.y.axis = TRUE, down.space = 1, left.space = 1, up.space = 1, right.space = 1, orient = 1, dist.legend = 0.5, tick.length = 0.5, box.type = "n", amplif.label = 1, amplif.axis = 1, display.extend = FALSE, return.par = TRUE) ; plot(1:100, log = "y") ; fun\_feature\_post\_plot(y.side = 2, x.lab = "Values", y.lab = "TEST", y.dist.legend = 0.7, y.log.scale = prior.par$ylog, just.label.add = ! prior.par$ann)*

*# Example of log axis with redrawn x-axis and y-axis:*

*# prior.par <- fun\_graph\_param\_prior\_plot(param.reinitial = TRUE) ; plot(1:100) ; fun\_feature\_post\_plot(x.side = 1, x.lab = "Values", y.side = 2, y.lab = "TEST", y.dist.legend = 0.7)*

*# DEBUGGING*

*# x.side = 0 ; x.categ = NULL ; x.categ.pos = NULL ; x.lab = "" ; x.dist.legend = 1 ; x.log.scale = FALSE ; x.nb.inter.tick = 1 ; y.side = 0 ; y.categ = NULL ; y.categ.pos = NULL ; y.lab = "" ; y.dist.legend = 0.7 ; y.log.scale = FALSE ; y.nb.inter.tick = 1 ; text.angle = 90 ; tick.length = 0.5 ; sec.tick.length = 0.3 ; bg.color = NULL ; grid.lwd = NULL ; grid.col = "white" ; corner.text = "" ; magnific = 1.5 ; magnific.corner.text = 1 ; par.reset = FALSE ; custom.par = NULL # for function debugging*

*# argument checking*

**arg.check <- NULL** *# for function debbuging*

**checked.arg.names <- NULL** *# for function debbuging*

**ee <- expression(arg.check <- c(arg.check, tempo$problem) , checked.arg.names <- c(checked.arg.names, tempo$param.name))**

**tempo <- fun\_param\_check(data = x.side, options = c(0, 1, 3), length = 1) ; eval(ee)**

**if( ! is.null(x.categ)){**

**tempo <- fun\_param\_check(data = x.categ, class = "character", na.contain = TRUE) ; eval(ee)**

**}**

**if( ! is.null(x.categ.pos)){**

**tempo <- fun\_param\_check(data = x.categ.pos, class = "numeric") ; eval(ee)**

**}**

**tempo <- fun\_param\_check(data = x.lab, class = "character", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = x.dist.legend, class = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = x.log.scale, class = "logical", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = x.nb.inter.tick, typeof = "integer", length = 1, double.as.integer.allowed = TRUE) ; eval(ee)**

**tempo <- fun\_param\_check(data = y.side, options = c(0, 2, 4), length = 1) ; eval(ee)**

**if( ! is.null(y.categ)){**

**tempo <- fun\_param\_check(data = y.categ, class = "character", na.contain = TRUE) ; eval(ee)**

**}**

**if( ! is.null(y.categ.pos)){**

**tempo <- fun\_param\_check(data = y.categ.pos, class = "numeric") ; eval(ee)**

**}**

**tempo <- fun\_param\_check(data = y.lab, class = "character", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = y.dist.legend, class = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = y.log.scale, class = "logical", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = y.nb.inter.tick, typeof = "integer", length = 1, double.as.integer.allowed = TRUE) ; eval(ee)**

**tempo <- fun\_param\_check(data = text.angle, class = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = tick.length, class = "numeric", length = 1, prop = TRUE) ; eval(ee)**

**tempo <- fun\_param\_check(data = sec.tick.length, class = "numeric", length = 1, prop = TRUE) ; eval(ee)**

**if( ! is.null(bg.color)){**

**tempo <- fun\_param\_check(data = bg.color, class = "character", length = 1) ; eval(ee)**

**if( ! (bg.color %in% colors() | grepl(pattern = "^#", bg.color))){** *# check color*

**tempo.cat <- paste0("\n\n================\n\nERROR: bg.color ARGUMENT MUST BE A HEXADECIMAL COLOR VECTOR STARTING BY # OR A COLOR NAME GIVEN BY colors()\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}**

