ESMValTool User's and Developer's Guide

Release 2.0.0

ESMValTool Development Team

Aug 05, 2020

ESMVALTOOL

Part I

Getting started

CHAPTER

ONE

INSTALLATION

1.1 Conda installation

In order to install the Conda package, you will need to install conda first. For a minimal conda installation go to https://conda.io/miniconda.html. It is recommended that you always use the latest version of conda, as problems have been reported when trying to use older versions.

Once you have installed conda, you can install ESMValCore by running:

```
conda install -c esmvalgroup -c conda-forge esmvalcore
```

1.2 Docker installation

ESMValCore is also provided through DockerHub in the form of docker containers. See https://docs.docker.com for more information about docker containers and how to run them.

You can get the latest release with

```
docker pull esmvalgroup/esmvalcore:stable
```

If you want to use the current master branch, use

```
docker pull esmvalgroup/esmvalcore:latest
```

To run a container using those images, use:

```
docker run esmvalgroup/esmvalcore:stable --help
```

Note that the container does not see the data or environmental variables available in the host by default. You can make data available with `-v /path:/path/in/container` and environmental variables with `-e VARNAME`.

For example, the following command would run a recipe

```
docker run -e HOME -v "$HOME":"$HOME" -v /data:/data esmvalgroup/esmvalcore:stable -c␣
↪~/config-user.yml ~/recipes/recipe_example.yml
```

with the environmental variable `$HOME` available inside the container and the data in the directories `$HOME` and `/data`, so these can be used to find the configuration file, recipe, and data.

It might be useful to define a bash alias or script to abbreviate the above command, for example

```
alias esmvaltool="docker run -e HOME -v $HOME:$HOME -v /data:/data esmvalgroup/
↪esmvalcore:stable"
```

would allow using the `esmvaltool` command without even noticing that the tool is running inside a Docker container.

1.3 Singularity installation

Docker is usually forbidden in clusters due to security reasons. However, there is a more secure alternative to run containers that is usually available on them: Singularity.

Singularity can use docker containers directly from DockerHub with the following command

```
singularity run docker://esmvalgroup/esmvalcore:stable -c ~/config-user.yml ~/recipes/
↪recipe_example.yml
```

Note that the container does not see the data available in the host by default. You can make host data available with `-B /path:/path/in/container`.

It might be useful to define a bash alias or script to abbreviate the above command, for example

```
alias esmvaltool="singularity run -B $HOME:$HOME -B /data:/data docker://esmvalgroup/
↪esmvalcore:stable"
```

would allow using the `esmvaltool` command without even noticing that the tool is running inside a Singularity container.

Some clusters may not allow to connect to external services, in those cases you can first create a singularity image locally:

```
singularity build esmvalcore.sif docker://esmvalgroup/esmvalcore:stable
```

and then upload the image file `esmvalcore.sif` to the cluster. To run the container using the image file `esmvalcore.sif` use:

```
singularity run esmvalcore.sif -c ~/config-user.yml ~/recipes/recipe_example.yml
```

1.4 Development installation

To install from source for development, follow these instructions.

- Download and install conda (this should be done even if the system in use already has a preinstalled version of conda, as problems have been reported with using older versions of conda)
- To make the `conda` command available, add `source <prefix>/etc/profile.d/conda.sh` to your `.bashrc` file and restart your shell. If using (t)csh shell, add `source <prefix>/etc/profile.d/conda.csh` to your `.cshrc`/`.tcshrc` file instead.
- Update conda: `conda update -y conda`
- Clone the ESMValCore Git repository: `git clone git@github.com:ESMValGroup/ESMValCore`
- Go to the source code directory: `cd ESMValCore`
- Create the esmvalcore conda environment `conda env create --name esmvalcore --file environment.yml`
- Activate the esmvalcore environment: `conda activate esmvalcore`

- Install in development mode: `pip install -e '.[develop]'`. If you are installing behind a proxy that does not trust the usual pip-urls you can declare them with the option `--trusted-host`, e.g. `pip install --trusted-host=pypi.python.org --trusted-host=pypi.org --trusted-host=files.pythonhosted.org -e .[develop]`
- Test that your installation was successful by running `esmvaltool -h`.

CHAPTER

TWO

CONFIGURATION FILES

2.1 Overview

There are several configuration files in ESMValCore:

- `config-user.yml`: sets a number of user-specific options like desired graphical output format, root paths to data, etc.;
- `config-developer.yml`: sets a number of standardized file-naming and paths to data formatting;
- `config-logging.yml`: stores information on logging.

and one configuration file which is distributed with ESMValTool:

- `config-references.yml`: stores information on diagnostic and recipe authors and scientific journals references;

2.2 User configuration file

The `config-user.yml` configuration file contains all the global level information needed by ESMValTool. It can be reused as many times the user needs to before changing any of the options stored in it. This file is essentially the gateway between the user and the machine-specific instructions to `esmvaltool`. By default, esmvaltool looks for it in the home directory, inside the `.esmvaltool` folder.

Users can get a copy of this file with default values by running

```
esmvaltool config get-config-user --path=${TARGET_FOLDER}
```

If the option `--path` is omitted, the file will be created in `${HOME}/.esmvaltool`

The following shows the default settings from the `config-user.yml` file with explanations in a commented line above each option:

```
# Diagnostics create plots? [true]/false
# turning it off will turn off graphical output from diagnostic
write_plots: true

# Diagnostics write NetCDF files? [true]/false
# turning it off will turn off netCDF output from diagnostic
write_netcdf: true

# Set the console log level debug, [info], warning, error
# for much more information printed to screen set log_level: debug
```

(continues on next page)

(continued from previous page)

```
log_level: info

# Exit on warning (only for NCL diagnostic scripts)? true/[false]
exit_on_warning: false

# Plot file format? [png]/pdf/ps/eps/epsi
output_file_type: png

# Destination directory where all output will be written
# including log files and performance stats
output_dir: ./esmvaltool_output

# Auxiliary data directory (used for some additional datasets)
# this is where e.g. files can be downloaded to by a download
# script embedded in the diagnostic
auxiliary_data_dir: ./auxiliary_data

# Use netCDF compression true/[false]
compress_netcdf: false

# Save intermediary cubes in the preprocessor true/[false]
# set to true will save the output cube from each preprocessing step
# these files are numbered according to the preprocessing order
save_intermediary_cubes: false

# Remove the preproc dir if all fine
# if this option is set to "true", ALL preprocessor files will be removed
# CAUTION when using: if you need those files, set it to false
remove_preproc_dir: true

# Run at most this many tasks in parallel [null]/1/2/3/4/..
# Set to null to use the number of available CPUs.
# If you run out of memory, try setting max_parallel_tasks to 1 and check the
# amount of memory you need for that by inspecting the file
# run/resource_usage.txt in the output directory. Using the number there you
# can increase the number of parallel tasks again to a reasonable number for
# the amount of memory available in your system.
max_parallel_tasks: null

# Path to custom config-developer file, to customise project configurations.
# See config-developer.yml for an example. Set to None to use the default
config_developer_file: null

# Get profiling information for diagnostics
# Only available for Python diagnostics
profile_diagnostic: false

# Rootpaths to the data from different projects (lists are also possible)
rootpath:
  CMIP5: [~/cmip5_inputpath1, ~/cmip5_inputpath2]
  OBS: ~/obs_inputpath
  default: ~/default_inputpath

# Directory structure for input data: [default]/BADC/DKRZ/ETHZ/etc
# See config-developer.yml for definitions.
drs:
  CMIP5: default
```

Most of these settings are fairly self-explanatory, e.g.:

```
# Diagnostics create plots? [true]/false
write_plots: true
# Diagnostics write NetCDF files? [true]/false
write_netcdf: true
```

The `write_plots` setting is used to inform ESMValTool diagnostics about your preference for creating figures. Similarly, the `write_netcdf` setting is a boolean which turns on or off the writing of netCDF files by the diagnostic scripts.

```
# Auxiliary data directory (used for some additional datasets)
auxiliary_data_dir: ~/auxiliary_data
```

The `auxiliary_data_dir` setting is the path to place any required additional auxiliary data files. This is necessary because certain Python toolkits, such as cartopy, will attempt to download data files at run time, typically geographic data files such as coastlines or land surface maps. This can fail if the machine does not have access to the wider internet. This location allows the user to specify where to find such files if they can not be downloaded at runtime.

Warning: This setting is not for model or observational datasets, rather it is for data files used in plotting such as coastline descriptions and so on.

A detailed explanation of the data finding-related sections of the `config-user.yml` (`rootpath` and `drs`) is presented in the *Data retrieval* section. This section relates directly to the data finding capabilities of ESMValTool and are very important to be understood by the user.

Note: You can choose your `config-user.yml` file at run time, so you could have several of them available with different purposes. One for a formalised run, another for debugging, etc. You can even provide any config user value as a run flag `--argument_name argument_value`

2.3 Developer configuration file

Most users and diagnostic developers will not need to change this file, but it may be useful to understand its content. It will be installed along with ESMValCore and can also be viewed on GitHub: esmvalcore/config-developer.yml. This configuration file describes the file system structure and CMOR tables for several key projects (CMIP6, CMIP5, obs4mips, OBS6, OBS) on several key machines (e.g. BADC, CP4CDS, DKRZ, ETHZ, SMHI, BSC). CMIP data is stored as part of the Earth System Grid Federation (ESGF) and the standards for file naming and paths to files are set out by CMOR and DRS. For a detailed description of these standards and their adoption in ESMValCore, we refer the user to *CMIP data - CMOR Data Reference Syntax (DRS) and the ESGF* section where we relate these standards to the data retrieval mechanism of the ESMValCore.

By default, esmvaltool looks for it in the home directory, inside the '.esmvaltool' folder.

Users can get a copy of this file with default values by running

```
``esmvaltool config get-config-developer --path=${TARGET_FOLDER}``.
```

If the option `--path` is omitted, the file will be created in *${HOME}/.esmvaltool*

Note: Remember to change your config-user file if you want to use a custom config-developer.

Example of the CMIP6 project configuration:

```
CMIP6:
  input_dir:
    default: '/'
    BADC: '{activity}/{institute}/{dataset}/{exp}/{ensemble}/{mip}/{short_name}/{grid}
↪/{latestversion}'
    DKRZ: '{activity}/{institute}/{dataset}/{exp}/{ensemble}/{mip}/{short_name}/{grid}
↪/{latestversion}'
    ETHZ: '{exp}/{mip}/{short_name}/{dataset}/{ensemble}/{grid}/'
  input_file: '{short_name}_{mip}_{dataset}_{exp}_{ensemble}_{grid}*.nc'
  output_file: '{project}_{dataset}_{mip}_{exp}_{ensemble}_{short_name}'
  cmor_type: 'CMIP6'
  cmor_strict: true
```

2.3.1 Input file paths

When looking for input files, the `esmvaltool` command provided by ESMValCore replaces the placeholders `[item]` in `input_dir` and `input_file` with the values supplied in the recipe. ESMValCore will try to automatically fill in the values for institute, frequency, and modeling_realm based on the information provided in the CMOR tables and/or `config-developer.yml` when reading the recipe. If this fails for some reason, these values can be provided in the recipe too.

The data directory structure of the CMIP projects is set up differently at each site. As an example, the CMIP6 directory path on BADC would be:

```
'{activity}/{institute}/{dataset}/{exp}/{ensemble}/{mip}/{short_name}/{grid}/
↪{latestversion}'
```

The resulting directory path would look something like this:

```
CMIP/MOHC/HadGEM3-GC31-LL/historical/r1i1p1f3/Omon/tos/gn/latest
```

For a more in-depth description of how to configure ESMValCore so it can find your data please see *CMIP data - CMOR Data Reference Syntax (DRS) and the ESGF*.

2.3.2 Preprocessor output files

The filename to use for preprocessed data is configured in a similar manner using `output_file`. Note that the extension `.nc` (and if applicable, a start and end time) will automatically be appended to the filename.

2.3.3 CMOR table configuration

ESMValCore comes bundled with several CMOR tables, which are stored in the directory esmvalcore/cmor/tables. These are copies of the tables available from PCMDI.

There are four settings related to CMOR tables available:

- `cmor_type`: can be `CMIP5` if the CMOR table is in the same format as the CMIP5 table or `CMIP6` if the table is in the same format as the CMIP6 table.
- `cmor_strict`: if this is set to `false`, the CMOR table will be extended with variables from the `esmvalcore/cmor/tables/custom` directory and it is possible to use variables with a `mip` which is different from the MIP table in which they are defined.

- `cmor_path`: path to the CMOR table. Defaults to the value provided in `cmor_type` written in lower case.
- `cmor_default_table_prefix`: defaults to the value provided in `cmor_type`.

2.4 References configuration file

The `config-references.yml` file contains the list of ESMValTool diagnostic and recipe authors, references and projects. Each author, project and reference referred to in the documentation section of a recipe needs to be in this file in the relevant section.

For instance, the recipe `recipe_ocean_example.yml` file contains the following documentation section:

```
documentation
  authors:
    - demo_le

  maintainer:
    - demo_le

  references:
    - demora2018gmd

  projects:
    - ukesm
```

These four items here are named people, references and projects listed in the `config-references.yml` file.

2.5 Logging configuration file

Warning: Section to be added

CHAPTER

THREE

FINDING DATA

3.1 Overview

Data discovery and retrieval is the first step in any evaluation process; ESMValTool uses a *semi-automated* data finding mechanism with inputs from both the user configuration file and the recipe file: this means that the user will have to provide the tool with a set of parameters related to the data needed and once these parameters have been provided, the tool will automatically find the right data. We will detail below the data finding and retrieval process and the input the user needs to specify, giving examples on how to use the data finding routine under different scenarios.

