Documentation and development policy for the CSTools package

The first key point for a successful development of a software package is to establish a set of common rules and best practices to be followed by the members involved in the development. These rules must define aspects such as:

a. How to keep track of the progress.
b. What guidelines the contributions have to fulfil.
c. How to contribute concurrently.
d. How to ensure the generated software works as expected.
e. How to handle discussions involving multiple members of the team.
f. When to generate documents.
g. What tools to use for any of the aspects above and how to use them.

This document summarizes the conventions and rules adopted by the CSTools development team.

a) Using a version control system (VCS) allows to keep track of all the changes in the code and documentation files of the project. This provides safety to the development as well as the ability to roll back changes and go back to older versions and to compare differences between subsequent versions.

b) Regarding the software documentation and code, some guidelines have been established. These are oriented to R because the existing software packages developed by the consortium are coded in R. If at some point in the project we decide to combine R with other languages, additional guidelines will have to be established.

As stated previously, all contributions will follow the standard R package structure. The package skeleton will be provided by the MEDSCOPE R package coordinators at BSC. In a nutshell, the code and documentation of each function will have to be in a separate file within the R folder of the package. The package is being developed in R 3.2.0.

Notice that, due to CRAN policy, the package should be of the minimum necessary size. To include data an agreement with the coordinator of the package should be reached.

roxygen2 is an R package that manages and generates the documentation (user guide) of the functions in an automated way, provided that the information is written in the code files following a specific syntax. All the code documentation will have to follow this syntax:
https://cran.r-project.org/web/packages/roxygen2/index.html

A very important general recommendation for the development of functionality is that the functions developed aim at processing the simplest data structure possible and that irrelevant complexities or extended data structures should be avoided. For example, a function to regrid a map, should be designed to receive as input a longitude-latitude array. A function to compute seasonal means should be designed to receive a time series and a corresponding vector of dates. See below an extended explanation of how functions should be designed in order to be included in CSTools.

In addition, to help files generated with roxygen2 documenting the R package, its datasets, and the usage and general functionality of the functions, we will also use inline comments to document specific function calls and routines directly in the source code. Code blocks are preceded by one or several lines of comments documenting the purpose of the following code block, whereas short comments on single lines are placed after the code on the same line.

For the coding style, we have adopted the Google's R style guide with some particularly discussed aspects:

- Whether to keep comments or not: we will keep comments. The emacs-ESS commenting style will be used: a single '#' symbol shifts the comment to the middle column, '##' keeps the comment aligned with the code, and '###' sends the comment to the left-most column
- Whether to keep blank lines or not: we will not keep blank lines
- Whether to replace all = assignments by the arrow assignment <-: yes
- Whether to put the opening braces in a new line or not: no
- Number of indentation spaces: 4
- Maximum length of the lines: soft limit of 80

formatR (and styleR) are R packages that automatically formats code to fulfill the guidelines with some adjustments. Its usage is not mandatory but can be helpful in some situations.

c) The usage of a VCS allows for concurrent developments. Since the development team is distributed in multiple locations, a distributed VCS is required, and a server has been set up to provide the project source to all the members in the development team. The selected VCS has been Git, and GitLab has been installed in the provider server to also allow for visual interaction with the project files and with other members.

In Git and all VCSs, it is possible to create copies of the source of the project without affecting the source of the others. Each of these copies is called a branch. It is important to define a strategy that defines when the development members should create a branch and when they should merge it back with the others' version. It is also known as a branching strategy. We will use the GitHub-flow branching strategy, which consists on using a master branch and a feature branch for each new development, enhancement, bugfix, ...

See https://guides.github.com/introduction/flow/ for an explanation, and see https://about.gitlab.com/2014/09/29/gitlab-flow/ for a comparison of 3 of the most widely used Git branching strategies.

It is also important to have a developing strategy that defines what are the best practices a developer has to follow for a safe development and what interactions should happen with other developers: we will open a GitLab merge request each time a feature is finished, where we mention two other developers that could review the code, and where a responsible is assigned (usually the most involved person in the development of the feature). Once a reviewer checks and approves the new feature, he/she notifies it in the merge request discussion. Then, after both reviewers have approved, the responsible will communicate to the Git project coordinator, who will:

- Merge the master branch into the feature branch to integrate any new changes
- Build and check the package
- Run the unit tests
- Do a final review and approve the feature
- If everything worked, merge the branch into master

It is sometimes too tedious to collaboratively edit text documents with a VCS. For this reason, Google Drive documents can be used to share drafts of documents that need to be reviewed by other members.

d) Recurrently testing software at an early stage is a crucial practice to mitigate the enormous expenses of detecting and fixing bugs after the software has been released and put in production. We will implement a double approach for testing any newly developed features:

- Going through a review procedure, carried out by 1 person within the same task, before a feature is accepted and merged into the master branch.
- Going through a set of unit tests, that the developer will have to provide with the feature, before a feature is accepted and merged into the master branch, potentially taking advantage of the R package 'testthat'. In CSTools, however, adding a set of 10-20 examples in the “examples” section of the documentation of each function will be enough.

Keeping a recipe of steps to be followed to compile and set up the software package is also useful to make this task as easy as possible as well as allowing non-technical users to use the package.

e) Discussions related to the development should be kept on the GitLab through the issue system. The main text of an issue can be modified afterwards to include conclusions reached in the issue discussion.

f) Index of used tools for each purpose:

- Version controlling, branching and developing strategy: Git, GitLab
- Document editing and sharing: Google Drive
- Documenting code: roxygen2
- Communication and discussion: GitLab issuing system
- Profiling: We will use R's system.time() function for most of the time profiling situations. For more detailed analyses, we can use also the microbenchmark package. Regarding the memory profiling, the package lineprof can help. See Wickham's http://adv-r.had.co.nz/Profiling.html for details.
- Development: We will use make to compile and set up automatically the developed software package. formatR can optionally be used to make the code policy-compliant. The code editor is not fixed. We will not depend on the usage of RStudio.