**if( ! is.null(grid.lwd)){**

**tempo <- fun\_param\_check(data = grid.lwd, class = "numeric", neg.values = FALSE) ; eval(ee)**

**}**

**if( ! is.null(grid.col)){**

**tempo <- fun\_param\_check(data = grid.col, class = "character", length = 1) ; eval(ee)**

**if( ! (grid.col %in% colors() | grepl(pattern = "^#", grid.col))){** *# check color*

**tempo.cat <- paste0("\n\n================\n\nERROR: grid.col ARGUMENT MUST BE A HEXADECIMAL COLOR VECTOR STARTING BY # OR A COLOR NAME GIVEN BY colors()\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**}**

**tempo <- fun\_param\_check(data = corner.text, class = "character", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = magnific, class = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = magnific.corner.text, class = "numeric", length = 1, neg.values = FALSE) ; eval(ee)**

**tempo <- fun\_param\_check(data = just.label.add, class = "logical", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = par.reset, class = "logical", length = 1) ; eval(ee)**

**if( ! is.null(custom.par)){**

**tempo <- fun\_param\_check(data = custom.par, typeof = "list", length = 1) ; eval(ee)**

**}**

**if(any(arg.check) == TRUE){**

**stop()**

**}**

*# source("C:/Users/Gael/Documents/Sources/debugging\_tools\_for\_r\_dev-v1.2/r\_debugging\_tools-v1.2.R") ; eval(parse(text = str\_basic\_arg\_check\_dev)) ; eval(parse(text = str\_arg\_check\_with\_fun\_param\_check\_dev)) # to check arguments status and if they have been checked using fun\_param\_check()*

*# end argument checking*

**text <- NULL**

**par(tcl = -par()$mgp[2] \* tick.length)**

**if(x.log.scale == TRUE){**

**grid.coord.x <- c(10^par("usr")[1], 10^par("usr")[2])**

**}else{**

**grid.coord.x <- c(par("usr")[1], par("usr")[2])**

**}**

**if(y.log.scale == TRUE){**

**grid.coord.y <- c(10^par("usr")[3], 10^par("usr")[4])**

**}else{**

**grid.coord.y <- c(par("usr")[3], par("usr")[4])**

**}**

**if( ! is.null(bg.color)){**

**rect(grid.coord.x[1], grid.coord.y[1], grid.coord.x[2], grid.coord.y[2], col = bg.color, border = NA)**

**}**

**if( ! is.null(grid.lwd)){**

**grid(nx = NA, ny = NULL, col = grid.col, lty = 1, lwd = grid.lwd)**

**}**

**if(x.log.scale == TRUE){**

**x.mid.left.fig.region <- 10^(par("usr")[1] - ((par("usr")[2] - par("usr")[1]) / (par("plt")[2] - par("plt")[1])) \* par("plt")[1] / 2)** *# in x coordinates, to position axis labeling at the bottom of the graph (according to x scale)*

**x.left.fig.region <- 10^(par("usr")[1] - ((par("usr")[2] - par("usr")[1]) / (par("plt")[2] - par("plt")[1])) \* par("plt")[1])** *# in x coordinates*

**x.mid.right.fig.region <- 10^(par("usr")[2] + ((par("usr")[2] - par("usr")[1]) / (par("plt")[2] - par("plt")[1])) \* (1 - par("plt")[2]) / 2)** *# in x coordinates, to position axis labeling at the top of the graph (according to x scale)*

**x.right.fig.region <- 10^(par("usr")[2] + ((par("usr")[2] - par("usr")[1]) / (par("plt")[2] - par("plt")[1])) \* (1 - par("plt")[2]))** *# in x coordinates*

**x.left.plot.region <- 10^par("usr")[1]** *# in x coordinates, left of the plot region (according to x scale)*

**x.right.plot.region <- 10^par("usr")[2]** *# in x coordinates, right of the plot region (according to x scale)*

**x.mid.plot.region <- 10^((par("usr")[2] + par("usr")[1]) / 2)** *# in x coordinates, right of the plot region (according to x scale)*

**}else{**

**x.mid.left.fig.region <- (par("usr")[1] - ((par("usr")[2] - par("usr")[1]) / (par("plt")[2] - par("plt")[1])) \* par("plt")[1] / 2)** *# in x coordinates, to position axis labeling at the bottom of the graph (according to x scale)*

**x.left.fig.region <- (par("usr")[1] - ((par("usr")[2] - par("usr")[1]) / (par("plt")[2] - par("plt")[1])) \* par("plt")[1])** *# in x coordinates*

**x.mid.right.fig.region <- (par("usr")[2] + ((par("usr")[2] - par("usr")[1]) / (par("plt")[2] - par("plt")[1])) \* (1 - par("plt")[2]) / 2)** *# in x coordinates, to position axis labeling at the top of the graph (according to x scale)*