3.2 CMIP data - CMOR Data Reference Syntax (DRS) and the ESGF

CMIP data is widely available via the Earth System Grid Federation (ESGF) and is accessible to users either via download from the ESGF portal or through the ESGF data nodes hosted by large computing facilities (like CEDA-Jasmin, DKRZ, etc). This data adheres to, among other standards, the DRS and Controlled Vocabulary standard for naming files and structured paths; the DRS ensures that files and paths to them are named according to a standardized convention. Examples of this convention, also used by ESMValTool for file discovery and data retrieval, include:

- CMIP6 file: `[variable_short_name]_[mip]_[dataset_name]_[experiment]_[ensemble]_[grid]_[sta`
  `nc`
- CMIP5 file: `[variable_short_name]_[mip]_[dataset_name]_[experiment]_[ensemble]_[start-date`
  `nc`
- OBS file: `[project]_[dataset_name]_[type]_[version]_[mip]_[short_name]_[start-date]-[end-d`
  `nc`

Similar standards exist for the standard paths (input directories); for the ESGF data nodes, these paths differ slightly, for example:

- CMIP6 path for BADC: `ROOT-BADC/[institute]/[dataset_name]/[experiment]/[ensemble]/[mip]/ [variable_short_name]/[grid];`
- CMIP6 path for ETHZ: `ROOT-ETHZ/[experiment]/[mip]/[variable_short_name]/[dataset_name]/[ensemble]/[grid]`

From the ESMValTool user perspective the number of data input parameters is optimized to allow for ease of use. We detail this procedure in the next section.

3.3 Data retrieval

Data retrieval in ESMValTool has two main aspects from the user's point of view:

- data can be found by the tool, subject to availability on disk;
- it is the user's responsibility to set the correct data retrieval parameters;

The first point is self-explanatory: if the user runs the tool on a machine that has access to a data repository or multiple data repositories, then ESMValTool will look for and find the available data requested by the user.

The second point underlines the fact that the user has full control over what type and the amount of data is needed for the analyses. Setting the data retrieval parameters is explained below.

3.3.1 Setting the correct root paths

The first step towards providing ESMValTool the correct set of parameters for data retrieval is setting the root paths to the data. This is done in the user configuration file `config-user.yml`. The two sections where the user will set the paths are `rootpath` and `drs`. `rootpath` contains pointers to `CMIP`, `OBS`, `default` and `RAWOBS` root paths; `drs` sets the type of directory structure the root paths are structured by. It is important to first discuss the `drs` parameter: as we've seen in the previous section, the DRS as a standard is used for both file naming conventions and for directory structures.

3.3.2 Explaining `config-user/drs: CMIP5:` or `config-user/drs: CMIP6:`

Whereas ESMValTool will **always** use the CMOR standard for file naming (please refer above), by setting the `drs` parameter the user tells the tool what type of root paths they need the data from, e.g.:

```
drs:
  CMIP6: BADC
```

will tell the tool that the user needs data from a repository structured according to the BADC DRS structure, i.e.:

`ROOT/[institute]/[dataset_name]/[experiment]/[ensemble]/[mip]/[variable_short_name]/[grid];`

setting the `ROOT` parameter is explained below. This is a strictly-structured repository tree and if there are any sort of irregularities (e.g. there is no `[mip]` directory) the data will not be found! `BADC` can be replaced with `DKRZ` or `ETHZ` depending on the existing `ROOT` directory structure. The snippet

```
drs:
  CMIP6: default
```

is another way to retrieve data from a `ROOT` directory that has no DRS-like structure; `default` indicates that the data lies in a directory that contains all the files without any structure.

Note: When using `CMIP6: default` or `CMIP5: default` it is important to remember that all the needed files must be in the same top-level directory set by `default` (see below how to set `default`).

3.3.3 Explaining `config-user/rootpath:`

`rootpath` identifies the root directory for different data types (`ROOT` as we used it above):

- `CMIP` e.g. `CMIP5` or `CMIP6`: this is the *root* path(s) to where the CMIP files are stored; it can be a single path or a list of paths; it can point to an ESGF node or it can point to a user private repository. Example for a CMIP5 root path pointing to the ESGF node on CEDA-Jasmin (formerly known as BADC):

```
CMIP5: /badc/cmip5/data/cmip5/output1
```

  Example for a CMIP6 root path pointing to the ESGF node on CEDA-Jasmin:

```
CMIP6: /badc/cmip6/data/CMIP6/CMIP
```

  Example for a mix of CMIP6 root path pointing to the ESGF node on CEDA-Jasmin and a user-specific data repository for extra data:

```
CMIP6: [/badc/cmip6/data/CMIP6/CMIP, /home/users/johndoe/cmip_data]
```

- `OBS`: this is the *root* path(s) to where the observational datasets are stored; again, this could be a single path or a list of paths, just like for CMIP data. Example for the OBS path for a large cache of observation datasets on CEDA-Jasmin:

```
OBS: /group_workspaces/jasmin4/esmeval/obsdata-v2
```

- `default`: this is the *root* path(s) to where files are stored without any DRS-like directory structure; in a nutshell, this is a single directory that should contain all the files needed by the run, without any sub-directory structure.
- `RAWOBS`: this is the *root* path(s) to where the raw observational data files are stored; this is used by `cmorize_obs`.

3.3.4 Dataset definitions in `recipe`

Once the correct paths have been established, ESMValTool collects the information on the specific datasets that are needed for the analysis. This information, together with the CMOR convention for naming files (see *CMOR-DRS*) will allow the tool to search and find the right files. The specific datasets are listed in any recipe, under either the `datasets` and/or `additional_datasets` sections, e.g.

```
datasets:
  - {dataset: HadGEM2-CC,  project: CMIP5, exp: historical, ensemble: r1i1p1, start_
↪year: 2001, end_year: 2004}
  - {dataset: UKESM1-0-LL, project: CMIP6, exp: historical, ensemble: r1i1p1f2, grid:␣
↪gn, start_year: 2004,  end_year: 2014}
```

`_data_finder` will use this information to find data for **all** the variables specified in `diagnostics/variables`.

3.4 Recap and example

Let us look at a practical example for a recap of the information above: suppose you are using a `config-user.yml` that has the following entries for data finding:

```
rootpath:  # running on CEDA-Jasmin
  CMIP6: /badc/cmip6/data/CMIP6/CMIP
drs:
  CMIP6: BADC  # since you are on CEDA-Jasmin
```

and the dataset you need is specified in your `recipe.yml` as:

```
- {dataset: UKESM1-0-LL, project: CMIP6, mip: Amon, exp: historical, grid: gn,␣
↪ensemble: r1i1p1f2, start_year: 2004,  end_year: 2014}
```

for a variable, e.g.:

```
diagnostics:
  some_diagnostic:
    description: some_description
    variables:
      ta:
        preprocessor: some_preprocessor
```

The tool will then use the root path `/badc/cmip6/data/CMIP6/CMIP` and the dataset information and will assemble the full DRS path using information from *CMOR-DRS* and establish the path to the files as:

```
/badc/cmip6/data/CMIP6/CMIP/MOHC/UKESM1-0-LL/historical/r1i1p1f2/Amon
```

then look for variable `ta` and specifically the latest version of the data file:

```
/badc/cmip6/data/CMIP6/CMIP/MOHC/UKESM1-0-LL/historical/r1i1p1f2/Amon/ta/gn/latest/
```

and finally, using the file naming definition from *CMOR-DRS* find the file:

```
/badc/cmip6/data/CMIP6/CMIP/MOHC/UKESM1-0-LL/historical/r1i1p1f2/Amon/ta/gn/latest/ta_
↪Amon_UKESM1-0-LL_historical_r1i1p1f2_gn_195001-201412.nc
```

3.5 Observational data

Observational data is retrieved in the same manner as CMIP data, for example using the `OBS` root path set to:

```
OBS: /group_workspaces/jasmin4/esmeval/obsdata-v2
```

and the dataset:

```
- {dataset: ERA-Interim,  project: OBS,  type: reanaly,  version: 1,  start_
↪year: 2014,  end_year: 2015,  tier: 3}
```

in `recipe.yml` in `datasets` or `additional_datasets`, the rules set in *CMOR-DRS* are used again and the file will be automatically found:

```
/group_workspaces/jasmin4/esmeval/obsdata-v2/Tier3/ERA-Interim/OBS_ERA-Interim_
↪reanaly_1_Amon_ta_201401-201412.nc
```

Since observational data are organized in Tiers depending on their level of public availability, the `default` directory must be structured accordingly with sub-directories `TierX` (`Tier1`, `Tier2` or `Tier3`), even when `drs: default`.

3.6 Data loading

Data loading is done using the data load functionality of *iris*; we will not go into too much detail about this since we can point the user to the specific functionality here but we will underline that the initial loading is done by adhering to the CF Conventions that *iris* operates by as well (see CF Conventions Document and the search page for CF standard names).

3.7 Data concatenation from multiple sources

Oftentimes data retrieving results in assembling a continuous data stream from multiple files or even, multiple experiments. The internal mechanism through which the assembly is done is via cube concatenation. One peculiarity of iris concatenation (see iris cube concatenation) is that it doesn't allow for concatenating time-overlapping cubes; this case is rather frequent with data from models overlapping in time, and is accounted for by a function that performs a flexible concatenation between two cubes, depending on the particular setup:

- cubes overlap in time: resulting cube is made up of the overlapping data plus left and right hand sides on each side of the overlapping data; note that in the case of the cubes coming from different experiments the resulting concatenated cube will have composite data made up from multiple experiments: assume [cube1: exp1, cube2: exp2] and cube1 starts before cube2, and cube2 finishes after cube1, then the concatenated cube will be made up of cube2: exp2 plus the section of cube1: exp1 that contains data not provided in cube2: exp2;
- cubes don't overlap in time: data from the two cubes is bolted together;

Note that two cube concatenation is the base operation of an iterative process of reducing multiple cubes from multiple data segments via cube concatenation ie if there is no time-overlapping data, the cubes concatenation is performed in one step.

CHAPTER

FOUR

WORKING WITH THE INSTALLED RECIPES

Although ESMValTool can be used just to simplify the managment of data and the creation of your own analysis code, one of its main strenghts is the continuosly growing set of diagnostics and metrics that it directly provides to the user. These metrics and diagnostics are provided as a set of preconfigured recipes that users can run or customize for their own analysis. The latest list of available recipes can be found here.

In order to make the managmenent of these installed recipes easier, ESMValTool provides the `recipes` command group with utilities that help the users in discovering and customizing the provided recipes.

The first command in this group allows users to get the complete list of installed recipes printed to the console:

```
esmvaltool recipes list
```

If the user then wants to explore any one of this recipes, they can be printed using the following command

```
esmvaltool recipes show recipe_name.yml
```

And finally, to get a local copy that can then be cusotmized and run, users can use the following command

```
esmvaltool recipes get recipe_name.yml
```

CHAPTER

FIVE

RUNNING

The ESMValCore package provides the `esmvaltool` command line tool, which can be used to run a *recipe*.

To run a recipe, call `esmvaltool run` with the desired recipe:

```
esmvaltool run recipe_python.yml
```

If the config user is not in the default `{HOME}\.esmvaltool\` path you can pass its path explicitly:

```
esmvaltool run --config_file /path/to/config-user.yml recipe_python.yml
```

It is also possible to explicitly change values from the config file using flags:

```
esmvaltool run --argument_name argument_value recipe_python.yml
```

To get help on additional commands, please use

```
esmvaltool --help
```

Note: ESMValTool command line interface is created using the Fire python package. This package supports the creation of completion scripts for the Bash and Fish shells. Go to https://google.github.io/python-fire/using-cli/#python-fires-flags to learn how to set up them.

CHAPTER

SIX

OUTPUT

ESMValTool automatically generates a new output directory with every run. The location is determined by the output_dir option in the config-user.yml file, the recipe name, and the date and time, using the the format: YYYYMMDD_HHMMSS.

For instance, a typical output location would be: output_directory/recipe_ocean_amoc_20190118_1027/

This is effectively produced by the combination: output_dir/recipe_name_YYYYMMDD_HHMMSS/

This directory will contain 4 further subdirectories:

1. *Diagnostic output* (work): A place for any diagnostic script results that are not plots, e.g. files in NetCDF format (depends on the diagnostics).
2. *Plots*: The location for all the plots, split by individual diagnostics and fields.
3. *Run*: This directory includes all log files, a copy of the recipe, a summary of the resource usage, and the *settings.yml* interface files and temporary files created by the diagnostic scripts.
4. *Preprocessed datasets* (preproc): This directory contains all the preprocessed netcdfs data and the *metadata.yml* interface files. Note that by default this directory will be deleted after each run, because most users will only need the results from the diagnostic scripts.

6.1 Preprocessed datasets

The preprocessed datasets will be stored to the preproc/ directory. Each variable in each diagnostic will have its own the *metadata.yml* interface files saved in the preproc directory.

If the option `save_intermediary_cubes` is set to `true` in the config-user.yml file, then the intermediary cubes will also be saved here. This option is set to false in the default `config-user.yml` file.

If the option `remove_preproc_dir` is set to `true` in the config-user.yml file, then the preproc directory will be deleted after the run completes. This option is set to true in the default `config-user.yml` file.

6.2 Run

The log files in the run directory are automatically generated by ESMValTool and create a record of the output messages produced by ESMValTool and they are saved in the run directory. They can be helpful for debugging or monitoring the job, but also allow a record of the job output to screen after the job has been completed.