The following steps are required to join the CSTools project on Gitlab. Joining Gitlab is a necessary step to take part in the development and discussions:

1. Access the BSC GitLab portal: https://earth.bsc.es/gitlab/users/sign_in
2. Create an account (bottom right form, "New user? Create an account"). If you already have an account you can jump to step 4.
3. You will receive an e-mail with a confirmation link you have to open.
4. Send an e-mail to the mantainer mentioning your user name and saying you would like to join the developments/discussions.

After you are granted access, you will be able to clone and contribute to the git project (https://earth.bsc.es/gitlab/external/cstools) and to join the discussions at https://earth.bsc.es/gitlab/external/cstools/issues

Common procedure and conventions for adding new functions to

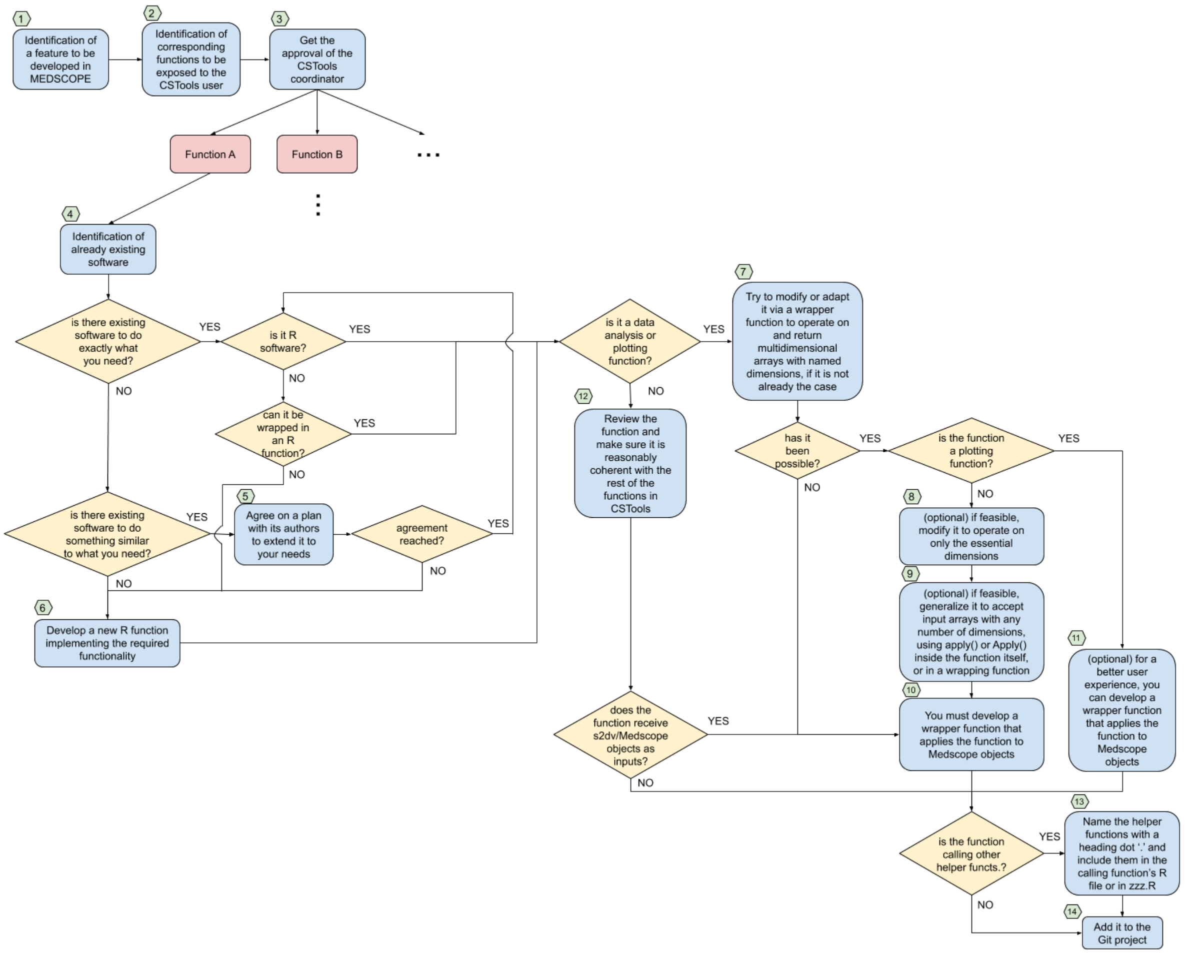
Since CSTools will gather contributions from multiple partners, defining a **common framework** is necessary in order to ensure an efficient and consistent combination of features in the software. This document presents a workflow to be followed when adding new features to CSTools in order to ensure a sustainable development of that common framework.

*Note: The development should be done in R 3.4.0.

The proposed workflow is represented in the diagram in figures 1 and 2. It covers a number of different situations that can be faced when willing to integrate a new feature in CSTools. The main steps in the workflow are shown in the list below. For each of the steps in the workflow, guidelines, best practices and examples are provided in a section of this document (numbered as shown in the list).