**x.right.fig.region <- (par("usr")[2] + ((par("usr")[2] - par("usr")[1]) / (par("plt")[2] - par("plt")[1])) \* (1 - par("plt")[2]))** *# in x coordinates*

**x.left.plot.region <- par("usr")[1]** *# in x coordinates, left of the plot region (according to x scale)*

**x.right.plot.region <- par("usr")[2]** *# in x coordinates, right of the plot region (according to x scale)*

**x.mid.plot.region <- (par("usr")[2] + par("usr")[1]) / 2** *# in x coordinates, right of the plot region (according to x scale)*

**}**

**if(y.log.scale == TRUE){**

**y.mid.bottom.fig.region <- 10^(par("usr")[3] - ((par("usr")[4] - par("usr")[3]) / (par("plt")[4] - par("plt")[3])) \* par("plt")[3] / 2)** *# in y coordinates, to position axis labeling at the bottom of the graph (according to y scale). Ex mid.bottom.space*

**y.bottom.fig.region <- 10^(par("usr")[3] - ((par("usr")[4] - par("usr")[3]) / (par("plt")[4] - par("plt")[3])) \* par("plt")[3])** *# in y coordinates*

**y.mid.top.fig.region <- 10^(par("usr")[4] + ((par("usr")[4] - par("usr")[3]) / (par("plt")[4] - par("plt")[3])) \* (1 - par("plt")[4]) / 2)** *# in y coordinates, to position axis labeling at the top of the graph (according to y scale). Ex mid.top.space*

**y.top.fig.region <- 10^(par("usr")[4] + ((par("usr")[4] - par("usr")[3]) / (par("plt")[4] - par("plt")[3])) \* (1 - par("plt")[4]))** *# in y coordinates*

**y.top.plot.region <- 10^par("usr")[4]** *# in y coordinates, top of the plot region (according to y scale)*

**y.bottom.plot.region <- 10^par("usr")[3]** *# in y coordinates, bottom of the plot region (according to y scale)*

**y.mid.plot.region <- (par("usr")[3] + par("usr")[4]) / 2** *# in x coordinates, right of the plot region (according to x scale)*

**}else{**

**y.mid.bottom.fig.region <- (par("usr")[3] - ((par("usr")[4] - par("usr")[3]) / (par("plt")[4] - par("plt")[3])) \* par("plt")[3] / 2)** *# in y coordinates, to position axis labeling at the bottom of the graph (according to y scale). Ex mid.bottom.space*

**y.bottom.fig.region <- (par("usr")[3] - ((par("usr")[4] - par("usr")[3]) / (par("plt")[4] - par("plt")[3])) \* par("plt")[3])** *# in y coordinates*

**y.mid.top.fig.region <- (par("usr")[4] + ((par("usr")[4] - par("usr")[3]) / (par("plt")[4] - par("plt")[3])) \* (1 - par("plt")[4]) / 2)** *# in y coordinates, to position axis labeling at the top of the graph (according to y scale). Ex mid.top.space*

**y.top.fig.region <- (par("usr")[4] + ((par("usr")[4] - par("usr")[3]) / (par("plt")[4] - par("plt")[3])) \* (1 - par("plt")[4]))** *# in y coordinates*

**y.top.plot.region <- par("usr")[4]** *# in y coordinates, top of the plot region (according to y scale)*

**y.bottom.plot.region <- par("usr")[3]** *# in y coordinates, bottom of the plot region (according to y scale)*

**y.mid.plot.region <- 10^((par("usr")[3] + par("usr")[4]) / 2)** *# in x coordinates, right of the plot region (according to x scale)*

**}**

**if(x.side == 1 | x.side == 3){**

**par(xpd=FALSE, xaxt="s")**

**if(is.null(x.categ) & x.log.scale == TRUE){**

**if(any(par()$xaxp[1:2] == 0)){**

**if(par()$xaxp[1] == 0){**

**par(xaxp = c(10^-30, par()$xaxp[2:3]))** *# because log10(par()$xaxp[1] == 0) == -Inf*

**}**

**if(par()$xaxp[2] == 0){**

**par(xaxp = c(par()$xaxp[1], 10^-30, par()$xaxp[3]))** *# because log10(par()$xaxp[2] == 0) == -Inf*

**}**

**}**

**axis(side=x.side, at=c(10^par()$usr[1], 10^par()$usr[2]), labels=rep("", 2), lwd=1, lwd.ticks=0)** *# draw the axis line*