The run directory will also contain a copy of the recipe and the *settings.yml* file, described below. The run directory is also where the diagnostics are executed, and may also contain several temporary files while diagnostics are running.

6.3 Diagnostic output

The work/ directory will contain all files that are output at the diagnostic stage. Ie, the model data is preprocessed by ESMValTool and stored in the preproc/ directory. These files are opened by the diagnostic script, then some processing is applied. Once the diagnostic level processing has been applied, the results should be saved to the work directory.

6.4 Plots

The plots directory is where diagnostics save their output figures. These plots are saved in the format requested by the option *output_file_type* in the config-user.yml file.

6.5 Settings.yml

The settings.yml file is automatically generated by ESMValTool. Each diagnostic will produce a unique settings.yml file.

The settings.yml file passes several global level keys to diagnostic scripts. This includes several flags from the config-user.yml file (such as 'write_netcdf', 'write_plots', etc...), several paths which are specific to the diagnostic being run (such as 'plot_dir' and 'run_dir') and the location on disk of the metadata.yml file (described below).

```
input_files:[[...]recipe_ocean_bgc_20190118_134855/preproc/diag_timeseries_scalars/
↪mfo/metadata.yml]
log_level: debug
output_file_type: png
plot_dir: [...]recipe_ocean_bgc_20190118_134855/plots/diag_timeseries_scalars/Scalar_
↪timeseries
profile_diagnostic: false
recipe: recipe_ocean_bgc.yml
run_dir: [...]recipe_ocean_bgc_20190118_134855/run/diag_timeseries_scalars/Scalar_
↪timeseries
script: Scalar_timeseries
version: 2.0a1
work_dir: [...]recipe_ocean_bgc_20190118_134855/work/diag_timeseries_scalars/Scalar_
↪timeseries
write_netcdf: true
write_plots: true
```

The first item in the settings file will be a list of *Metadata.yml* files. There is a metadata.yml file generated for each field in each diagnostic.

6.6 Metadata.yml

The metadata.yml files is automatically generated by ESMValTool. Along with the settings.yml file, it passes all the paths, boolean flags, and additional arguments that your diagnostic needs to know in order to run.

The metadata is loaded from cfg as a dictionairy object in python diagnostics.

Here is an example metadata.yml file:

```
?
  [...]/recipe_ocean_bgc_20190118_134855/preproc/diag_timeseries_scalars/mfo/CMIP5_
↪HadGEM2-ES_Omon_historical_r1i1p1_TO0M_mfo_2002-2004.nc
  : cmor_table: CMIP5
  dataset: HadGEM2-ES
  diagnostic: diag_timeseries_scalars
  end_year: 2004
  ensemble: r1i1p1
  exp: historical
  field: TO0M
  filename: [...]recipe_ocean_bgc_20190118_134855/preproc/diag_timeseries_scalars/mfo/
↪CMIP5_HadGEM2-ES_Omon_historical_r1i1p1_TO0M_mfo_2002-2004.nc
  frequency: mon
  institute: [INPE, MOHC]
  long_name: Sea Water Transport
  mip: Omon
  modeling_realm: [ocean]
  preprocessor: prep_timeseries_scalar
  project: CMIP5
  recipe_dataset_index: 0
  short_name: mfo
  standard_name: sea_water_transport_across_line
  start_year: 2002
  units: kg s-1
  variable_group: mfo
```

As you can see, this is effectively a dictionary with several items including data paths, metadata and other information.

There are several tools available in python which are built to read and parse these files. The tools are avaialbe in the shared directory in the diagnostics directory.

Part II

The recipe format

CHAPTER

SEVEN

OVERVIEW

After `config-user.yml`, the `recipe.yml` is the second file the user needs to pass to `esmvaltool` as command line option, at each run time point. Recipes contain the data and data analysis information and instructions needed to run the diagnostic(s), as well as specific diagnostic-related instructions.

Broadly, recipes contain a general section summarizing the provenance and functionality of the diagnostics, the datasets which need to be run, the preprocessors that need to be applied, and the diagnostics which need to be run over the preprocessed data. This information is provided to ESMValTool in four main recipe sections: *Documentation*, *Datasets*, *Preprocessors* and *Diagnostics*, respectively.

7.1 Recipe section: `documentation`

The documentation section includes:

- The recipe's author's user name (`authors`, matching the definitions in the *References configuration file*)
- A description of the recipe (`description`, written in MarkDown format)
- A list of scientific references (`references`, matching the definitions in the *References configuration file*)
- the project or projects associated with the recipe (`projects`, matching the definitions in the *References configuration file*)

For example, the documentation section of `recipes/recipe_ocean_amoc.yml` is the following:

```
documentation:
  description: |
    Recipe to produce time series figures of the derived variable, the
    Atlantic meridional overturning circulation (AMOC).
    This recipe also produces transect figures of the stream functions for
    the years 2001-2004.

  authors:
    - demo_le

  maintainer:
    - demo_le

  references:
    - demora2018gmd

  projects:
    - ukesm
```

Note: Note that all authors, projects, and references mentioned in the description section of the recipe need to be included in the `config-references.yml` file. The author name uses the format: `surname_name`. For instance, John Doe would be: `doe_john`. This information can be omitted by new users whose name is not yet included in `config-references.yml`.

7.2 Recipe section: `datasets`

The `datasets` section includes dictionaries that, via key-value pairs, define standardized data specifications:

- dataset name (key `dataset`, value e.g. `MPI-ESM-LR` or `UKESM1-0-LL`)
- project (key `project`, value `CMIP5` or `CMIP6` for CMIP data, `OBS` for observational data, `ana4mips` for ana4mips data, `obs4mips` for obs4mips data, `EMAC` for EMAC data)
- experiment (key `exp`, value e.g. `historical`, `amip`, `piControl`, `RCP8.5`)
- mip (for CMIP data, key `mip`, value e.g. `Amon`, `Omon`, `LImon`)
- ensemble member (key `ensemble`, value e.g. `r1i1p1`, `r1i1p1f1`)
- time range (e.g. key-value `start_year: 1982`, `end_year: 1990`. Please note that yaml interprets numbers with a leading `0` as octal numbers, so we recommend to avoid them. For example, use `128` to specify the year 128 instead of `0128`.)
- model grid (native grid `grid: gn` or regridded grid `grid: gr`, for CMIP6 data only).

For example, a datasets section could be:

```
datasets:
  - {dataset: CanESM2, project: CMIP5, exp: historical, ensemble: r1i1p1, start_year:␣
↪2001, end_year: 2004}
  - {dataset: UKESM1-0-LL, project: CMIP6, exp: historical, ensemble: r1i1p1f2, start_
↪year: 2001, end_year: 2004, grid: gn}
  - {dataset: EC-EARTH3, alias: custom_alias, project: CMIP6, exp: historical,␣
↪ensemble: r1i1p1f1, start_year: 2001, end_year: 2004, grid: gn}
```

It is possible to define the experiment as a list to concatenate two experiments. Here it is an example concatenating the *historical* experiment with *rcp85*

```
datasets:
  - {dataset: CanESM2, project: CMIP5, exp: [historical, rcp85], ensemble: r1i1p1,␣
↪start_year: 2001, end_year: 2004}
```

It is also possible to define the ensemble as a list when the two experiments have different ensemble names. In this case, the specified datasets are concatenated into a single cube:

```
datasets:
  - {dataset: CanESM2, project: CMIP5, exp: [historical, rcp85], ensemble: [r1i1p1,␣
↪r1i2p1], start_year: 2001, end_year: 2004}
```

ESMValTool also supports a simplified syntax to add multiple ensemble members from the same dataset. In the ensemble key, any element in the form *(x:y)* will be replaced with all numbers from x to y (both inclusive), adding a dataset entry for each replacement. For example, to add ensemble members r1i1p1 to r10i1p1 you can use the following abbreviated syntax:

```
datasets:
  - {dataset: CanESM2, project: CMIP5, exp: historical, ensemble: "r(1:10)i1p1",␣
↪start_year: 2001, end_year: 2004}
```

It can be included multiple times in one definition. For example, to generate the datasets definitions for the ensemble members r1i1p1 to r5i1p1 and from r1i2p1 to r5i1p1 you can use:

```
datasets:
  - {dataset: CanESM2, project: CMIP5, exp: historical, ensemble: "r(1:5)i(1:2)p1",␣
↪start_year: 2001, end_year: 2004}
```

Please, bear in mind that this syntax can only be used in the ensemble tag. Also, note that the combination of multiple experiments and ensembles, like exp: [historical, rcp85], ensemble: [r1i1p1, "r(2:3)i1p1"] is not supported and will raise an error.

Note that this section is not required, as datasets can also be provided in the *Diagnostics* section.

7.3 Recipe section: `preprocessors`

The preprocessor section of the recipe includes one or more preprocessors, each of which may call the execution of one or several preprocessor functions.

Each preprocessor section includes:

- A preprocessor name (any name, under `preprocessors`);
- A list of preprocessor steps to be executed (choose from the API);
- Any or none arguments given to the preprocessor steps;
- The order that the preprocessor steps are applied can also be specified using the `custom_order` preprocessor function.

The following snippet is an example of a preprocessor named `prep_map` that contains multiple preprocessing steps (*Horizontal regridding* with two arguments, *Time manipulation* with no arguments (i.e., calculating the average over the time dimension) and *Multi-model statistics* with two arguments):

```
preprocessors:
  prep_map:
    regrid:
      target_grid: 1x1
      scheme: linear
    climate_statistics:
      operator: mean
    multi_model_statistics:
      span: overlap
      statistics: [mean]
```

Note: In this case no `preprocessors` section is needed the workflow will apply a `default` preprocessor consisting of only basic operations like: loading data, applying CMOR checks and fixes (*CMORization and dataset-specific fixes*) and saving the data to disk.

Preprocessor operations will be applied using the default order as listed in *Preprocessor functions*. Preprocessor tasks can be set to run in the order they are listed in the recipe by adding `custom_order: true` to the preprocessor definition.

7.4 Recipe section: `diagnostics`

The diagnostics section includes one or more diagnostics. Each diagnostic section will include:

- the variable(s) to preprocess, including the preprocessor to be applied to each variable;
- the diagnostic script(s) to be run;
- a description of the diagnostic and lists of themes and realms that it applies to;
- an optional `additional_datasets` section.

7.4.1 The diagnostics section defines tasks

The diagnostic section(s) define the tasks that will be executed when running the recipe. For each variable a preprocessing task will be defined and for each diagnostic script a diagnostic task will be defined. If variables need to be derived from other variables, a preprocessing task for each of the variables needed to derive that variable will be defined as well. These tasks can be viewed in the main_log_debug.txt file that is produced every run. Each task has a unique name that defines the subdirectory where the results of that task are stored. Task names start with the name of the diagnostic section followed by a '/' and then the name of the variable section for a preprocessing task or the name of the diagnostic script section for a diagnostic task.

A (simplified) example diagnostics section could look like

```
diagnostics:
  diagnostic_name:
    description: Air temperature tutorial diagnostic.
    themes:
      - phys
    realms:
      - atmos
    variables:
      variable_name:
        short_name: ta
        preprocessor: preprocessor_name
        mip: Amon
    scripts:
      script_name:
        script: examples/diagnostic.py
```

Note that the example recipe above contains a single diagnostic section called `diagnostic_name` and will result in two tasks:

- a preprocessing task called `diagnostic_name/variable_name` that will preprocess air temperature data for each dataset in the *Datasets* section of the recipe (not shown).
- a diagnostic task called `diagnostic_name/script_name`

The path to the script provided in the `script` option should be either the absolute path to the script, or the path relative to the `esmvaltool/diag_scripts` directory.

Depending on the installation configuration, you may get an error of "file does not exist" when the system tries to run the diagnostic script using relative paths. If this happens, use an absolute path instead.

7.4.2 Ancestor tasks

Some tasks require the result of other tasks to be ready before they can start, e.g. a diagnostic script needs the preprocessed variable data to start. Thus each tasks has zero or more ancestor tasks. By default, each diagnostic task in a diagnostic section has all variable preprocessing tasks in that same section as ancestors. However, this can be changed using the `ancestors` keyword. Note that wildcard expansion can be used to define ancestors.

```
diagnostics:
  diagnostic_1:
    variables:
      airtemp:
        short_name: ta
        preprocessor: preprocessor_name
        mip: Amon
    scripts:
      script_a:
        script: diagnostic_a.py
  diagnostic_2:
    variables:
      precip:
        short_name: pr
        preprocessor: preprocessor_name
        mip: Amon
    scripts:
      script_b:
        script: diagnostic_b.py
        ancestors: [diagnostic_1/script_a, precip]
```

The example recipe above will result in four tasks:

- a preprocessing task called `diagnostic_1/airtemp`
- a diagnostic task called `diagnostic_1/script_a`
- a preprocessing task called `diagnostic_2/precip`
- a diagnostic task called `diagnostic_2/script_b`

the preprocessing tasks do not have any ancestors, while the diagnostic_a.py script will receive the preprocessed air temperature data (has ancestor `diagnostic_1/airtemp`) and the diagnostic_b.py script will receive the results of diagnostic_a.py and the preprocessed precipitation data (has ancestors `diagnostic_1/script_a` and `diagnostic_2/precip`).

7.4.3 Task priority

Tasks are assigned a priority, with tasks appearing earlier on in the recipe getting higher priority. The tasks will be executed sequentially or in parellel, depending on the setting of `max_parallel_tasks` in the *User configuration file*. When there are fewer than `max_parallel_tasks` running, tasks will be started according to their priority. For obvious reasons, only tasks that are not waiting for ancestor tasks can be started. This feature makes it possible to reduce the processing time of recipes with many tasks, by placing tasks that take relatively long near the top of the recipe. Of course this only works when settings `max_parallel_tasks` to a value larger than 1. The current priority and run time of individual tasks can be seen in the log messages shown when running the tool (a lower number means higher priority).