1. Identification of features to be developed
2. Identification of corresponding functions to be exposed to the user
3. Approval of the coordinator
4. Identification of already existing software
5. Coordination with authors of similar existing software
6. Development of new R functions if needed
7. Adaptation of functions to multidimensional arrays with named dimensions
8. Reduction of functions to operate on only essential dimensions
9. Application of “reduced” functions to arrays of any number of dimensions
10. Adaptation to the data structure via “wrapper” functions
11. Inclusion of plotting functions
12. Inclusion of functions to be exposed to the user, not devoted to analysis or plotting
13. Inclusion of helper functions not to be exposed to the user
14. Adding your contribution to the Git project
15. Adding checks to a function

This document also includes a section at the end with an example showing how a user would use a function that has been integrated following the presented workflow.



1
Identification of a feature to be developed in MEDSCOPE
2
Identification of corresponding functions to be exposed to the CSTools user
3
Get the approval of the CSTools coordinator
Function A
Function B
...
4
Identification of already existing software
is there existing software to do exactly what you need?
YES
NO
is it R software?
YES
NO
can it be wrapped in an R function?
YES
NO
is there existing software to do something similar to what you need?
YES
NO
5
Agree on a plan with its authors to extend it to your needs
agreement reached?
YES
NO
6
Develop a new R function implementing the required functionality
is it a data analysis or plotting function?
YES
NO
7
Try to modify or adapt it via a wrapper function to operate on and return multidimensional arrays with named dimensions, if it is not already the case
12
Review the function and make sure it is reasonably coherent with the rest of the functions in CSTools
has it been possible?
YES
NO
is the function a plotting function?
YES
NO
8
(optional) if feasible, modify it to operate on only the essential dimensions
9
(optional) if feasible, generalize it to accept input arrays with any number of dimensions, using apply() or Apply() inside the function itself, or in a wrapping function
10
You must develop a wrapper function that applies the function to Medscope objects
11
(optional) for a better user experience, you can develop a wrapper function that applies the function to Medscope objects
does the function receive s2dv/Medscope objects as inputs?
YES
NO
is the function calling other helper functs.?
YES
NO
13
Name the helper functions with a heading dot '.' and include them in the calling function's R file or in zzz.R
14
Add it to the Git project

Figure S1: A representation of the workflow for a sustainable integration of new features in CSTools.

1. Identification of features to be developed

The first fundamental step is to identify the features to be developed and included in CStools. Please, contact the package maintainer.

2. Identification of corresponding functions to be exposed to the user

Given a feature that needs to be developed, the next step is to identify the corresponding set of functions that should be exposed to the users, through the CSTool package, in order for them to carry out the procedure. Ideally, each feature should be associated with a single entry-point function, which should have a meaningful name and a concise set of parameters, although in some cases it will make sense to expose more than one function associated with a feature.

This effort at the first stage will ensure the package is tidy and easy to use. Below you can see what a CSTool script should look like ideally.

```
# Install and load R packages:
install_git('http://earth.bsc.es/gitlab/external/CSTools.git',
          branch = 'master')
library(CSTools)

# Loading the data
data <- CST_Load(var = 'tas', exp = 'ecmwf-s5', ...)

# Extracting the target season
data_season <- SeasonSelect(data, ...)

# Further data preparation code that might not make sense to
# include in a function, e.g. for conversion of units.
specific <- intermediate code

# Steps required for the analysis use case
anomalies <- AnomaliesCalculation(data_season, ...)
analysis <- Utility(anomalies, ...)
statistics <- MeasureOfPerformance(anomalies, ...)

# Plotting data and results
PlottingFunction(anomalies, ...)
PlotEquiMap(statistics, ...)
```

Except for specific calculations that the user may require for each specific use case, most of the lines of code should be calls to functions that directly receive outputs of previous functions as inputs. These blocks of information transferred from one function to the other are **CSTools objects** or "s2dv_cubes", and should comply with the data structure described below, thus preserving useful information attached to the data, such as the dates and latitude and longitude coordinates. Following the concept shown in this ideal script, effort should be put in the development of the CSTools functions in order to prevent the user from using "for" loops or other R control structures in the main script.

Note that as many functions as needed can be developed and included in the package in order to fulfil the project commitments, but not all of them necessarily have to be exposed as part of the package interface (API). Those that are not to be exposed can be named with a leading dot as detailed in step 13.

3. Approval of the coordinator

Once you have decided the set of functions to be exposed, please check with the prototype coordinators and wait for their approval. They will check once more that the set of proposed functions is consistent with the other functions and meaningful enough for a CSTools user not to get lost.

4. and 5. Identification of already existing software

Before starting work on any feature to be added, you should check whether similar functions already exist to carry out the same operation, both CSTools or in other R packages or any other software package, and explore ways of extending them if possible. This can save development time and result in a cleaner package.

If software exists to perform similar operations not exactly as desired, it is preferable to reach an agreement with the authors and try to define a collaborative development plan to make the most of what is already available.

If software exists that fulfils your requirements and it is not coded in the R language, you should check whether it is possible to call it from within R and wrapped in an R function. If this is possible, you can move on to the next steps (7. or 12.).

If no functions exist that perform the desired procedure, or they exist but would be too time consuming to extend/adapt, then proceed with its development as described in step 6.

6. Development of new R functions if needed

When a function needs to be developed from scratch, you should follow some tips for maximum compatibility with the CSTools package. The tips vary depending on the type of the function to be developed.

If it is a data analysis function (applying any kind of statistical analysis procedure to a data set), it should be designed to receive multidimensional arrays with named dimensions as main parameters, plus additional parameters, as well as return multidimensional arrays with named dimensions. The function should operate on arrays with only the essential dimensions relevant to the analysis procedure, and should be wrapped in a "wrapper" function that applies it to richer CSTools objects. See steps 7., 8., 9. and 10. for details.