**mtext(side = x.side, text = x.lab, line = x.dist.legend / 0.2, las = 0, cex = magnific)**

**par(tcl = -par()$mgp[2] \* sec.tick.length)** *# length of the secondary ticks are reduced*

**suppressWarnings(rug(10^outer(c((log10(par("xaxp")[1]) -1):log10(par("xaxp")[2])), log10(1:10), "+"), ticksize = NA, side = x.side))** *# ticksize = NA to allow the use of par()$tcl value*

**par(tcl = -par()$mgp[2] \* tick.length)** *# back to main ticks*

**axis(side = x.side, at = c(1e-15, 1e-14, 1e-13, 1e-12, 1e-11, 1e-10, 1e-9, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6, 1e7, 1e8, 1e9, 1e10), labels = expression(10^-15, 10^-14, 10^-13, 10^-12, 10^-11, 10^-10, 10^-9, 10^-8, 10^-7, 10^-6, 10^-5, 10^-4, 10^-3, 10^-2, 10^-1, 10^0, 10^1, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 10^8, 10^9, 10^10), lwd = 0, lwd.ticks = 1, cex.axis = magnific)**

**x.text <- 10^par("usr")[2]**

**}else if(is.null(x.categ) & x.log.scale == FALSE){**

**axis(side=x.side, at=c(par()$usr[1], par()$usr[2]), labels=rep("", 2), lwd=1, lwd.ticks=0)** *# draw the axis line*

**axis(side=x.side, at=round(seq(par()$xaxp[1], par()$xaxp[2], length.out=par()$xaxp[3]+1), 2))**

**mtext(side = x.side, text = x.lab, line = x.dist.legend / 0.2, las = 0, cex = magnific)**

**if(x.nb.inter.tick > 0){**

**inter.tick.unit <- (par("xaxp")[2] - par("xaxp")[1]) / par("xaxp")[3]**

**par(tcl = -par()$mgp[2] \* sec.tick.length)** *# length of the ticks are reduced*

**suppressWarnings(rug(seq(par("xaxp")[1] - 10 \* inter.tick.unit, par("xaxp")[2] + 10 \* inter.tick.unit, by = inter.tick.unit / (1 + x.nb.inter.tick)), ticksize = NA, x.side))** *# ticksize = NA to allow the use of par()$tcl value*

**par(tcl = -par()$mgp[2] \* tick.length)** *# back to main ticks*

**}**

**x.text <- par("usr")[2]**

**}else if(( ! is.null(x.categ)) & x.log.scale == FALSE){**

**if(is.null(x.categ.pos)){**

**x.categ.pos <- 1:length(x.categ)**

**}else if(length(x.categ.pos) != length(x.categ)){**

**stop("\n\nPROBLEM: x.categ.pos MUST BE THE SAME LENGTH AS x.categ\n\n")**

**}**

**par(xpd = TRUE)**

**if(x.side == 1){**

**segments(x0 = x.left.plot.region, x1 = x.right.plot.region, y0 = y.bottom.plot.region, y1 = y.bottom.plot.region)** *# draw the line of the axis*

**text(x = x.categ.pos, y = y.mid.bottom.fig.region, labels = x.categ, srt = text.angle, cex = magnific)**

**}else if(x.side == 3){**

**segments(x0 = x.left.plot.region, x1 = x.right.plot.region, y0 = y.top.plot.region, y1 = y.top.plot.region)** *# draw the line of the axis*

**text(x = x.categ.pos, y = y.mid.top.fig.region, labels = x.categ, srt = text.angle, cex = magnific)**

**}else{**

**stop("\n\nARGUMENT x.side CAN ONLY BE 1 OR 3\n\n")**

**}**

**par(xpd = FALSE)**

**x.text <- par("usr")[2]**

**}else{**

**stop("\n\nPROBLEM WITH THE x.side (", x.side ,") OR x.log.scale (", x.log.scale,") ARGUMENTS\n\n")**

**}**

**}else{**

**x.text <- par("usr")[2]**

**}**

**if(y.side == 2 | y.side == 4){**

**par(xpd=FALSE, yaxt="s")**

**if(is.null(y.categ) & y.log.scale == TRUE){**

**if(any(par()$yaxp[1:2] == 0)){**

**if(par()$yaxp[1] == 0){**

**par(yaxp = c(10^-30, par()$yaxp[2:3]))** *# because log10(par()$yaxp[1] == 0) == -Inf*

**}**

**if(par()$yaxp[2] == 0){**

**par(yaxp = c(par()$yaxp[1], 10^-30, par()$yaxp[3]))** *# because log10(par()$yaxp[2] == 0) == -Inf*

**}**

**}**

**axis(side=y.side, at=c(10^par()$usr[3], 10^par()$usr[4]), labels=rep("", 2), lwd=1, lwd.ticks=0)** *# draw the axis line*