7.4.4 Variable and dataset definitions

To define a variable/dataset combination that corresponds to an actual variable from a dataset, the keys in each variable section are combined with the keys of each dataset definition. If two versions of the same key are provided, then the key in the datasets section will take precedence over the keys in variables section. For many recipes it makes more sense to define the `start_year` and `end_year` items in the variable section, because the diagnostic script assumes that all the data has the same time range.

7.4.5 Diagnostic and variable specific datasets

The `additional_datasets` option can be used to add datasets beyond those listed in the *Datasets* section. This is useful if specific datasets need to be used only by a specific diagnostic or variable, i.e. it can be added both at diagnostic level, where it will apply to all variables in that diagnostic section or at individual variable level. For example, this can be a good way to add observational datasets, which are usually variable-specific.

7.4.6 Running a simple diagnostic

The following example, taken from `recipe_ocean_example.yml`, shows a diagnostic named *diag_map*, which loads the temperature at the ocean surface between the years 2001 and 2003 and then passes it to the `prep_map` preprocessor. The result of this process is then passed to the ocean diagnostic map script, `ocean/diagnostic_maps.py`.

```
diagnostics:

  diag_map:
    description: Global Ocean Surface regridded temperature map
    variables:
      tos: # Temperature at the ocean surface
        preprocessor: prep_map
        start_year: 2001
        end_year: 2003
    scripts:
      Global_Ocean_Surface_regrid_map:
        script: ocean/diagnostic_maps.py
```

7.4.7 Passing arguments to a diagnostic script

The diagnostic script section(s) may include custom arguments that can be used by the diagnostic script; these arguments are stored at runtime in a dictionary that is then made available to the diagnostic script via the interface link, independent of the language the diagnostic script is written in. Here is an example of such groups of arguments:

```
scripts:
  autoassess_strato_test_1: &autoassess_strato_test_1_settings
    script: autoassess/autoassess_area_base.py
    title: "Autoassess Stratosphere Diagnostic Metric MPI-MPI"
    area: stratosphere
    control_model: MPI-ESM-LR
    exp_model: MPI-ESM-MR
    obs_models: [ERA-Interim]  # list to hold models that are NOT for metrics but for␣
↪obs operations
    additional_metrics: [ERA-Interim, inmcm4]  # list to hold additional datasets for␣
↪metrics
```

In this example, apart from specifying the diagnostic script `script: autoassess/autoassess_area_base.py`, we pass a suite of parameters to be used by the script (`area`, `control_model` etc). These parameters are stored in key-value pairs in the diagnostic configuration file, an interface file that can be used by importing the `run_diagnostic` utility:

```
from esmvaltool.diag_scripts.shared import run_diagnostic

# write the diagnostic code here e.g.
def run_some_diagnostic(my_area, my_control_model, my_exp_model):
    """Diagnostic to be run."""
    if my_area == 'stratosphere':
        diag = my_control_model / my_exp_model
        return diag

def main(cfg):
    """Main diagnostic run function."""
    my_area = cfg['area']
    my_control_model = cfg['control_model']
    my_exp_model = cfg['exp_model']
    run_some_diagnostic(my_area, my_control_model, my_exp_model)

if __name__ == '__main__':

    with run_diagnostic() as config:
        main(config)
```

This way a lot of the optional arguments necessary to a diagnostic are at the user's control via the recipe.

7.4.8 Running your own diagnostic

If the user wants to test a newly-developed `my_first_diagnostic.py` which is not yet part of the ESMValTool diagnostics library, he/she do it by passing the absolute path to the diagnostic:

```
diagnostics:

  myFirstDiag:
    description: John Doe wrote a funny diagnostic
    variables:
      tos: # Temperature at the ocean surface
        preprocessor: prep_map
        start_year: 2001
        end_year: 2003
    scripts:
      JoeDiagFunny:
        script: /home/users/john_doe/esmvaltool_testing/my_first_diagnostic.py
```

This way the user may test a new diagnostic thoroughly before committing to the GitHub repository and including it in the ESMValTool diagnostics library.

7.4.9 Re-using parameters from one `script` to another

Due to `yaml` features it is possible to recycle entire diagnostics sections for use with other diagnostics. Here is an example:

```
scripts:
  cycle: &cycle_settings
    script: perfmetrics/main.ncl
    plot_type: cycle
    time_avg: monthlyclim
  grading: &grading_settings
    <<: *cycle_settings
    plot_type: cycle_latlon
    calc_grading: true
    normalization: [centered_median, none]
```

In this example the hook `&cycle_settings` can be used to pass the `cycle:` parameters to `grading:` via the shortcut `<<: *cycle_settings`.

CHAPTER

EIGHT

PREPROCESSOR

In this section, each of the preprocessor modules is described, roughly following the default order in which preprocessor functions are applied:

- *Variable derivation*
- *CMORization and dataset-specific fixes*
- *Vertical interpolation*
- *Weighting*
- *Land-sea masking*
- *Horizontal regridding*
- *Missing values masks*
- *Multi-model statistics*
- *Time manipulation*
- *Area manipulation*
- *Volume manipulation*
- *Cycles*
- *Detrend*
- *Unit conversion*
- *Other*

See *Preprocessor functions* for implementation details and the exact default order.

8.1 Overview

The ESMValTool preprocessor can be used to perform a broad range of operations on the input data before diagnostics or metrics are applied. The preprocessor performs these operations in a centralized, documented and efficient way, thus reducing the data processing load on the diagnostics side. For an overview of the preprocessor structure see the *Recipe section: preprocessors*.

Each of the preprocessor operations is written in a dedicated python module and all of them receive and return an Iris cube , working sequentially on the data with no interactions between them. The order in which the preprocessor operations is applied is set by default to minimize the loss of information due to, for example, temporal and spatial subsetting or multi-model averaging. Nevertheless, the user is free to change such order to address specific scientific requirements, but keeping in mind that some operations must be necessarily performed in a specific order. This is the

case, for instance, for multi-model statistics, which required the model to be on a common grid and therefore has to be called after the regridding module.

8.2 Variable derivation

The variable derivation module allows to derive variables which are not in the CMIP standard data request using standard variables as input. The typical use case of this operation is the evaluation of a variable which is only available in an observational dataset but not in the models. In this case a derivation function is provided by the ESMValTool in order to calculate the variable and perform the comparison. For example, several observational datasets deliver total column ozone as observed variable (*toz*), but CMIP models only provide the ozone 3D field. In this case, a derivation function is provided to vertically integrate the ozone and obtain total column ozone for direct comparison with the observations.

To contribute a new derived variable, it is also necessary to define a name for it and to provide the corresponding CMOR table. This is to guarantee the proper metadata definition is attached to the derived data. Such custom CMOR tables are collected as part of the ESMValCore package. By default, the variable derivation will be applied only if the variable is not already available in the input data, but the derivation can be forced by setting the appropriate flag.

```
variables:
  toz:
    derive: true
    force_derivation: false
```

The required arguments for this module are two boolean switches:

- `derive`: activate variable derivation
- `force_derivation`: force variable derivation even if the variable is directly available in the input data.

See also *`esmvalcore.preprocessor.derive()`*. To get an overview on derivation scripts and how to implement new ones, please go to *Variable derivation*.

8.3 CMORization and dataset-specific fixes

8.3.1 Data checking

Data preprocessed by ESMValTool is automatically checked against its cmor definition. To reduce the impact of this check while maintaining it as reliable as possible, it is split in two parts: one will check the metadata and will be done just after loading and concatenating the data and the other one will check the data itself and will be applied after all extracting operations are applied to reduce the amount of data to process.

Checks include, but are not limited to:

- Requested coordinates are present and comply with their definition.
- Correctness of variable names, units and other metadata.
- Compliance with the valid minimum and maximum values allowed if defined.

The most relevant (i.e. a missing coordinate) will raise an error while others (i.e an incorrect long name) will be reported as a warning.

Some of those issues will be fixed automatically by the tool, including the following:

- Incorrect standard or long names.
- Incorrect units, if they can be converted to the correct ones.

- Direction of coordinates.
- Automatic clipping of longitude to 0 - 360 interval.

8.3.2 Dataset specific fixes

Sometimes, the checker will detect errors that it can not fix by itself. ESMValTool deals with those issues by applying specific fixes for those datasets that require them. Fixes are applied at three different preprocessor steps:

- fix_file: apply fixes directly to a copy of the file. Copying the files is costly, so only errors that prevent Iris to load the file are fixed here. See *esmvalcore.preprocessor.fix_file()*
- fix_metadata: metadata fixes are done just before concatenating the cubes loaded from different files in the final one. Automatic metadata fixes are also applied at this step. See *esmvalcore.preprocessor.fix_metadata()*
- fix_data: data fixes are applied before starting any operation that will alter the data itself. Automatic data fixes are also applied at this step. See *esmvalcore.preprocessor.fix_data()*

To get an overview on data fixes and how to implement new ones, please go to *Dataset fixes*.

8.4 Vertical interpolation

Vertical level selection is an important aspect of data preprocessing since it allows the scientist to perform a number of metrics specific to certain levels (whether it be air pressure or depth, e.g. the Quasi-Biennial-Oscillation (QBO) u30 is computed at 30 hPa). Dataset native vertical grids may not come with the desired set of levels, so an interpolation operation will be needed to regrid the data vertically. ESMValTool can perform this vertical interpolation via the `extract_levels` preprocessor. Level extraction may be done in a number of ways.

Level extraction can be done at specific values passed to `extract_levels` as `levels:` with its value a list of levels (note that the units are CMOR-standard, Pascals (Pa)):

```
preprocessors:
  preproc_select_levels_from_list:
    extract_levels:
      levels: [100000., 50000., 3000., 1000.]
      scheme: linear
```

It is also possible to extract the CMIP-specific, CMOR levels as they appear in the CMOR table, e.g. `plev10` or `plev17` or `plev19` etc:

```
preprocessors:
  preproc_select_levels_from_cmip_table:
    extract_levels:
      levels: {cmor_table: CMIP6, coordinate: plev10}
      scheme: nearest
```

Of good use is also the level extraction with values specific to a certain dataset, without the user actually polling the dataset of interest to find out the specific levels: e.g. in the example below we offer two alternatives to extract the levels and vertically regrid onto the vertical levels of `ERA-Interim`:

```
preprocessors:
  preproc_select_levels_from_dataset:
    extract_levels:
      levels: ERA-Interim
```

(continues on next page)

(continued from previous page)

```
      # This also works, but allows specifying the pressure coordinate name
      # levels: {dataset: ERA-Interim, coordinate: air_pressure}
      scheme: linear_horizontal_extrapolate_vertical
```

By default, vertical interpolation is performed in the dimension coordinate of the z axis. If you want to explicitly declare the z axis coordinate to use (for example, `air_pressure`' in variables that are provided in model levels and not pressure levels) you can override that automatic choice by providing the name of the desired coordinate:

```
preprocessors:
  preproc_select_levels_from_dataset:
    extract_levels:
      levels: ERA-Interim
      scheme: linear_horizontal_extrapolate_vertical
      coordinate: air_pressure
```

- See also *esmvalcore.preprocessor.extract_levels()*.
- See also `esmvalcore.preprocessor.get_cmor_levels()`.

Note: For both vertical and horizontal regridding one can control the extrapolation mode when defining the interpolation scheme. Controlling the extrapolation mode allows us to avoid situations where extrapolating values makes little physical sense (e.g. extrapolating beyond the last data point). The extrapolation mode is controlled by the *extrapolation_mode* keyword. For the available interpolation schemes available in Iris, the extrapolation_mode keyword must be one of:

- `extrapolate`: the extrapolation points will be calculated by extending the gradient of the closest two points;
- `error`: a `ValueError` exception will be raised, notifying an attempt to extrapolate;
- `nan`: the extrapolation points will be be set to NaN;
- `mask`: the extrapolation points will always be masked, even if the source data is not a `MaskedArray`; or
- `nanmask`: if the source data is a MaskedArray the extrapolation points will be masked, otherwise they will be set to NaN.

8.5 Weighting

8.5.1 Land/sea fraction weighting

This preprocessor allows weighting of data by land or sea fractions. In other words, this function multiplies the given input field by a fraction in the range 0-1 to account for the fact that not all grid points are completely land- or sea-covered.

The application of this preprocessor is very important for most carbon cycle variables (and other land surface outputs), which are e.g. reported in units of $kgC\ m^{-2}$. Here, the surface unit actually refers to 'square meter of land/sea' and NOT 'square meter of gridbox'. In order to integrate these globally or regionally one has to weight by both the surface quantity and the land/sea fraction.

For example, to weight an input field with the land fraction, the following preprocessor can be used:

```
preprocessors:
  preproc_weighting:
    weighting_landsea_fraction:
```

(continues on next page)

(continued from previous page)

```
      area_type: land
      exclude: ['CanESM2', 'reference_dataset']
```

Allowed arguments for the keyword `area_type` are `land` (fraction is 1 for grid cells with only land surface, 0 for grid cells with only sea surface and values in between 0 and 1 for coastal regions) and `sea` (1 for sea, 0 for land, in between for coastal regions). The optional argument `exclude` allows to exclude specific datasets from this preprocessor, which is for example useful for climate models which do not offer land/sea fraction files. This arguments also accepts the special dataset specifiers `reference_dataset` and `alternative_dataset`.