If it is a plotting function, it should either be designed to receive multidimensional arrays with named dimensions as main parameters, plus additional parameters, or, if this is not possible and it operates another kind of data structure, it should be wrapped in a "wrapper" function that applies it to richer CSTools objects. Optionally, if the function has been developed to operate multidimensional arrays, a wrapper function can also be developed for a better user experience, although it can be left as is and it would still be user friendly, since CSTools objects are, in the end, collections of multidimensional arrays with named dimensions. See steps 7. and 11. for details.

If it is any other kind of function, it should be put in contrast with other similar functions in the package, if any, to make sure it is coherent with them, and finally be wrapped in a "wrapper" function. See steps 12. and 10. for details.

7. Adaptation of functions to multidimensional arrays with named dimensions

If a function to be included is a data analysis function (e.g. calibration, bias correction, verification, PCA, ...) or a plotting function, it should preferably be developed/adapted to ingest multidimensional arrays with named dimensions (plus additional parameters), and return multidimensional arrays with named dimensions. This will make integration with CSTools easier, as well as will make the function easier to use by the R community, and more flexible and powerful as shown in steps 8. and 9. Examples of non-data-analysis functions are functions to generate weights, or to subset or label data.

If the function is being adapted from already existing code, i.e. it is not being programmed from scratch, it could be wrapped in a wrapper function that provides and prepares incoming multidimensional arrays to be sent to the already existing function.

In any case, do not forget to document the expected inputs and outputs of this function, including the name and order of the dimensions of the ingested and returned arrays, as well as to include exhaustive parameter checks.

Here is an example of an adapted function which performs a weighted area average of a longitude-latitude-time array.

```
WeightedAreaAverage <- function(x, lats) {
  # x: numeric array with expected dimensions
  #    'time', 'lon' and 'lat'
  # lats: numeric vector with expected dimension
  #    'lat'

  # Check parameter 'x'
  # Check parameter 'lats'

  lat_weights <- cos(lats * pi / 180)

  res <- mean(apply(x, c(1, 2),
                    function(y) mean(y * lat_weights, na.rm = TRUE)
                   ),
              na.rm = TRUE)

  res
  # res: numeric array with dimensions
  #    'time', 'lon' and 'lat'
  # lats: numeric vector with expected dimension
  #    'lat'
}
```

If for some reason it is not possible to adapt the function to multidimensional arrays with named dimensions, a wrapper function can be developed directly which applies the function to be integrated to s2dv_cube objects (see step 10.). This is however not recommended since adapting the function to multidimensional arrays makes it more flexible, powerful and easier to reuse in other projects, as shown in steps 8. and 9.

8. Reduction of functions to operate on only essential dimensions

If you have been able to adapt your function to multidimensional arrays with named dimensions, the next desirable enhancement would be to "reduce" it to operate on arrays with as few dimensions as possible.

"A very important general recommendation for the development of functionality is that the functions developed aim at processing the simplest data structure possible and that irrelevant complexities or extended data structures should be avoided. For example, a function to regrid a map, should be designed to receive as input a longitude-latitude array. A function to compute seasonal means should be designed to receive a time series and a corresponding vector of dates."

Here is a "reduced" version of the example area-averaging function introduced in the previous section. The leading "R" in the name of the function stands for "reduced".

```
RWeightedAreaAverage<- function(x, lats) {
 # x: numeric array with expected dimensions
 #   'lon' and 'lat'
 # lats: numeric vector with expected dimension
 #   'lat'

 lat_weights <- cos(lats * pi / 180)
 res <- mean(apply(x, 1,
                   function(y) mean(y * lat_weights, na.rm = TRUE)
                  ),
             na.rm = TRUE)
 res
}
```

The advantages of developing this kind of “reduced” functions include the following:

- Concise and easy to understand
- Easy to maintain
- Easy to use by the R community, since they rely on base R vectors and arrays
- Easy to parallelize and be employed in HPC platforms

In some cases though, a function can be substantially more complex than the example shown here. For example, it could be a function that performs an operation on results obtained from other functions, each potentially with their own wrapper. In these cases, programming or adapting a function to be an atomic function would not be possible or would not make sense. If you are not sure whether your function is suitable for this or not, please contact the package maintainer.

9. Application of “reduced” functions to arrays of any number of dimensions

Once you have a function that operates with multidimensional arrays with named dimensions, it is recommended and relatively easy to adapt it to operate on multidimensional arrays with any number of additional non-essential dimensions. This can be achieved making use of the apply or multiApply::Apply functions, and can be done in two possible ways:

- Developing a generalizing wrapper function that applies a “reduced” function to multidimensional arrays with additional dimensions

- Extending the internals of a “reduced” function to accept multidimensional arrays with additional dimensions as inputs, and repeatedly apply the original procedure to them

In the example below, you can see how the multiApply::Apply function is being used to straightforwardly apply the “reduced” function used in the previous examples to multidimensional arrays with any number of dimensions, making use of a generalizing wrapper function.

```
WeightedAreaAverage <- function(data, lats, ncores = 1) {
  # Check parameter ‘data’
  # Check parameter ‘lats’
  # Check parameter 'ncores'
  Apply(data,
        target = c('lat', 'lon'),
        RWeightedAreaAverage,
        lats = lats,
        ncores = ncores)[[1]]
}
```

BONUS: Do you see any improvement that could make the RWeighted AreaAverage() and WeightedAreaAverage() functions faster? HINT: An atomic function should do as little work as possible.

An example of "reduced" function that has been extended internally to accept arrays with any number of additional dimensions can be found here.

Note on multiApply::Apply(): Apply differs from the base R apply function in that the latter does not allow to apply functions that operate on more than one multidimensional input array, nor provides a multi-core capability. However apply is more efficient in cases where a simple function is applied to a single array. You can find more information on Apply in its official GitLab repository: https://earth.bsc.es/gitlab/ces/multiApply.