**mtext(side = y.side, text = y.lab, line = y.dist.legend / 0.2, las = 0, cex = magnific)**

**par(tcl = -par()$mgp[2] \* sec.tick.length)** *# length of the ticks are reduced*

**suppressWarnings(rug(10^outer(c((log10(par("yaxp")[1])-1):log10(par("yaxp")[2])), log10(1:10), "+"), ticksize = NA, side = y.side))** *# ticksize = NA to allow the use of par()$tcl value*

**par(tcl = -par()$mgp[2] \* tick.length)** *# back to main tick length*

**axis(side = y.side, at = c(1e-15, 1e-14, 1e-13, 1e-12, 1e-11, 1e-10, 1e-9, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6, 1e7, 1e8, 1e9, 1e10), labels = expression(10^-15, 10^-14, 10^-13, 10^-12, 10^-11, 10^-10, 10^-9, 10^-8, 10^-7, 10^-6, 10^-5, 10^-4, 10^-3, 10^-2, 10^-1, 10^0, 10^1, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 10^8, 10^9, 10^10), lwd = 0, lwd.ticks = 1, cex.axis = magnific)**

**y.text <- 10^(par("usr")[4] + (par("usr")[4] - par("usr")[3]) / (par("plt")[4] - par("plt")[3]) \* (1 - par("plt")[4]))**

**mtext(side = y.side, text = y.lab, line = y.dist.legend / 0.2, las = 0, cex = magnific)**

**}else if(is.null(y.categ) & y.log.scale == FALSE){**

**axis(side=y.side, at=c(par()$usr[3], par()$usr[4]), labels=rep("", 2), lwd=1, lwd.ticks=0)** *# draw the axis line*

**axis(side=y.side, at=round(seq(par()$yaxp[1], par()$yaxp[2], length.out=par()$yaxp[3]+1), 2))**

**mtext(side = y.side, text = y.lab, line = y.dist.legend / 0.2, las = 0, cex = magnific)**

**if(y.nb.inter.tick > 0){**

**inter.tick.unit <- (par("yaxp")[2] - par("yaxp")[1]) / par("yaxp")[3]**

**par(tcl = -par()$mgp[2] \* sec.tick.length)** *# length of the ticks are reduced*

**suppressWarnings(rug(seq(par("yaxp")[1] - 10 \* inter.tick.unit, par("yaxp")[2] + 10 \* inter.tick.unit, by = inter.tick.unit / (1 + y.nb.inter.tick)), ticksize = NA, side=y.side))** *# ticksize = NA to allow the use of par()$tcl value*

**par(tcl = -par()$mgp[2] \* tick.length)** *# back to main tick length*

**}**

**y.text <- (par("usr")[4] + (par("usr")[4] - par("usr")[3]) / (par("plt")[4] - par("plt")[3]) \* (1 - par("plt")[4]))**

**}else if(( ! is.null(y.categ)) & y.log.scale == FALSE){**

**if(is.null(y.categ.pos)){**

**y.categ.pos <- 1:length(y.categ)**

**}else if(length(y.categ.pos) != length(y.categ)){**

**stop("\n\nPROBLEM: y.categ.pos MUST BE THE SAME LENGTH AS y.categ\n\n")**

**}**

**axis(side = y.side, at = y.categ.pos, labels = rep("", length(y.categ)), lwd=0, lwd.ticks=1)** *# draw the line of the axis*

**par(xpd = TRUE)**

**if(y.side == 2){**

**text(x = x.mid.left.fig.region, y = y.categ.pos, labels = y.categ, srt = text.angle, cex = magnific)**

**}else if(y.side == 4){**

**text(x = x.mid.right.fig.region, y = y.categ.pos, labels = y.categ, srt = text.angle, cex = magnific)**

**}else{**

**stop("\n\nARGUMENT y.side CAN ONLY BE 2 OR 4\n\n")**

**}**

**par(xpd = FALSE)**

**y.text <- (par("usr")[4] + (par("usr")[4] - par("usr")[3]) / (par("plt")[4] - par("plt")[3]) \* (1 - par("plt")[4]))**

**}else{**

**stop("\n\nPROBLEM WITH THE y.side (", y.side ,") OR y.log.scale (", y.log.scale,") ARGUMENTS\n\n")**

**}**

**}else{**

**y.text <- (par("usr")[4] + (par("usr")[4] - par("usr")[3]) / (par("plt")[4] - par("plt")[3]) \* (1 - par("plt")[4]))**

**}**

**par(xpd=NA)**

**text(x = x.mid.right.fig.region, y = y.text, corner.text, adj=c(1, 1.1), cex = magnific.corner.text)** *# text at the topright corner. Replace x.right.fig.region by x.text if text at the right edge of the plot region*