Optionally you can specify your own custom fx variable to be used in cases when e.g. a certain experiment is preferred for fx data retrieval:

```
preprocessors:
  preproc_weighting:
    weighting_landsea_fraction:
      area_type: land
      exclude: ['CanESM2', 'reference_dataset']
      fx_variables: [{'short_name': 'sftlf', 'exp': 'piControl'}, {'short_name':
↪'sftof', 'exp': 'piControl'}]
```

See also *esmvalcore.preprocessor.weighting_landsea_fraction()*.

8.6 Masking

8.6.1 Introduction to masking

Certain metrics and diagnostics need to be computed and performed on specific domains on the globe. The ESMValTool preprocessor supports filtering the input data on continents, oceans/seas and ice. This is achieved by masking the model data and keeping only the values associated with grid points that correspond to, e.g., land, ocean or ice surfaces, as specified by the user. Where possible, the masking is realized using the standard mask files provided together with the model data as part of the CMIP data request (the so-called fx variable). In the absence of these files, the Natural Earth masks are used: although these are not model-specific, they represent a good approximation since they have a much higher resolution than most of the models and they are regularly updated with changing geographical features.

8.6.2 Land-sea masking

In ESMValTool, land-sea-ice masking can be done in two places: in the preprocessor, to apply a mask on the data before any subsequent preprocessing step and before running the diagnostic, or in the diagnostic scripts themselves. We present both these implementations below.

To mask out a certain domain (e.g., sea) in the preprocessor, `mask_landsea` can be used:

```
preprocessors:
  preproc_mask:
    mask_landsea:
      mask_out: sea
```

and requires only one argument: `mask_out`: either `land` or `sea`.

The preprocessor automatically retrieves the corresponding mask (`fx: stfof` in this case) and applies it so that sea-covered grid cells are set to missing. Conversely, it retrieves the `fx: sftlf` mask when land needs to be masked out, respectively.

Optionally you can specify your own custom fx variable to be used in cases when e.g. a certain experiment is preferred for fx data retrieval:

```
preprocessors:
  landmask:
    mask_landsea:
      mask_out: sea
      fx_variables: [{'short_name': 'sftlf', 'exp': 'piControl'}, {'short_name':
↪'sftof', 'exp': 'piControl'}]
```

If the corresponding fx file is not found (which is the case for some models and almost all observational datasets), the preprocessor attempts to mask the data using Natural Earth mask files (that are vectorized rasters). As mentioned above, the spatial resolution of the the Natural Earth masks are much higher than any typical global model (10m for land and glaciated areas and 50m for ocean masks).

See also *esmvalcore.preprocessor.mask_landsea()*.

8.6.3 Ice masking

Note that for masking out ice sheets, the preprocessor uses a different function, to ensure that both land and sea or ice can be masked out without losing generality. To mask ice out, `mask_landseaice` can be used:

```
preprocessors:
  preproc_mask:
    mask_landseaice:
      mask_out: ice
```

and requires only one argument: `mask_out`: either `landsea` or `ice`.

As in the case of `mask_landsea`, the preprocessor automatically retrieves the `fx_variables: [sftgif]` mask.

Optionally you can specify your own custom fx variable to be used in cases when e.g. a certain experiment is preferred for fx data retrieval:

```
preprocessors:
  landseaicemask:
    mask_landseaice:
      mask_out: sea
      fx_variables: [{'short_name': 'sftgif', 'exp': 'piControl'}]
```

See also *esmvalcore.preprocessor.mask_landseaice()*.

8.6.4 Glaciated masking

For masking out glaciated areas a Natural Earth shapefile is used. To mask glaciated areas out, `mask_glaciated` can be used:

```
preprocessors:
  preproc_mask:
    mask_glaciated:
      mask_out: glaciated
```

and it requires only one argument: `mask_out`: only `glaciated`.

See also *esmvalcore.preprocessor.mask_landseaice()*.

8.6.5 Missing values masks

Missing (masked) values can be a nuisance especially when dealing with multimodel ensembles and having to compute multimodel statistics; different numbers of missing data from dataset to dataset may introduce biases and artificially assign more weight to the datasets that have less missing data. This is handled in ESMValTool via the missing values masks: two types of such masks are available, one for the multimodel case and another for the single model case.

The multimodel missing values mask (`mask_fillvalues`) is a preprocessor step that usually comes after all the single-model steps (regridding, area selection etc) have been performed; in a nutshell, it combines missing values masks from individual models into a multimodel missing values mask; the individual model masks are built according to common criteria: the user chooses a time window in which missing data points are counted, and if the number of missing data points relative to the number of total data points in a window is less than a chosen fractional threshold, the window is discarded i.e. all the points in the window are masked (set to missing).

```
preprocessors:
  missing_values_preprocessor:
    mask_fillvalues:
      threshold_fraction: 0.95
      min_value: 19.0
      time_window: 10.0
```

In the example above, the fractional threshold for missing data vs. total data is set to 95% and the time window is set to 10.0 (units of the time coordinate units). Optionally, a minimum value threshold can be applied, in this case it is set to 19.0 (in units of the variable units).

See also *esmvalcore.preprocessor.mask_fillvalues()*.

8.6.6 Common mask for multiple models

It is possible to use `mask_fillvalues` to create a combined multimodel mask (all the masks from all the analyzed models combined into a single mask); for that purpose setting the `threshold_fraction` to 0 will not discard any time windows, essentially keeping the original model masks and combining them into a single mask; here is an example:

```
preprocessors:
  missing_values_preprocessor:
    mask_fillvalues:
      threshold_fraction: 0.0      # keep all missing values
      min_value: -1e20             # small enough not to alter the data
      #  time_window: 10.0         # this will not matter anymore
```

8.6.7 Minimum, maximum and interval masking

Thresholding on minimum and maximum accepted data values can also be performed: masks are constructed based on the results of thresholding; inside and outside interval thresholding and masking can also be performed. These functions are `mask_above_threshold`, `mask_below_threshold`, `mask_inside_range`, and `mask_outside_range`.

These functions always take a cube as first argument and either `threshold` for threshold masking or the pair `minimum`, ``maximum` for interval masking.

See also *esmvalcore.preprocessor.mask_above_threshold()* and related functions.

8.7 Horizontal regridding

Regridding is necessary when various datasets are available on a variety of *lat-lon* grids and they need to be brought together on a common grid (for various statistical operations e.g. multimodel statistics or for e.g. direct inter-comparison or comparison with observational datasets). Regridding is conceptually a very similar process to interpolation (in fact, the regridder engine uses interpolation and extrapolation, with various schemes). The primary difference is that interpolation is based on sample data points, while regridding is based on the horizontal grid of another cube (the reference grid).

The underlying regridding mechanism in ESMValTool uses the cube.regrid() from Iris.

The use of the horizontal regridding functionality is flexible depending on what type of reference grid and what interpolation scheme is preferred. Below we show a few examples.

8.7.1 Regridding on a reference dataset grid

The example below shows how to regrid on the reference dataset `ERA-Interim` (observational data, but just as well CMIP, obs4mips, or ana4mips datasets can be used); in this case the *scheme* is *linear*.

```
preprocessors:
  regrid_preprocessor:
    regrid:
      target_grid: ERA-Interim
      scheme: linear
```

8.7.2 Regridding on an `MxN` grid specification

The example below shows how to regrid on a reference grid with a cell specification of `2.5x2.5` degrees. This is similar to regridding on reference datasets, but in the previous case the reference dataset grid cell specifications are not necessarily known a priori. Regridding on an `MxN` cell specification is oftentimes used when operating on localized data.

```
preprocessors:
  regrid_preprocessor:
    regrid:
      target_grid: 2.5x2.5
      scheme: nearest
```

In this case the `NearestNeighbour` interpolation scheme is used (see below for scheme definitions).

When using a `MxN` type of grid it is possible to offset the grid cell centrepoints using the *lat_offset* and `lon_offset` arguments:

- `lat_offset`: offsets the grid centers of the latitude coordinate w.r.t. the pole by half a grid step;
- `lon_offset`: offsets the grid centers of the longitude coordinate w.r.t. Greenwich meridian by half a grid step.

```
preprocessors:
  regrid_preprocessor:
    regrid:
      target_grid: 2.5x2.5
      lon_offset: True
      lat_offset: True
      scheme: nearest
```

8.7.3 Regridding (interpolation, extrapolation) schemes

The schemes used for the interpolation and extrapolation operations needed by the horizontal regridding functionality directly map to their corresponding implementations in Iris:

- `linear`: Linear(extrapolation_mode='mask').
- `linear_extrapolate`: Linear(extrapolation_mode='extrapolate').
- `nearest`: Nearest(extrapolation_mode='mask').
- `area_weighted`: AreaWeighted().
- `unstructured_nearest`: UnstructuredNearest().

See also *esmvalcore.preprocessor.regrid()*

Note: For both vertical and horizontal regridding one can control the extrapolation mode when defining the interpolation scheme. Controlling the extrapolation mode allows us to avoid situations where extrapolating values makes little physical sense (e.g. extrapolating beyond the last data point). The extrapolation mode is controlled by the *extrapolation_mode* keyword. For the available interpolation schemes available in Iris, the extrapolation_mode keyword must be one of:

- `extrapolate` – the extrapolation points will be calculated by extending the gradient of the closest two points;
- `error` – a `ValueError` exception will be raised, notifying an attempt to extrapolate;
- `nan` – the extrapolation points will be be set to NaN;
- `mask` – the extrapolation points will always be masked, even if the source data is not a `MaskedArray`; or
- `nanmask` – if the source data is a MaskedArray the extrapolation points will be masked, otherwise they will be set to NaN.

Note: The regridding mechanism is (at the moment) done with fully realized data in memory, so depending on how fine the target grid is, it may use a rather large amount of memory. Empirically target grids of up to `0.5x0.5` degrees should not produce any memory-related issues, but be advised that for resolutions of `< 0.5` degrees the regridding becomes very slow and will use a lot of memory.

8.8 Multi-model statistics

Computing multi-model statistics is an integral part of model analysis and evaluation: individual models display a variety of biases depending on model set-up, initial conditions, forcings and implementation; comparing model data to observational data, these biases have a significantly lower statistical impact when using a multi-model ensemble. ESMValTool has the capability of computing a number of multi-model statistical measures: using the preprocessor module `multi_model_statistics` will enable the user to ask for either a multi-model `mean`, `median`, `max`, `min`, `std`, and / or `pXX.YY` with a set of argument parameters passed to `multi_model_statistics`. Percentiles can be specified like `p1.5` or `p95`. The decimal point will be replaced by a dash in the output file.

Note that current multimodel statistics in ESMValTool are local (not global), and are computed along the time axis. As such, can be computed across a common overlap in time (by specifying `span: overlap` argument) or across the full length in time of each model (by specifying `span: full` argument).

Restrictive computation is also available by excluding any set of models that the user will not want to include in the statistics (by setting `exclude: [excluded models list]` argument). The implementation has a few restrictions that apply to the input data: model datasets must have consistent shapes, and from a statistical point of

view, this is needed since weights are not yet implemented; also higher dimensional data is not supported (i.e. anything with dimensionality higher than four: time, vertical axis, two horizontal axes).

```
preprocessors:
  multimodel_preprocessor:
    multi_model_statistics:
      span: overlap
      statistics: [mean, median]
      exclude: [NCEP]
```

see also *esmvalcore.preprocessor.multi_model_statistics()*.

When calling the module inside diagnostic scripts, the input must be given as a list of cubes. The output will be saved in a dictionary where each entry contains the resulting cube with the requested statistic operations.

```
from esmvalcore.preprocessor import multi_model_statistics
statistics = multi_model_statistics([cube1,...,cubeN], 'overlap', ['mean', 'median'])
mean_cube = statistics['mean']
median_cube = statistics['median']
```

Note: Note that the multimodel array operations, albeit performed in per-time/per-horizontal level loops to save memory, could, however, be rather memory-intensive (since they are not performed lazily as yet). The Section on *Information on maximum memory required* details the memory intake for different run scenarios, but as a thumb rule, for the multimodel preprocessor, the expected maximum memory intake could be approximated as the number of datasets multiplied by the average size in memory for one dataset.

8.9 Time manipulation

The `_time.py` module contains the following preprocessor functions:

- *extract_time*: Extract a time range from a cube.
- *extract_season*: Extract only the times that occur within a specific season.
- *extract_month*: Extract only the times that occur within a specific month.
- *daily_statistics*: Compute statistics for each day
- *monthly_statistics*: Compute statistics for each month
- *seasonal_statistics*: Compute statistics for each season
- *annual_statistics*: Compute statistics for each year
- *decadal_statistics*: Compute statistics for each decade
- *climate_statistics*: Compute statistics for the full period
- *anomalies*: Compute (standardized) anomalies
- *regrid_time*: Aligns the time axis of each dataset to have common time points and calendars.
- *timeseries_filter*: Allows application of a filter to the time-series data.

Statistics functions are applied by default in the order they appear in the list. For example, the following example applied to hourly data will retrieve the minimum values for the full period (by season) of the monthly mean of the daily maximum of any given variable.

```
daily_statistics:
  operator: max

monthly_statistics:
  operator: mean

climate_statistics:
  operator: min
  period: season
```

8.9.1 extract_time

This function subsets a dataset between two points in times. It removes all times in the dataset before the first time and after the last time point. The required arguments are relatively self explanatory:

- start_year
- start_month
- start_day
- end_year
- end_month
- end_day

These start and end points are set using the datasets native calendar. All six arguments should be given as integers - the named month string will not be accepted.

See also *esmvalcore.preprocessor.extract_time()*.