10. Adaptation to the CSTools data structure via "wrapper" functions

CSTools is expected to operate with more complex data objects (with more dimensions and attributes) than the essential inputs required by the different atomic functions. This is why wrapper functions should be provided for each functionality, which:

a. are aware of the data structure established as standard for the s2dv_cube objects

b. deal with additional complexities (dimensions and attributes)

c. include calls to one or more (potentially "reduced") functions or other wrapper functions

Each developer would be responsible to write its own wrapper function(s) unless a generalized function (as shown in step 9.) was provided. In that case, the BSC would be in charge of developing the wrapper function.

This wrapper function should do an exhaustive check of the parameters unless already implemented in another function used inside the wrapper.

An example of a wrapper function for the example of "reduced" function shown before is provided next.

```
CST_WeightedAreaAverage <- function(medscope_exp_data,
                                    medscope_obs_data,
                                    ncores = 1) {
  # medscope_exp_data: Medscope experimental data object. A named list
  #                    with the following components:
  #   $data: numeric array with expected dimensions
  #       'dataset', 'member', 'sdate', 'ftime', 'lat', 'lon'
  #   $lon: numeric vector with expected dimensions
  #       'longitude'
  #   $lat: numeric vector with expected dimensions
  #       'latitude'
  #   [...other components...]
  # medscope_obs_data: similar to the previous parameter
  # ncores: number of cores to use for the computation

  medscope_exp_data$data <- Apply(medscope_exp_data$data,
                                  target = c('lat', 'lon'),
                                  RWeightedAreaAverage,
                                  lats = medscope_exp_data$lat,
                                  ncores = ncores)[[1]]

  medscope_obs_data$data <- Apply(medscope_obs_data$data,
                                  target = c('lat', 'lon'),
                                  RWeightedAreaAverage,
                                  lats = medscope_obs_data$lat,
                                  ncores = ncores)[[1]]

  medscope_exp_data$lon <- NULL
  medscope_obs_data$lon <- NULL
  medscope_exp_data$lat <- NULL
  medscope_obs_data$lat <- NULL

  list(exp = medscope_exp_data, obs = medscope_obs_data)
}
```

As can be seen in the example, the wrapper function is relying on a data structure for complex data objects combining multiple experimental or observational data sets, with multiple members, simulation start dates and forecast time steps, as well as other additional attributes.

Particularly, the data structure used and proposed here is a variation of the s2dverification's data structure (see below). This is due mainly to the fact that s2dverification includes a Load function to straightforwardly load, select and homogenize C3S data into the R workspace, and it returns data objects following the mentioned data structure. The function CST_Load has been developed and included in the CSTools package, which uses s2dverification's Load function to load the data and returns two s2dv_cubes, that comply with the data structure shown next.

```
A named list with the following components:
  $ data: numeric array with expected dimensions
        'dataset', 'member', 'sdate', 'ftime', 'lat', 'lon'
  $ lon: numeric vector with expected dimensions
        'longitude'
  $ lat: numeric vector with expected dimensions
        'latitude'
  $ Variable: List of 2
    ..$ varName: character string with the short name of the variable
    ..$ level: character string with the pressure level, or NULL
    ..- attr(*, 'units'): a character string with the units of the
                    variable
    ..- attr(*, 'longname'): a character string with the long name of the
                      variable
    [... other attributes ...]
  $ Dates: List of 2
  ..$ start: array of POSIXct dates with the dimensions ‘sdate’, ‘time’
          this array contains the initial date of each forecast time
          step for each start date (e.g. if the loaded data are
          monthly means centered on the 15th of each month, the
          initial dates would be the 1st of each month).
  ..$ end: array of POSIXct dates with the dimensions ‘sdate’, ‘time’
        this array contains the final date of each forecast time
        step for each start date (e.g. if the loaded data are
        monthly means centered on the 15th of each month, the
        final dates would be the 28st/30st/31st of each month)
[... other components ...]
```

A real example of a populated s2dverification data object can be found in Appendix 2.

11. Inclusion of plotting functions

Plotting functions should either be designed to receive multidimensional arrays with named dimensions as main parameters, plus additional parameters, or wrapped in a “wrapper” function that apply them to richer s2dv_cube objects. Optionally, if the function has been adapted to multidimensional arrays, a wrapper function can also be developed for a better user experience.

12. Inclusion of functions to be exposed to the user, not devoted to analysis or plotting

If a function that is not devoted to data analysis or plotting needs to be included and exposed in CSTools, it should be put in contrast with other similar functions in the package to make sure it is coherent with them, and finally be wrapped in a “wrapper” function. In case of doubt, contact the package administrator.

13. Inclusion of helper functions not to be exposed to the user

Helper functions that are not to be exposed to the users of CSTools, can be included either in the R/zzz.R file if used in many places, or in the .R file of the function where they are called if only used by that function.

14. Adding your contribution to the Git project

GitLab is the web platform for our Version Control System of the project. The structure of the project required for building an R package are: DESCRIPTION and NAMESPACES files and man and R folders (at the minimum). In the master branch, the functions and documentation which have been reviewed, tested and accepted for publication appear. **To add a new function, you will need to create a new branch,** in which you can have a record of the evolution of your work, **from the master branch**. **At the end of the process, your function along with the documentation and probably other modifications will be merged into the master branch**. During this process, especially at the beginning and at the end, your branch must be updated with the master branch. The following text explains how to use git commands to collaborate in the CSTools GitLab project.

Note: Avoid pushing big size files to the gitlab repository. Please, remember that GitLab is designed to keep track of code development and, to add data to the package, you need an agreement with the coordinator.