**if(just.label.add == TRUE & x.side == 0 & x.lab != ""){**

**text(x = x.mid.plot.region, y = y.mid.bottom.fig.region, x.lab, adj=c(0.5, 0.5), cex = magnific)** *# x label*

**}**

**if(just.label.add == TRUE & y.side == 0 & y.lab != ""){**

**text(x = y.mid.plot.region, y = x.mid.left.fig.region, y.lab, adj=c(0.5, 0.5), cex = magnific)** *# x label*

**}**

**par(xpd=FALSE)**

**if(par.reset == TRUE){**

**tempo.par <- fun\_open\_window(pdf.disp = FALSE, return.output = TRUE)**

**invisible(dev.off())** *# close the new window*

**if( ! is.null(custom.par)){**

**if( ! names(custom.par) %in% names(tempo.par$ini.par)){**

**tempo.cat <- paste0("\n\n================\n\nERROR: custom.par ARGUMENT SHOULD HAVE THE NAMES OF THE COMPARTMENT LIST COMING FROM THE par() LIST\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**par(custom.par)**

**text <- c(text, "\nGRAPH PARAMETERS SET TO VALUES DEFINED BY custom.par ARGUMENT\n")**

**}else{**

**par(tempo.par$ini.par)**

**text <- c(text, "\nGRAPH PARAMETERS RESET TO par() DEFAULT VALUES\n")**

**}**

**}**

**output <- list(x.mid.left.fig.region = x.mid.left.fig.region, x.left.fig.region = x.left.fig.region, x.mid.right.fig.region = x.mid.right.fig.region, x.right.fig.region = x.right.fig.region, x.left.plot.region = x.left.plot.region, x.right.plot.region = x.right.plot.region, x.mid.plot.region = x.mid.plot.region, y.mid.bottom.fig.region = y.mid.bottom.fig.region, y.bottom.fig.region = y.bottom.fig.region, y.mid.top.fig.region = y.mid.top.fig.region, y.top.fig.region = y.top.fig.region, y.top.plot.region = y.top.plot.region, y.bottom.plot.region = y.bottom.plot.region, y.mid.plot.region = y.mid.plot.region, text = text)**

**return(output)**

**}**

######## fun\_close\_specif\_window() #### Closing specific graphic windows

**# Check OK: clear to go Apollo**

**fun\_close\_specif\_window <- function(kind = "pdf", return.text = FALSE){**

*# AIM:*

*# close only specific graphic windows (devices)*

*# REQUIRED FUNCTIONS*

*# fun\_param\_check()*

*# ARGUMENTS:*

*# kind: vector, among c("windows", "quartz", "x11", "X11", "pdf", "bmp", "png", "tiff"), indicating the kind of graphic windows (devices) to close. BEWARE: either "windows", "quartz", "x11" or "X11" means that all the X11 GUI graphics devices will be closed, whatever the OS used*

*# return.text: print text regarding the kind parameter and the devices that were finally closed?*

*# RETURN*

*# text regarding the kind parameter and the devices that were finally closed*

*# EXAMPLES*

*# windows() ; windows() ; pdf() ; dev.list() ; fun\_close\_specif\_window(kind = c("pdf", "x11"), return.text = TRUE) ; dev.list()*

*# DEBUGGING*

*# kind = c("windows", "pdf") ; return.text = FALSE # for function debugging*

*# argument checking*

**arg.check <- NULL** *# for function debbuging*

**checked.arg.names <- NULL** *# for function debbuging*

**ee <- expression(arg.check <- c(arg.check, tempo$problem) , checked.arg.names <- c(checked.arg.names, tempo$param.name))**

**tempo <- fun\_param\_check(data = kind, options = c("windows", "quartz", "x11", "X11", "pdf", "bmp", "png", "tiff")) ; eval(ee)**