8.9.2 extract_season

Extract only the times that occur within a specific season.

This function only has one argument: season. This is the named season to extract. ie: DJF, MAM, JJA, SON.

Note that this function does not change the time resolution. If your original data is in monthly time resolution, then this function will return three monthly datapoints per year.

If you want the seasonal average, then this function needs to be combined with the seasonal_mean function, below.

See also *esmvalcore.preprocessor.extract_season()*.

8.9.3 extract_month

The function extracts the times that occur within a specific month. This function only has one argument: month. This value should be an integer between 1 and 12 as the named month string will not be accepted.

See also *esmvalcore.preprocessor.extract_month()*.

8.9.4 daily_statistics

This function produces statistics for each day in the dataset.

Parameters:

- operator: operation to apply. Accepted values are 'mean', 'median', 'std_dev', 'min', 'max' and 'sum'. Default is 'mean'

See also *esmvalcore.preprocessor.daily_statistics()*.

8.9.5 monthly_statistics

This function produces statistics for each month in the dataset.

Parameters:

- operator: operation to apply. Accepted values are 'mean', 'median', 'std_dev', 'min', 'max' and 'sum'. Default is 'mean'

See also *esmvalcore.preprocessor.monthly_statistics()*.

8.9.6 seasonal_statistics

This function produces statistics for each season (DJF, MAM, JJA, SON) in the dataset. Note that this function will not check for missing time points. For instance, if you are looking at the DJF field, but your datasets starts on January 1st, the first DJF field will only contain data from January and February.

We recommend using the extract_time to start the dataset from the following December and remove such biased initial datapoints.

Parameters:

- operator: operation to apply. Accepted values are 'mean', 'median', 'std_dev', 'min', 'max' and 'sum'. Default is 'mean'

See also `esmvalcore.preprocessor.seasonal_mean()`.

8.9.7 annual_statistics

This function produces statistics for each year.

Parameters:

- operator: operation to apply. Accepted values are 'mean', 'median', 'std_dev', 'min', 'max' and 'sum'. Default is 'mean'

See also *esmvalcore.preprocessor.annual_statistics()*.

8.9.8 decadal_statistics

This function produces statistics for each decade.

Parameters:

- operator: operation to apply. Accepted values are 'mean', 'median', 'std_dev', 'min', 'max' and 'sum'. Default is 'mean'

See also *esmvalcore.preprocessor.decadal_statistics()*.

8.9.9 climate_statistics

This function produces statistics for the whole dataset. It can produce scalars (if the full period is chosen) or daily, monthly or seasonal statics.

Parameters:

- operator: operation to apply. Accepted values are 'mean', 'median', 'std_dev', 'min', 'max' and 'sum'. Default is 'mean'
- period: define the granularity of the statistics: get values for the full period, for each month or day of year. Available periods: 'full', 'season', 'seasonal', 'monthly', 'month', 'mon', 'daily', 'day'. Default is 'full'

Examples:

- Monthly climatology:

```
climate_statistics:
    operator: mean
    period: month
```

- Daily maximum for the full period:

```
climate_statistics:
  operator: max
  period: day
```

- Minimum value in the period:

```
climate_statistics:
  operator: min
  period: full
```

See also *esmvalcore.preprocessor.climate_statistics()*.

8.9.10 anomalies

This function computes the anomalies for the whole dataset. It can compute anomalies from the full, seasonal, monthly and daily climatologies. Optionally standardized anomalies can be calculated.

Parameters:

- period: define the granularity of the climatology to use: full period, seasonal, monthly or daily. Available periods: 'full', 'season', 'seasonal', 'monthly', 'month', 'mon', 'daily', 'day'. Default is 'full'
- reference: Time slice to use as the reference to compute the climatology on. Can be 'null' to use the full cube or a dictionary with the parameters from *extract_time*. Default is null
- standardize: if true calculate standardized anomalies (default: false)

Examples:

- Anomalies from the full period climatology:

```
anomalies:
```

- Anomalies from the full period monthly climatology:

```
anomalies:
  period: month
```

- Standardized anomalies from the full period climatology:

```
anomalies:
  standardized: true
```

- Standardized Anomalies from the 1979-2000 monthly climatology:

```
anomalies:
  period: month
  reference:
    start_year: 1979
    start_month: 1
    start_day: 1
    end_year: 2000
    end_month: 12
    end_day: 31
  standardize: true
```

See also *esmvalcore.preprocessor.anomalies()*.

8.9.11 regrid_time

This function aligns the time points of each component dataset so that the Iris cubes from different datasets can be subtracted. The operation makes the datasets time points common; it also resets the time bounds and auxiliary coordinates to reflect the artificially shifted time points. Current implementation for monthly and daily data; the `frequency` is set automatically from the variable CMOR table unless a custom `frequency` is set manually by the user in recipe.

See also *esmvalcore.preprocessor.regrid_time()*.

8.9.12 timeseries_filter

This function allows the user to apply a filter to the timeseries data. This filter may be of the user's choice (currently only the `low-pass` Lanczos filter is implemented); the implementation is inspired by this iris example and uses aggregation via a rolling window.

Parameters:

- window: the length of the filter window (in units of cube time coordinate).
- span: period (number of months/days, depending on data frequency) on which weights should be computed e.g. for 2-yearly: span = 24 (2 x 12 months). Make sure span has the same units as the data cube time coordinate.
- filter_type: the type of filter to be applied; default 'lowpass'. Available types: 'lowpass'.

- filter_stats: the type of statistic to aggregate on the rolling window; default 'sum'. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max'.

Examples:

- Lowpass filter with a monthly mean as operator:

```
timeseries_filter:
    window: 3  # 3-monthly filter window
    span: 12   # weights computed on the first year
    filter_type: lowpass  # low-pass filter
    filter_stats: mean    # 3-monthly mean lowpass filter
```

See also *esmvalcore.preprocessor.timeseries_filter()*.

8.10 Area manipulation

The area manipulation module contains the following preprocessor functions:

- *extract_region*: Extract a region from a cube based on `lat/lon` corners.
- *extract_named_regions*: Extract a specific region from in the region coordinate.
- *extract_shape*: Extract a region defined by a shapefile.
- *extract_point*: Extract a single point (with interpolation)
- *zonal_statistics*: Compute zonal statistics.
- *meridional_statistics*: Compute meridional statistics.
- *area_statistics*: Compute area statistics.

8.10.1 `extract_region`

This function masks data outside a rectangular region requested. The boundaries of the region are provided as latitude and longitude coordinates in the arguments:

- `start_longitude`
- `end_longitude`
- `start_latitude`
- `end_latitude`

Note that this function can only be used to extract a rectangular region. Use `extract_shape` to extract any other shaped region from a shapefile.

See also *esmvalcore.preprocessor.extract_region()*.

8.10.2 extract_named_regions

This function extracts a specific named region from the data. This function takes the following argument: `regions` which is either a string or a list of strings of named regions. Note that the dataset must have a `region` coordinate which includes a list of strings as values. This function then matches the named regions against the requested string.

See also *esmvalcore.preprocessor.extract_named_regions()*.

8.10.3 extract_shape

Extract a shape or a representative point for this shape from the data.

Parameters:

- `shapefile`: path to the shapefile containing the geometry of the region to be extracted. If the file contains multiple shapes behaviour depends on the decomposed parameter. This path can be relative to `auxiliary_data_dir` defined in the *User configuration file*.
- **`method`: the method to select the region, selecting either all points** contained by the shape or a single representative point. Choose either 'contains' or 'representative'. If not a single grid point is contained in the shape, a representative point will be selected.
- **`crop`: by default *extract_region* will be used to crop the data to a** minimal rectangular region containing the shape. Set to `false` to only mask data outside the shape. Data on irregular grids will not be cropped.
- `decomposed`: by default `false`, in this case the union of all the regions in the shape file is masked out. If `true`, the regions in the shapefiles are masked out separately, generating an auxiliary dimension for the cube for this.

Examples:

- Extract the shape of the river Elbe from a shapefile:

```
extract_shape:
  shapefile: Elbe.shp
  method: contains
```

See also *esmvalcore.preprocessor.extract_shape()*.

8.10.4 extract_point

Extract a single point from the data. This is done using either nearest or linear interpolation.

Returns a cube with the extracted point(s), and with adjusted latitude and longitude coordinates (see below).

Multiple points can also be extracted, by supplying an array of latitude and/or longitude coordinates. The resulting point cube will match the respective latitude and longitude coordinate to those of the input coordinates. If the input coordinate is a scalar, the dimension will be missing in the output cube (that is, it will be a scalar).

Parameters:

- `cube`: the input dataset cube.
- `latitude`, `longitude`: coordinates (as floating point values) of the point to be extracted. Either (or both) can also be an array of floating point values.
- `scheme`: interpolation scheme: either `'linear'` or `'nearest'`. There is no default.

8.10.5 zonal_statistics

The function calculates the zonal statistics by applying an operator along the longitude coordinate. This function takes one argument:

- `operator`: Which operation to apply: mean, std_dev, median, min, max or sum

See also `esmvalcore.preprocessor.zonal_means()`.

8.10.6 meridional_statistics

The function calculates the meridional statistics by applying an operator along the latitude coordinate. This function takes one argument:

- `operator`: Which operation to apply: mean, std_dev, median, min, max or sum

See also `esmvalcore.preprocessor.meridional_means()`.

8.10.7 area_statistics

This function calculates the average value over a region - weighted by the cell areas of the region. This function takes the argument, `operator`: the name of the operation to apply.

This function can be used to apply several different operations in the horizontal plane: mean, standard deviation, median variance, minimum and maximum.

Note that this function is applied over the entire dataset. If only a specific region, depth layer or time period is required, then those regions need to be removed using other preprocessor operations in advance.

The `fx_variables` argument specifies the fx variables that the user wishes to input to the function; the user may specify it as a list of variables e.g.

```
fx_variables: ['areacello', 'volcello']
```

or as list of dictionaries, with specific variable parameters (they key-value pair may be as specific as a CMOR variable can permit):

```
fx_variables: [{'short_name': 'areacello', 'mip': 'Omon'}, {'short_name': 'volcello,␣
↪mip': 'fx'}]
```

The recipe parser wil automatically find the data files that are associated with these variables and pass them to the function for loading and processing.

See also *esmvalcore.preprocessor.area_statistics()*.

8.11 Volume manipulation

The `_volume.py` module contains the following preprocessor functions:

- `extract_volume`: Extract a specific depth range from a cube.
- `volume_statistics`: Calculate the volume-weighted average.
- `depth_integration`: Integrate over the depth dimension.
- `extract_transect`: Extract data along a line of constant latitude or longitude.
- `extract_trajectory`: Extract data along a specified trajectory.

8.11.1 extract_volume

Extract a specific range in the z-direction from a cube. This function takes two arguments, a minimum and a maximum (`z_min` and `z_max`, respectively) in the z-direction.

Note that this requires the requested z-coordinate range to be the same sign as the Iris cube. That is, if the cube has z-coordinate as negative, then `z_min` and `z_max` need to be negative numbers.

See also *esmvalcore.preprocessor.extract_volume()*.

8.11.2 volume_statistics

This function calculates the volume-weighted average across three dimensions, but maintains the time dimension.

This function takes the argument: `operator`, which defines the operation to apply over the volume.

No depth coordinate is required as this is determined by Iris. This function works best when the `fx_variables` provide the cell volume.

The `fx_variables` argument specifies the fx variables that the user wishes to input to the function; the user may specify it as a list of variables e.g.

```
fx_variables: ['areacello', 'volcello']
```

or as list of dictionaries, with specific variable parameters (they key-value pair may be as specific as a CMOR variable can permit):

```
fx_variables: [{'short_name': 'areacello', 'mip': 'Omon'}, {'short_name': 'volcello,␣
↪mip': 'fx'}]
```

The recipe parser wil automatically find the data files that are associated with these variables and pass them to the function for loading and processing.

See also *esmvalcore.preprocessor.volume_statistics()*.

8.11.3 depth_integration

This function integrates over the depth dimension. This function does a weighted sum along the z-coordinate, and removes the z direction of the output cube. This preprocessor takes no arguments.

See also *esmvalcore.preprocessor.depth_integration()*.

8.11.4 extract_transect

This function extracts data along a line of constant latitude or longitude. This function takes two arguments, although only one is strictly required. The two arguments are `latitude` and `longitude`. One of these arguments needs to be set to a float, and the other can then be either ignored or set to a minimum or maximum value.

For example, if we set latitude to 0 N and leave longitude blank, it would produce a cube along the Equator. On the other hand, if we set latitude to 0 and then set longitude to `[40., 100.]` this will produce a transect of the Equator in the Indian Ocean.

See also *esmvalcore.preprocessor.extract_transect()*.

8.11.5 extract_trajectory

This function extract data along a specified trajectory. The three arguments are: `latitudes`, `longitudes` and number of point needed for extrapolation `number_points`.

If two points are provided, the `number_points` argument is used to set a the number of places to extract between the two end points.

If more than two points are provided, then `extract_trajectory` will produce a cube which has extrapolated the data of the cube to those points, and `number_points` is not needed.

Note that this function uses the expensive `interpolate` method from `Iris.analysis.trajectory`, but it may be necessary for irregular grids.

See also *esmvalcore.preprocessor.extract_trajectory()*.

8.12 Cycles

The `_cycles.py` module contains the following preprocessor functions:

- `amplitude`: Extract the peak-to-peak amplitude of a cycle aggregated over specified coordinates.