Git 0: Create an account on the GitLab portal, as described at the very end of the "Documentation and development policy" linked in the introduction of this document.

Git 1: The first time you want to contribute to CSTools, you will need to clone the GitLab repository to your computer. You should browse to a folder where to create a local copy of the repository. Then, execute the following line in your terminal:

```
git clone http://earth.bsc.es/gitlab/external/cstools.git
```

After typing your GitLab credentials, a new folder will be created in your local folder called 'cstools' which contains the copy of GitLab CSTools. You can browse it.

Git 2: The following command can be used at any time to check the branch and its status.

```
git status
```

The first time you run it after cloning the repo, its output will be:

```
/cstools> git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

Git 3: As it's on branch master, a new branch should be created to add the new desired function. The name of the branch must start by 'develop-' and be followed by a relevant word for the function which is going to be added (e.g. for a distribution function it can be 'develop-distribution'). Once the name is selected, use the following command:

```
git checkout -b develop-<name>
```

Automatically a message appears "Switched to a new branch 'develop-<name>'" indicating that the git is pointing to the new branch.

You can now add your function and make as many modifications you need to it and its documentation. At any moment, you can save modifications (even if they aren't final modifications) to keep track of them in the repository. For that, you will need the following commands:

Git 4: First, git needs to know when you have finished to do modifications. So, the following command will track the modifications by adding an index (non relevant for the user):

```
git add file1 file2 file3
```

Git 5: Then, a message can be added to these indexed modifications which are stored in a new snapshot of the project's state in the Git history, by creating a new commit representing the current state of the index.

```
git commit -m "Short message explaining the modification"
```

GIt 6: To save the local changes in the remote GitLab repository (your GitLab credentials may be requested).

```
git push origin HEAD
```

You can use steps Git 4 to Git 6 as many times as it would be necessary.

Git 7: Update your local repository:

```
git pull origin HEAD
```

The most relevant action of this command is that it has updated your ‘develop-<name>’ branch with file in master branch from the remote repository. **It must be used before creating a new branch and before merging the develop-<name> branch with master**. This is done in order to identify any incompatibilities that might arise from changes in the master brand with the branch you are developing (which might have started from a different version of the master branch). Note that it can be done at any time.

Git 8: To create a new branch (e.g. to add a different new function), you must move to the master branch by executing:

```
git checkout master
```

Then, repeat step Git 7, to confirm that the master branch is updated and start from step Git 3.

Another tool is the use of roxygen2 to simplify the process of creating a R package. This package helps to create the documentation in the proper format to pass the CRAN checks. For each function, a header including the definitions must be included in the .R file (you can consider following MultivarRMSE.R as an example of header, taking into account that more sections can be added to the header). In order to automatically create the documentation .Rd file which is in the folder man, open R, check that your working directory is cstools and run:

```
devtools::document()
```

This command would modify or create the .Rd file and the NAMESPACE file.
Note: check the DESCRIPTION file to see if your name appears in the authors section.

In the same R session, the documentation generated in the final format can be visualized running

```
?FunctionName
```

Note: the format will differ if you are using R or RStudio.

Once the documentation is created, it's possible to build and check the CSTools package including your new function. In your terminal, browse to the parent directory of cstools and run the following commands:

```
R CMD build cstools
R CMD check --as-cran CSTools_0.0.1.tar.gz
```

The documentation of the package must be corrected until any error, any warning and just one note (with information relative to the maintainer of the package) would be returned by the last command.

Note that further steps will be required before submitting the package to CRAN.

Adding checks to a function

In order to build a robust software, each function should contain a section for checking the arguments that a user could specify. For this reason, the first lines in the function should be function checks, as described in this section. It's recommended to previously write the functions documentation, as the definition of the parameters would simplify the formulation of checks (see examples below).

In the function() arguments, the order of the parameter should be sort by relevance: from essential parameters (such as data) to auxiliar or optional parameters (such as NA values behaviour).

- Example 1: logical parameter (from AnoMultiMetric.R)

#'@param **multimodel** a logical value indicating whether a Multi-Model Mean should be computed.

```
if (!is.logical(multimodel)) {
  stop("Parameter 'multimodel' must be a logical value.")
}
```

```
if (length(multimodel) > 1) {
  multimodel <- multimodel[1]
  warning("Parameter 'multimodel' has length > 1 and only the first",
          "element will be used.")
}
```

- Example 2: the user selects an option with this parameter (from AnoMultiMetric.R)

#'@param **metric** a character string giving the metric for computing the maximum skill. This must be one of the strings 'correlation', 'rms' or 'rmsss'.

```
if (length(metric) > 1) {
  metric <- metric[1]
  warning("Parameter 'multimodel' has length > 1 and only the first element will be used.")
}

if (metric == 'correlation') {
  corr <- Corr(AvgExp, AvgObs, posloop = 1, poscor = 2)
} else if (metric == 'rms') {
  corr <- RMS(AvgExp, AvgObs, posloop = 1, posRMS = 2)
} else if (metric == 'rmsss') {
  corr <- RMSSS(AvgExp, AvgObs, posloop = 1, posRMS = 2)
} else {
  stop("Parameter 'metric' must be a character string indicating",
       "one of the options: 'correlation', 'rms' or 'rmse'.")
}
```

- Example 3: auxiliar parameter (from MultivarRMSE.R). This parameter has been defined as NULL by default in the function() arguments definition.

#'@param **weight** (optional) a vector of weight values to assign to each variable. If no weights are defined, a value of 1 is assigned to every variable.