**tempo <- fun\_param\_check(data = return.text, class = "logical", length = 1) ; eval(ee)**

**if(any(arg.check) == TRUE){**

**stop()**

**}**

*# source("C:/Users/Gael/Documents/Sources/debugging\_tools\_for\_r\_dev-v1.2/r\_debugging\_tools-v1.2.R") ; eval(parse(text = str\_basic\_arg\_check\_dev)) ; eval(parse(text = str\_arg\_check\_with\_fun\_param\_check\_dev)) # to check arguments status and if they have been checked using fun\_param\_check()*

*# end argument checking*

**text <- paste0("THE REQUIRED KIND OF GRAPHIC DEVICES TO CLOSE ARE ", paste(kind, collapse = " "))**

**if(Sys.info()["sysname"] == "Windows"){** *# Note that .Platform$OS.type() only says "unix" for macOS and Linux and "Windows" for Windows*

**if(any(kind %in% c("windows", "quartz", "x11", "X11"))){**

**tempo <- kind %in% c("windows", "quartz", "x11", "X11")**

**kind[tempo] <- "windows"** *#* *# term are replaced by what is displayed when using a <- dev.list() ; names(a)*

**}**

**}else if(Sys.info()["sysname"] == "Linux"){**

**if(any(kind %in% c("windows", "quartz", "x11", "X11"))){**

**tempo.device <- suppressWarnings(try(X11(), silent = TRUE))[]** *# open a X11 window to try to recover the X11 system used*

**if( ! is.null(tempo.device)){**

**text <- paste0(text, "\nCANNOT CLOSE GUI GRAPHIC DEVICES AS REQUIRED BECAUSE THIS LINUX SYSTEM DOES NOT HAVE IT")**

**}else{**

**tempo <- kind %in% c("windows", "quartz", "x11", "X11")**

**kind[tempo] <- names(dev.list()[length(dev.list())])** *#* *# term are replaced by what is displayed when using a <- dev.list() ; names(a)*

**invisible(dev.off())** *# close the X11 opened by tempo*

**}**

**}**

**}else{** *# for macOS*

**if(any(kind %in% c("windows", "quartz", "x11", "X11"))){**

**tempo <- kind %in% c("windows", "quartz", "x11", "X11")**

**kind[tempo] <- "quartz"** *# term are replaced by what is displayed when using a <- dev.list() ; names(a)*

**}**

**}**

**kind <- unique(kind)**

**if(length(dev.list()) != 0){**

**for(i in length(names(dev.list())):1){**

**if(names(dev.list())[i] %in% kind){**

**text <- paste0(text, "\n", names(dev.list())[i], " DEVICE NUMBER ", dev.list()[i], " HAS BEEN CLOSED")**

**invisible(dev.off(dev.list()[i]))**

**}**

**}**

**}**

**if(return.text == TRUE){**

**return(text)**

**}**

**}**

################ Exporting results (text & tables)

######## fun\_export\_data() #### Print string or data object into output file

**# Check OK: clear to go Apollo**

**fun\_export\_data <- function(data, output ="results.txt", path = "C:/Users/Gael/Desktop", no.overwrite = TRUE, rownames.kept = FALSE, vector.cat = FALSE, sep = 2){**

*# AIM:*

*# print a string or a data object into an output file*

*# REQUIRED FUNCTIONS*

*# fun\_param\_check()*

*# ARGUMENTS*

*# data: object to print in the output file*

*# output: name of the output file*

*# path: location of the output file*

*# no.overwrite: (logical) if output file already exists, defines if the printing is appended (default TRUE) or if the output file content is erased before printing (FALSE)*

*# rownames.kept: (logical) defines whether row names have to be removed or not in small tables (less than length.rows rows)*

*# vector.cat (logical). If TRUE print a vector of length > 1 using cat() instead of capture.output(). Otherwise (default FALSE) the opposite*

*# sep: number of separating lines after printed data (must be integer)*

*# RETURN*

*# nothing*

*# EXAMPLES*

*# fun\_export\_data(data = 1:3, output = "results.txt", path = "C:/Users/Gael/Desktop", no.overwrite = TRUE, rownames.kept = FALSE, vector.cat = FALSE, sep = 2)*

*# DEBUGGING*

*# data = 1:3 ; output = "results.txt" ; path = "C:/Users/Gael/Desktop" ; no.overwrite = TRUE ; rownames.kept = FALSE ; vector.cat = FALSE ; sep = 2 # for function debugging*