8.12.1 amplitude

This function extracts the peak-to-peak amplitude (maximum value minus minimum value) of a field aggregated over specified coordinates. Its only argument is `coords`, which can either be a single coordinate (given as `str`) or multiple coordinates (given as `list` of `str`). Usually, these coordinates refer to temporal categorised coordinates like *year*, *month*, *day of year*, etc. For example, to extract the amplitude of the annual cycle for every single year in the data, use `coords: year`; to extract the amplitude of the diurnal cycle for every single day in the data, use `coords: [year, day_of_year]`.

See also *esmvalcore.preprocessor.amplitude()*.

8.13 Detrend

ESMValTool also supports detrending along any dimension using the preprocessor function 'detrend'. This function has two parameters:

- `dimension`: dimension to apply detrend on. Default: "time"
- `method`: It can be `linear` or `constant`. Default: `linear`

If method is `linear`, detrend will calculate the linear trend along the selected axis and subtract it to the data. For example, this can be used to remove the linear trend caused by climate change on some variables is selected dimension is time.

If method is `constant`, detrend will compute the mean along that dimension and subtract it from the data

See also *esmvalcore.preprocessor.detrend()*.

8.14 Unit conversion

Converting units is also supported. This is particularly useful in cases where different datasets might have different units, for example when comparing CMIP5 and CMIP6 variables where the units have changed or in case of observational datasets that are delivered in different units.

In these cases, having a unit conversion at the end of the processing will guarantee homogeneous input for the diagnostics.

Note: Conversion is only supported between compatible units! In other words, converting temperature units from `degC` to `Kelvin` works fine, changing precipitation units from a rate based unit to an amount based unit is not supported at the moment.

See also *esmvalcore.preprocessor.convert_units()*.

8.15 Information on maximum memory required

In the most general case, we can set upper limits on the maximum memory the analysis will require:

`Ms = (R + N) x F_eff - F_eff` - when no multimodel analysis is performed;

`Mm = (2R + N) x F_eff - 2F_eff` - when multimodel analysis is performed;

where

- `Ms`: maximum memory for non-multimodel module
- `Mm`: maximum memory for multimodel module
- `R`: computational efficiency of module; *R* is typically 2-3
- `N`: number of datasets
- `F_eff`: average size of data per dataset where `F_eff = e x f x F` where `e` is the factor that describes how lazy the data is (`e = 1` for fully realized data) and `f` describes how much the data was shrunk by the immediately previous module, e.g. time extraction, area selection or level extraction; note that for fix_data `f` relates only to the time extraction, if data is exact in time (no time selection) `f = 1` for fix_data so for cases when we deal with a lot of datasets `R + N \approx N`, data is fully realized, assuming an average size of 1.5GB for 10 years of *3D* netCDF data, `N` datasets will require:

`Ms = 1.5 x (N - 1)` GB

`Mm = 1.5 x (N - 2)` GB

As a rule of thumb, the maximum required memory at a certain time for multimodel analysis could be estimated by multiplying the number of datasets by the average file size of all the datasets; this memory intake is high but also assumes that all data is fully realized in memory; this aspect will gradually change and the amount of realized data will decrease with the increase of `dask` use.

8.16 Other

Miscellaneous functions that do not belong to any of the other categories.

8.16.1 Clip

This function clips data values to a certain minimum, maximum or range. The function takes two arguments:

- `minimum`: Lower bound of range. Default: `None`
- `maximum`: Upper bound of range. Default: `None`

The example below shows how to set all values below zero to zero.

```
preprocessors:
  clip:
    minimum: 0
    maximum: null
```

Part III

Diagnostic script interfaces

In order to communicate with diagnostic scripts, ESMValCore uses YAML files. The YAML files provided by ESMValCore to the diagnostic script tell the diagnostic script the settings that were provided in the recipe and where to find the pre-processed input data. On the other hand, the YAML file provided by the diagnostic script to ESMValCore tells ESMValCore which pre-processed data was used to create what plots. The latter is optional, but needed for recording provenance.

CHAPTER

NINE

PROVENANCE

When ESMValCore (the `esmvaltool` command) runs a recipe, it will first find all data and run the default preprocessor steps plus any additional preprocessing steps defined in the recipe. Next it will run the diagnostic script defined in the recipe and finally it will store provenance information. Provenance information is stored in the W3C PROV XML format and also plotted in an SVG file for human inspection. In addition to provenance information, a caption is also added to the plots.

CHAPTER

TEN

INFORMATION PROVIDED BY ESMVALCORE TO THE DIAGNOSTIC SCRIPT

To provide the diagnostic script with the information it needs to run (e.g. location of input data, various settings), the ESMValCore creates a YAML file called settings.yml and provides the path to this file as the first command line argument to the diagnostic script.

The most interesting settings provided in this file are

```
run_dir:  /path/to/recipe_output/run/diagnostic_name/script_name
work_dir: /path/to/recipe_output/work/diagnostic_name/script_name
plot_dir: /path/to/recipe_output/plots/diagnostic_name/script_name
input_files:
  - /path/to/recipe_output/preproc/diagnostic_name/ta/metadata.yml
  - /path/to/recipe_output/preproc/diagnostic_name/pr/metadata.yml
```

Custom settings in the script section of the recipe will also be made available in this file.

There are three directories defined:

- `run_dir` use this for storing temporary files
- `work_dir` use this for storing NetCDF files containing the data used to make a plot
- `plot_dir` use this for storing plots

Finally `input_files` is a list of YAML files, containing a description of the preprocessed data. Each entry in these YAML files is a path to a preprocessed file in NetCDF format, with a list of various attributes. An example preprocessor metadata.yml file could look like this:

```
? /path/to/recipe_output/preproc/diagnostic_name/pr/CMIP5_GFDL-ESM2G_Amon_historical_
↪r1i1p1_T2Ms_pr_2000-2002.nc
: alias: GFDL-ESM2G
  cmor_table: CMIP5
  dataset: GFDL-ESM2G
  diagnostic: diagnostic_name
  end_year: 2002
  ensemble: r1i1p1
  exp: historical
  filename: /path/to/recipe_output/preproc/diagnostic_name/pr/CMIP5_GFDL-ESM2G_Amon_
↪historical_r1i1p1_T2Ms_pr_2000-2002.nc
  frequency: mon
  institute: [NOAA-GFDL]
  long_name: Precipitation
  mip: Amon
  modeling_realm: [atmos]
  preprocessor: preprocessor_name
```

(continues on next page)

(continued from previous page)

```
  project: CMIP5
  recipe_dataset_index: 1
  reference_dataset: MPI-ESM-LR
  short_name: pr
  standard_name: precipitation_flux
  start_year: 2000
  units: kg m-2 s-1
  variable_group: pr
? /path/to/recipe_output/preproc/diagnostic_name/pr/CMIP5_MPI-ESM-LR_Amon_historical_
↪r1i1p1_T2Ms_pr_2000-2002.nc
: alias: MPI-ESM-LR
  cmor_table: CMIP5
  dataset: MPI-ESM-LR
  diagnostic: diagnostic_name
  end_year: 2002
  ensemble: r1i1p1
  exp: historical
  filename: /path/to/recipe_output/preproc/diagnostic1/pr/CMIP5_MPI-ESM-LR_Amon_
↪historical_r1i1p1_T2Ms_pr_2000-2002.nc
  frequency: mon
  institute: [MPI-M]
  long_name: Precipitation
  mip: Amon
  modeling_realm: [atmos]
  preprocessor: preprocessor_name
  project: CMIP5
  recipe_dataset_index: 2
  reference_dataset: MPI-ESM-LR
  short_name: pr
  standard_name: precipitation_flux
  start_year: 2000
  units: kg m-2 s-1
  variable_group: pr
```

CHAPTER

ELEVEN

INFORMATION PROVIDED BY THE DIAGNOSTIC SCRIPT TO ESMVALCORE

After the diagnostic script has finished running, ESMValCore will try to store provenance information. In order to link the produced files to input data, the diagnostic script needs to store a YAML file called `diagnostic_provenance.yml` in it's `run_dir`.

For output file produced by the diagnostic script, there should be an entry in the `diagnostic_provenance.yml` file. The name of each entry should be the path to the output file. Each file entry should at least contain the following items

- `ancestors` a list of input files used to create the plot
- `caption` a caption text for the plot
- `plot_file` if the diagnostic also created a plot file, e.g. in .png format.

Each file entry can also contain items from the categories defined in the file `esmvaltool/config_references.yml`. The short entries will automatically be replaced by their longer equivalent in the final provenance records. It is possible to add custom provenance information by adding custom items to entries.

An example `diagnostic_provenance.yml` file could look like this

```
? /path/to/recipe_output/work/diagnostic_name/script_name/CMIP5_GFDL-ESM2G_Amon_
↪historical_r1i1p1_T2Ms_pr_2000-2002_mean.nc
: ancestors:[/path/to/recipe_output/preproc/diagnostic_name/pr/CMIP5_GFDL-ESM2G_Amon_
↪historical_r1i1p1_T2Ms_pr_2000-2002.nc]
  authors: [andela_bouwe, righi_mattia]
  caption: Average Precipitation between 2000 and 2002 according to GFDL-ESM2G.
  domains: [global]
  plot_file: /path/to/recipe_output/plots/diagnostic_name/script_name/CMIP5_GFDL␣
↪ESM2G_Amon_historical_r1i1p1_T2Ms_pr_2000-2002_mean.png
  plot_type: zonal
  references: [acknow_project]
  statistics: [mean]

? /path/to/recipe_output/work/diagnostic_name/script_name/CMIP5_MPI-ESM-LR_Amon_
↪historical_r1i1p1_T2Ms_pr_2000-2002_mean.nc
: ancestors: [/path/to/recipe_output/preproc/diagnostic_name/pr/CMIP5_MPI-ESM-LR_Amon_
↪historical_r1i1p1_T2Ms_pr_2000-2002.nc]
  authors: [andela_bouwe, righi_mattia]
  caption: Average Precipitation between 2000 and 2002 according to MPI-ESM-LR.
  domains: [global]
  plot_file: /path/to/recipe_output/plots/diagnostic_name/script_name/CMIP5_MPI-ESM-
↪LR_Amon_historical_r1i1p1_T2Ms_pr_2000-2002_mean.png
  plot_type: zonal
```

(continues on next page)

(continued from previous page)

```
  references: [acknow_project]
  statistics: [mean]
```

You can check whether your diagnostic script successfully provided the provenance information to the ESMValCore by verifying that

- for each output file in the `work_dir`, a file with the same name, but ending with _provenance.xml is created
- any NetCDF files created by your diagnostic script contain a 'provenance' global attribute
- any PNG plots created by your diagnostic script contain the provenance information in the 'Image History' attribute

Note that this information is included automatically by ESMValCore if the diagnostic script provides the required `diagnostic_provenance.yml` file.

Part IV

Development

CHAPTER

TWELVE

DATASET FIXES

Some (model) datasets contain (known) errors that would normally prevent them from being processed correctly by the ESMValTool. The errors can be in the metadata describing the dataset and/or in the actual data. Typical examples of such errors are missing or wrong attributes (e.g. attribute ''units" says 1e-9 but data are actually in 1e-6), missing or mislabeled coordinates (e.g. ''lev" instead of ''plev" or missing coordinate bounds like ''lat_bnds'') or problems with the actual data (e.g. cloud liquid water only instead of sum of liquid + ice as specified by the CMIP data request).

The ESMValTool can apply on the fly fixes to data sets that have known errors that can be fixed automatically.

Note: CMORization as a fix. As of early 2020, we've started implementing CMORization as fixes for observational datasets. Previously, CMORization was an additional function implemented in ESMValTool. This meant that users always had to store 2 copies of their observational data: both raw and CMORized. Implementing CMORization as a fix removes this redundancy, as the fixes are applied 'on the fly' when running a recipe. **ERA5** is the first dataset for which this 'CMORization on the fly' is supported. For more information about CMORization, see: Contributing a CMORizing script for an observational dataset.

12.1 Fix structure

Fixes are Python classes stored in `esmvalcore/cmor/_fixes/[PROJECT]/[DATASET].py` that derive from `esmvalcore.cmor._fixes.fix.Fix` and are named after the short name of the variable they fix. You can use the name `AllVars` if you want the fix to be applied to the whole dataset

Warning: Be careful to replace any - with _ in your dataset name. We need this replacement to have proper python module names.

The fixes are automatically loaded and applied when the dataset is preprocessed.

12.2 Fixing a dataset

To illustrate the process of creating a fix we are going to construct a new one from scratch for a fictional dataset. We need to fix a CMIPX model called PERFECT-MODEL that is reporting a missing latitude coordinate for variable tas.

12.2.1 Check the output

Next to the error message, you should see some info about the iris cube: size, coordinates. In our example it looks like this:

```
air_temperature/ (K) (time: 312; altitude: 90; longitude: 180)
    Dimension coordinates:
        time                                    x              -              -
        altitude                                -              x              -
        longitude                               -              -              x
    Auxiliary coordinates:
        day_of_month                            x              -              -
        day_of_year                             x              -              -
        month_number                            x              -              -
        year                                    x              -              -
    Attributes:
        {'cmor_table': 'CMIPX', 'mip': 'Amon', 'short_name': 'tas', 'frequency': 'mon
↪'})
```

So now the mistake is clear: the latitude coordinate is badly named and the fix should just rename it.

12.2.2 Create the fix

We start by creating the module file. In our example the path will be `esmvalcore/cmor/_fixes/CMIPX/PERFECT_MODEL.py`. If it already exists just add the class to the file, there is no limit in the number of fixes we can have in any given file.