```
if (is.null(weight)) {
  weight <- c(rep(1, nvar))
} else if (length(weight) != nvar) {
  stop("Parameter 'weight' must have a length equal to the number of variables.")
}
```

- Example 4: data parameter as array with named dimensions.

Note: In this example, there are two optional arguments to the function which are the position of longitude and latitude dimensions in the input array.

#'@param **data** an array with minimum two dimensions of latitude and longitude.

```
if (is.null(data)) {
  stop("Parameters 'data' cannot be NULL.")
}
if (!is.numeric(data)) {
  stop("Parameters 'data' must be numeric.")
}
if (!is.array(data) && !is.matrix(data)) {
  stop("Parameter 'data' must be an array or matrix.")
}

dims <- 1:length(dim(data))

if (is.null(londim)) {
  if ("lon" %in% names(dim(data))) {
    londim <- which(names(dim(data)) == "lon")
  } else if (length(lon) %in% dim(data)) {
    londim <- which(dim(data) == length(lon))
    if (length(londim) > 1) {
      stop("More than one dimension of the parameter 'data' has ",
           "the same length as 'lon' parameter.")
    }
  } else {
    stop("None of the dimensions of the parameter 'data' are of ",
         "the same length as 'lon'.")
  }
}

if (is.null(latdim)) {
  if ("lat" %in% names(dim(data))) {
    latdim <- which(names(dim(data)) == "lat")
  } else if (length(lat) %in% dim(data)) {
    latdim <- which(dim(data) == length(lat))
    if (length(latdim) > 1) {
      stop("More than one dimension of the parameter 'data' has ",
           "the same length as 'lat' parameter.")
    }
  } else {
    stop("Non of the dimensions of the parameter 'data' are of ",
         "the same length as 'lat' parameter.")
  }
}

if (londim == latdim) {
    stop("Parameter 'londim' and 'latdim' cannot be equal.")
}

if (dim(data)[londim] != length(lon)) {
  stop("The longitudinal dimension of parameter 'data' must be of ",
       "the same length of parameter  'lon'.")
}
```

```
if (dim(data)[latdim] != length(lat)) {
  stop("The latitudinal dimension of parameter 'data' must be of ",
       "the same length of parameter  'lat'.")
}
```

Would your function work if parameters contain NA values? Should the function crash or return a warning?

Appendix 1

This appendix shows a complex folder structure.

```
/path/to/data/
   |--exp/
   | |--mfS5S5/
               | | |--daily_mean/
   | |    |--prlr/
   | |       |--prlr_19931101.nc
   | |       |--prlr_19941101.nc
   | |--ecmwfS5/
               |     |--monthly_mean/
   |    | |--tasmax/
   |    |    |--tasmax_19931101.nc
   |    |    |--tasmax_19941101.nc
               |     |--daily_mean/
   |       |--tasmax/
   |       | |--tasmax_19931101.nc
   |       | |--tasmax_19941101.nc
   |       |--prlr/
   |          |--prlr_19931101.nc
   |          |--prlr_19941101.nc
   |--recon/
```

```
        |--erainterim/
           |--daily_mean/
              |--tas/
              |  |--tas_199301.nc
              |  |--tas_199302.nc
              |  |--    ...
              |  |--tas_199412.nc
              |--tasmax/
                 |--tasmax_199301.nc
                 |--tasmax_199302.nc
                 |--    ...
                 |--tasmax_199412.nc
```

Appendix 2

This appendix shows the detailed output of the s2dverification::Load function when retrieving some example data.

```
data <- Load(var = 'tasmax',
             exp = list(ecmwf_s5_repos),
             obs = list(eraint_repos),
             nmember = NULL,
             sdates = sdates,
             leadtimemin = 1, leadtimemax = 3,
             lonmin = -20, lonmax = 50,
             latmin = 15, latmax = 70,
             output = 'lonlat',
             storefreq = 'daily')
* The load call you issued is:
*   Load(var = "tasmax", exp = list(structure(list(name = "ecmwfS5", path =
*       "/path/to/data/exp/$EXP_NAME$/$STORE_FREQ$_mean/$VAR_NAME$/$VAR_NAME$_$START_DATE$.nc"),
*       .Names = c("name", "path"))), obs = list(structure(list(name =
*       "erainterim", path =
*       "/path/to/data/recon/$OBS_NAME$/$STORE_FREQ$_mean/$VAR_NAME$/$VAR_NAME$_$YEAR$$MONTH$.
nc",
*       nc_var_name = "tas"), .Names = c("name", "path",
*       "nc_var_name"))), sdates = c("19931101", "19941101"), grid =
*       NULL, output = "lonlat", storefreq = "daily", ...)
* See the full call in '$load_parameters' after Load() finishes.
* Fetching first experimental files to work out 'var_exp' size...
* Exploring dimensions...
*   /path/to/data/exp/ecmwfS5/daily_mean/tasmax/tasmax_19931101.nc
* Success. Detected dimensions of experimental data: 1, 25, 2, 1, 181,
*   360
* Fetching first observational files to work out 'var_obs' size...
* Exploring dimensions...
*   /path/to/data/recon/erainterim/daily_mean/tasmax/tasmax_199311.nc
* Success. Detected dimensions of observational data: 1, 1, 2, 1, 181,
*   360
* Will now proceed to read and process 4 data files:
*   /path/to/data/exp/ecmwfS5/daily_mean/tasmax/tasmax_19931101.nc
*   /path/to/data/exp/ecmwfS5/daily_mean/tasmax/tasmax_19941101.nc
*   /path/to/data/recon/erainterim/daily_mean/tasmax/tasmax_199311.nc
*   /path/to/data/recon/erainterim/daily_mean/tasmax/tasmax_199411.nc
* Total size of requested data: 27106560 bytes.
*   - Experimental data: ( 1 x 25 x 2 x 1 x 181 x 360 ) x 8 bytes =
*   26064000 bytes.
*   - Observational data: ( 1 x 1 x 2 x 1 x 181 x 360 ) x 8 bytes =
*   1042560 bytes.
* If size of requested data is close to or above the free shared RAM
*   memory, R will crash.
starting worker pid=11764 on localhost:11801 at 17:33:05.259
starting worker pid=11780 on localhost:11801 at 17:33:05.444
starting worker pid=11796 on localhost:11801 at 17:33:05.631
starting worker pid=11812 on localhost:11801 at 17:33:05.818
starting worker pid=11828 on localhost:11801 at 17:33:05.993
starting worker pid=11844 on localhost:11801 at 17:33:06.172
starting worker pid=11860 on localhost:11801 at 17:33:06.344
starting worker pid=11876 on localhost:11801 at 17:33:06.513
* Loading... This may take several minutes...