*# argument checking*

**arg.check <- NULL** *# for function debbuging*

**checked.arg.names <- NULL** *# for function debbuging*

**ee <- expression(arg.check <- c(arg.check, tempo$problem) , checked.arg.names <- c(checked.arg.names, tempo$param.name))**

**tempo <- fun\_param\_check(data = output, class = "character", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = path, class = "character", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = no.overwrite, class = "logical", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = rownames.kept, class = "logical", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = vector.cat, class = "logical", length = 1) ; eval(ee)**

**tempo <- fun\_param\_check(data = sep, typeof = "integer", length = 1, double.as.integer.allowed = TRUE) ; eval(ee)**

**if(any(arg.check) == TRUE){**

**stop()**

**}**

*# source("C:/Users/Gael/Documents/Sources/debugging\_tools\_for\_r\_dev-v1.2/r\_debugging\_tools-v1.2.R") ; eval(parse(text = str\_basic\_arg\_check\_dev)) ; eval(parse(text = str\_arg\_check\_with\_fun\_param\_check\_dev)) # to check arguments status and if they have been checked using fun\_param\_check()*

*# the 4 next lines are just to know how to detect a missing argument. Important here because if data is not provided, print the code of the data function*

**arg.user.list <- as.list(match.call(expand.dots=FALSE))[-1]** *# recover all the arguments provided by the function user (excluding the argument with defaults values not provided by the user. Thus, it is really the list indicated by the user)*

**default.arg.list <- formals(fun = sys.function(sys.parent()))** *# list of all the arguments of the function with their default values (not the values of the user !). It seems that ls() as first line of the function provide the names of the arguments (empty, called, etc., or not)*

**arg.without.default.value <- sapply(default.arg.list, is.symbol) & sapply(sapply(default.arg.list, as.character), identical, "")** *# logical to detect argument without default values (these are typeof "symbol" and class "name" and empty character*

**if( ! all(names(default.arg.list)[arg.without.default.value] %in% names(arg.user.list))){** *# test that the arguments with no null values are provided by the user*

**tempo.cat <- paste0("\n\n================\n\nERROR: VALUE REQUIRED FOR THESE ARGUMENTS WITH NO DEFAULTS VALUES: ", paste(names(default.arg.list)[arg.without.default.value][ ! names(default.arg.list)[arg.without.default.value] %in% names(arg.user.list)], collapse = " "), "\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**if(output == ""){**

**tempo.cat <- paste0("\n\n================\n\nERROR: output ARGUMENT DOES NOT CORRESPOND TO A VALID CHARACTER STRING\n\n================\n\n")**

**stop(tempo.cat)**

**}**

**if(dir.exists(path) == FALSE){**

**tempo.cat <- paste0("\n\n================\n\nERROR: path ARGUMENT DOES NOT CORRESPOND TO EXISTING DIRECTORY\n\n================\n\n")**

**stop(tempo.cat)**

**}**

*# end argument checking*

**if(all(class(data) %in% c("matrix", "data.frame", "table"))){**

**if(rownames.kept == FALSE & all(class(data) == "data.frame") & nrow(data) != 0 & nrow(data) <= 4){** *# for data frames with nrows <= 4*

**rownames.output.tables <- ""**

**length.rows <- nrow(data)**

**for(i in 1:length.rows){** *# replace the rownames of the first 4 rows by increasing number of spaces (beacause identical row names not allowed in data frames). This method cannot be extended to more rows as the printed data frame is shifted on the right because of "big empty rownames"*

**rownames.output.tables <- c(rownames.output.tables, paste0(rownames.output.tables[i]," ", collapse=""))**

**}**

**row.names(data) <- rownames.output.tables[1:length.rows]**

**}else if(rownames.kept == FALSE & all(class(data) %in% c("matrix", "table"))){**

**rownames(data) <- rep("", nrow(data))**  *# identical row names allowed in matrices and tables*

**}**

**capture.output(data, file=paste0(path, "/", output), append = no.overwrite)**

**}else if(is.vector(data) & all(class(data) != "list") & (length(data) == 1 | vector.cat == TRUE)){**

**cat(data, file= paste0(path, "/", output), append = no.overwrite)**

**}else{** *# other (array, list, factor or vector with vector.cat = FALSE)*

**capture.output(data, file=paste0(path, "/", output), append = no.overwrite)**

**}**

**sep.final <- paste0(rep("\n", sep), collapse = "")**

**write(sep.final, file= paste0(path, "/", output), append = TRUE)** *# add a sep*

**}**