str(data)
List of 11
 $ mod           : num [1, 1:25, 1:2, 1, 1:181, 1:360] 252 252 252 251 252 ...
  ..- attr(*, "dimensions")= chr [1:6] "dataset" "member" "sdate" "ftime" ...
 $ obs           : num [1, 1, 1:2, 1, 1:181, 1:360] 249 244 249 244 247 ...
  ..- attr(*, "dimensions")= chr [1:6] "dataset" "member" "sdate" "ftime" ...
 $ lon           : num [1:360(1d)] 0 1 2 3 4 5 6 7 8 9 ...
  ..- attr(*, "cdo_grid_name")= chr "r360x181"
  ..- attr(*, "data_across_gw")= logi TRUE
  ..- attr(*, "array_across_gw")= logi FALSE
  ..- attr(*, "first_lon")= num 0
  ..- attr(*, "last_lon")= num 359
  ..- attr(*, "projection")= chr "none"
 $ lat           : num [1:181(1d)] 90 89 88 87 86 85 84 83 82 81 ...
  ..- attr(*, "cdo_grid_name")= chr "r360x181"
  ..- attr(*, "first_lat")= num -90
  ..- attr(*, "last_lat")= num 90
  ..- attr(*, "projection")= chr "none"
 $ Variable      :List of 2
  ..$ varName: chr "tasmax"
  ..$ level  : NULL
  ..- attr(*, "use_dictionary")= logi FALSE
  ..- attr(*, "units")= chr "K"
  ..- attr(*, "longname")= chr "2 metre temperature"
  ..- attr(*, "description")= chr "none"
  ..- attr(*, "daily_agg_cellfun")= chr "none"
  ..- attr(*, "monthly_agg_cellfun")= chr "none"
  ..- attr(*, "verification_time")= chr "none"
 $ Datasets      :List of 2
  ..$ exp:List of 1
  .. ..$ ecmwfS5:List of 2
  .. .. ..$ InitializationDates:List of 25
  .. .. .. ..$ Member_1 : POSIXct[1:2], format: "1993-11-01" ...
  .. .. .. ..$ Member_2 : POSIXct[1:2], format: "1993-11-01" ...
  .. .. ..      ...
  .. .. .. ..$ Member_25: POSIXct[1:2], format: "1993-11-01" ...
  .. .. ..$ Members           : chr [1:25] "Member_1" "Member_2" "Member_3" "Member_4" ...
  .. .. ..- attr(*, "dataset")= chr "ecmwfS5"
  .. .. ..- attr(*, "source")= chr "/path/to/data/exp/ecmwfS5/daily_mean/tasmax"
  .. .. ..- attr(*, "URL")= chr "none"
  ..$ obs:List of 1
  .. ..$ erainterim:List of 2
  .. .. ..$ InitializationDates:List of 1
  .. .. .. ..$ Member_1: POSIXct[1:2], format: "1993-11-01" ...
  .. .. ..$ Members           : chr "Member_1"
  .. .. ..- attr(*, "dataset")= chr "erainterim"
  .. .. ..- attr(*, "source")= chr "/path/to/data/recon/erainterim/daily_mean/tasmax"
  .. .. ..- attr(*, "URL")= chr "none"
 $ Dates         :List of 2
  ..$ start: POSIXct[1:2], format: "1993-11-01" "1994-11-01"
  ..$ end  : POSIXct[1:2], format: "1993-11-02" "1994-11-02"
 $ when          : POSIXct[1:1], format: "2018-11-22 17:33:10"
 $ source_files   : chr [1:4] "/path/to/data/exp/ecmwfS5/daily_mean/tasmax/tasmax_19931101.nc"
"/path/to/data/exp/ecmwfS5/daily_mean/tasmax/tasmax_19941101.nc"
"/path/to/data/recon/erainterim/daily_mean/tasmax/tasmax_199311.nc"
"/path/to/data/recon/erainterim/daily_mean/tasmax/tasmax_199411.nc"
 $ not_found_files: NULL
 $ load_parameters:List of 28
  .. [ ... other components ... ]
```

Appendix 3

This appendix shows an example of loading 6-hourly data with the startR::Start function. More information can be found at https://earth.bsc.es/gitlab/es/startR/.

```
#devtools::install_git('https://earth.bsc.es/gitlab/es/startR')
library(startR)

repos <- '/esarchive/exp/ecmwf/system5_m1/6hourly/$var$/$var$_$sdate$.nc'

data <- Start(dat = repos,
              var = 'tas',
              sdate = c('19931101', '19941101'),
              ensemble = 'all',
              time = 'all',
              latitude = values(list(10, 120)),
              longitude = indices(1:40),
              retrieve = TRUE)

lons <- attr(data, 'Variables')$common$longitude
lats <- attr(data, 'Variables')$common$latitude
```