Then we have to create the class for the fix deriving from `esmvalcore.cmor._fixes.Fix`

```
"""Fixes for PERFECT-MODEL."""
from esmvalcore.cmor.fix import Fix

class tas(Fix):
     """Fixes for tas variable."""
```

Next we must choose the method to use between the ones offered by the Fix class:

- `fix_file` : should be used only to fix errors that prevent data loading. As a rule of thumb, you should only use it if the execution halts before reaching the checks.
- `fix_metadata` : you want to change something in the cube that is not the data (e.g variable or coordinate names, data units).
- `fix_data`: you need to fix the data. Beware: coordinates data values are part of the metadata.

In our case we need to rename the coordinate `altitude` to `latitude`, so we will implement the `fix_metadata` method:

```
"""Fixes for PERFECT-MODEL."""
from esmvalcore.cmor.fix import Fix

class tas(Fix):
    """Fixes for tas variable."""

    def fix_metadata(self, cubes):
        """
        Fix metadata for tas.
```

(continues on next page)

(continued from previous page)

```
        Fix the name of the latitude coordinate, which is called altitude
        in the original file.
        """
        # Sometimes Iris will interpret the data as multiple cubes.
        # Good CMOR datasets will only show one but we support the
        # multiple cubes case to be able to fix the errors that are
        # leading to that extra cubes.
        # In our case this means that we can safely assume that the
        # tas cube is the first one
        tas_cube = cubes[0]
        latitude = tas_cube.coord('altitude')

        # Fix the names. Latitude values, units and
        latitude.short_name = 'lat'
        latitude.standard_name = 'latitude'
        latitude.long_name = 'latitude'
        return cubes
```

This will fix the error. The next time you run ESMValTool you will find that the error is fixed on the fly and, hopefully, your recipe will run free of errors.

Sometimes other errors can appear after you fix the first one because they were hidden by it. In our case, the latitude coordinate could have bad units or values outside the valid range for example. Just extend your fix to address those errors.

12.2.3 Finishing

Chances are that you are not the only one that wants to use that dataset and variable. Other users could take advantage of your fixes as soon as possible. Please, create a separated pull request for the fix and submit it.

It will also be very helpful if you just scan a couple of other variables from the same dataset and check if they share this error. In case that you find that it is a general one, you can change the fix name to `AllVars` so it gets executed for all variables in the dataset. If you find that this is shared only by a handful of similar vars you can just make the fix for those new vars derive from the one you just created:

```
"""Fixes for PERFECT-MODEL."""
from esmvalcore.cmor.fix import Fix

class tas(Fix):
    """Fixes for tas variable."""

    def fix_metadata(self, cubes):
        """
        Fix metadata for tas.

        Fix the name of the latitude coordinate, which is called altitude
        in the original file.
        """
        # Sometimes Iris will interpret the data as multiple cubes.
        # Good CMOR datasets will only show one but we support the
        # multiple cubes case to be able to fix the errors that are
        # leading to that extra cubes.
        # In our case this means that we can safely assume that the
        # tas cube is the first one
        tas_cube = cubes[0]
```

(continues on next page)

(continued from previous page)

```
        latitude = tas_cube.coord('altitude')

        # Fix the names. Latitude values, units and
        latitude.short_name = 'lat'
        latitude.standard_name = 'latitude'
        latitude.long_name = 'latitude'
        return cubes


class ps(tas):
    """Fixes for ps variable."""
```

12.3 Common errors

The above example covers one of the most common cases: variables / coordinates that have names that do not match the expected. But there are some others that use to appear frequently. This section describes the most common cases.

12.3.1 Bad units declared

It is quite common that a variable declares to be using some units but the data is stored in another. This can be solved by overwriting the units attribute with the actual data units.

```
def fix_metadata(self, cubes):
    cube.units = 'real_units'
```

Detecting this error can be tricky if the units are similar enough. It also has a good chance of going undetected until you notice strange results in your diagnostic.

For the above example, it can be useful to access the variable definition and associated coordinate definitions as provided by the CMOR table. For example:

```
def fix_metadata(self, cubes):
    cube.units = self.vardef.units
```

To learn more about what is available in these definitions, see: *esmvalcore.cmor.table.VariableInfo* and *esmvalcore.cmor.table.CoordinateInfo*.

12.3.2 Coordinates missing

Another common error is to have missing coordinates. Usually it just means that the file does not follow the CF-conventions and Iris can therefore not interpret it.

If this is the case, you should see a warning from the ESMValTool about discarding some cubes in the fix metadata step. Just before that warning you should see the full list of cubes as read by Iris. If that list contains your missing coordinate you can create a fix for this model:

```
def fix_metadata(self, cubes):
    coord_cube = cubes.extract_strict('COORDINATE_NAME')
    # Usually this will correspond to an auxiliary coordinate
    # because the most common error is to forget adding it to the
    # coordinates attribute
```

(continues on next page)

(continued from previous page)

```
    coord = iris.coords.AuxCoord(
        coord_cube.data,
        var_name=coord_cube.var_name,
        standard_name=coord_cube.standard_name,
        long_name=coord_cube.long_name,
        units=coord_cube.units,
    )

    # It may also have bounds as another cube
    coord.bounds = cubes.extract_strict('BOUNDS_NAME').data

    data_cube = cubes.extract_strict('VAR_NAME')
    data_cube.add_aux_coord(coord, DIMENSIONS_INDEX_TUPLE)
    return [data_cube]
```

12.4 Customizing checker strictness

The data checker classifies its issues using four different levels of severity. From highest to lowest:

- `CRITICAL`: issues that most of the time will have severe consequences.
- **`ERROR`: issues that usually lead to unexpected errors, but can be safely** ignored sometimes.
- **`WARNING`: something is not up to the standard but is unlikely to have** consequences later.
- **`DEBUG`: any info that the checker wants to communicate. Regardless of** checker strictness, those will always be reported as debug messages.

Users can have control about which levels of issues are interpreted as errors, and therefore make the checker fail or warnings or debug messages. For this purpose there is an optional command line option *–check-level* that can take a number of values, listed below from the lowest level of strictness to the highest:

- **`ignore`: all issues, regardless of severity, will be reported as** warnings. Checker will never fail. Use this at your own risk.
- **`relaxed`: only CRITICAL issues are treated as errors. We recommend not to** rely on this mode, although it can be useful if there are errors preventing the run that you are sure you can manage on the diagnostics or that will not affect you.
- **`default`: fail if there are any CRITICAL or ERROR issues (DEFAULT); Provides** a good measure of safety.
- **`strict`: fail if there are any warnings, this is the highest level of** strictness. Mostly useful for checking datasets that you have produced, to be sure that future users will not be distracted by inoffensive warnings.

CHAPTER

THIRTEEN

VARIABLE DERIVATION

The variable derivation module allows to derive variables which are not in the CMIP standard data request using standard variables as input. All derivations scripts are located in ESMValCore/esmvalcore/preprocessor/_derive/. A typical example looks like this:

```
"""Derivation of variable `dummy`."""
from ._baseclass import DerivedVariableBase


class DerivedVariable(DerivedVariableBase):
    """Derivation of variable `dummy`."""

    @staticmethod
    def required(project):
        """Declare the variables needed for derivation."""
        mip = 'fx'
        if project == 'CMIP6':
            mip = 'Ofx'
        required = [
            {'short_name': 'var_a'},
            {'short_name': 'var_b', 'mip': mip, 'optional': True},
        ]
        return required

    @staticmethod
    def calculate(cubes):
        """Compute `dummy`."""

        # `cubes` is a CubeList containing all required variables.
        cube = do_something_with(cubes)

        # Return single cube at the end
        return cube
```

The static function `required(project)` returns a `list` of `dict` containing all required variables for deriving the derived variable. Its only argument is the `project` of the specific dataset. In this particular example script, the derived variable `dummy` is derived from `var_a` and `var_b`. It is possible to specify arbitrary attributes for each required variable, e.g. `var_b` uses the mip `fx` (or `Ofx` in the case of CMIP6) instead of the original one of `dummy`. Note that you can also declare a required variable as `optional=True`, which allows the skipping of this particular variable during data extraction. For example, this is useful for fx variables which are often not available for observational datasets. Otherwise, the tool will fail if not all required variables are available for all datasets.

The actual derivation takes place in the static function `calculate(cubes)` which returns a single `cube` containing the derived variable. Its only argument `cubes` is a `CubeList` containing all required variables.

Part V

Contributions are very welcome

We greatly value contributions of any kind. Contributions could include, but are not limited to documentation improvements, bug reports, new or improved code, scientific and technical code reviews, infrastructure improvements, mailing list and chat participation, community help/building, education and outreach. We value the time you invest in contributing and strive to make the process as easy as possible. If you have suggestions for improving the process of contributing, please do not hesitate to propose them.

If you have a bug or other issue to report or just need help, please open an issue on the issues tab on the ESMValCore github repository.

If you would like to contribute a new preprocessor function, derived variable, fix for a dataset, or another new feature, please discuss your idea with the development team before getting started, to avoid double work and/or disappointment later. A good way to do this is to open an issue on GitHub.

To get started developing, follow the instructions below. For help with common new features, please have a look at *Development*.

CHAPTER

FOURTEEN

GETTING STARTED

To install for development, follow the instructions in *Installation*.

CHAPTER

FIFTEEN

RUNNING TESTS

Go to the directory where the repository is cloned and run `python setup.py test`. Optionally you can skip tests which require additional dependencies for supported diagnostic script languages by adding `--addopts '-m "not installation"'` to the previous command. Tests will also be run automatically by CircleCI.

CHAPTER

SIXTEEN

CODE STYLE

To increase the readability and maintainability or the ESMValCore source code, we aim to adhere to best practices and coding standards. All pull requests are reviewed and tested by one or more members of the core development team. For code in all languages, it is highly recommended that you split your code up in functions that are short enough to view without scrolling.

16.1 Python

The standard document on best practices for Python code is PEP8 and there is PEP257 for documentation. We make use of numpy style docstrings to document Python functions that are visible on readthedocs.

Most formatting issues in Python code can be fixed automatically by running the commands

```
isort some_file.py
```

to sort the imports in the standard way and

```
yapf -i some_file.py
```

to add/remove whitespace as required by the standard.

To check if your code adheres to the standard, go to the directory where the repository is cloned, e.g. `cd ESMValTool`. and run

```
prospector esmvaltool/diag_scripts/your_diagnostic/your_script.py
```

Run

```
python setup.py lint
```

to see the warnings about the code style of the entire project.

We use `pycodestyle` on CircleCI to automatically check that there are no formatting mistakes and Codacy for monitoring (Python) code quality. Running prospector locally will give you quicker and sometimes more accurate results.

16.2 YAML

Please use `yamllint` to check that your YAML files do not contain mistakes.

16.3 Any text file

A generic tool to check for common spelling mistakes is codespell.

CHAPTER

SEVENTEEN

DOCUMENTATION

17.1 What should be documented

Any code documentation that is visible on readthedocs should be well written and adhere to the standards for documentation for the respective language. Note that there is no need to write extensive documentation for functions that are not visible on readthedocs. However, adding a one line docstring describing what a function does is always a good idea. When making changes/introducing a new preprocessor function, also update the preprocessor documentation.

17.2 How to build the documentation locally

Go to the directory where the repository is cloned and run

```
python setup.py build_sphinx -Ea
```

Make sure that your newly added documentation builds without warnings or errors.

CHAPTER

EIGHTEEN

BRANCHES, PULL REQUESTS AND CODE REVIEW

The default git branch is `master`. Use this branch to create a new feature branch from and make a pull request against. This page offers a good introduction to git branches, but it was written for BitBucket while we use GitHub, so replace the word BitBucket by GitHub whenever you read it.

It is recommended that you open a draft pull request early, as this will cause CircleCI to run the unit tests and Codacy to analyse your code. It's also easier to get help from other developers if your code is visible in a pull request.

You can view the results of the automatic checks below your pull request. If one of the tests shows a red cross instead of a green approval sign, please click the link and try to solve the issue. Note that this kind of automated checks make it easier to review code, but they are not flawless, so occasionally Codacy will report false positives.

18.1 Contributing to the ESMValCore package

Contributions to ESMValCore should

- Preferably be covered by unit tests. Unit tests are mandatory for new preprocessor functions or modifications to existing functions. If you do not know how to start with writing unit tests, let us know in a comment on the pull request and a core development team member will try to help you get started.
- Be accompanied by appropriate documentation.
- Introduce no new issues on Codacy.

18.2 List of authors

If you make a (significant) contribution to ESMValCore, please add your name to the list of authors in CITATION.cff and regenerate the file .zenodo.json by running the command

```
pip install cffconvert
cffconvert --ignore-suspect-keys --outputformat zenodo --outfile .zenodo.json
```

CHAPTER

NINETEEN

HOW TO MAKE A RELEASE

To make a new release of the package, follow these steps:

19.1 1. Check that the nightly build on CircleCI was successful

Check the `nightly` build on CircleCI. All tests should pass before making a release.

19.2 2. Make a pull request to increase the version number

The version number is stored in `esmvalcore/_version.py`, `package/meta.yaml`, `CITATION.cff`. Make sure to update all files. See https://semver.org for more information on choosing a version number.

19.3 3. Make the release on GitHub

Click the releases tab and draft the new release. Do not forget to tick the pre-release box for a beta release. Use the script `esmvalcore/utils/draft_release_notes.py` to create a draft version of the release notes and edit those.

19.4 4. Create and upload the Conda package

Follow these steps to create a new conda package:

- Check out the tag corresponding to the release, e.g. `git checkout v2.0.0b6`
- Edit package/meta.yaml and uncomment the lines starting with `git_rev` and `git_url`, remove the line starting with `path` in the `source` section.
- Activate the base environment `conda activate base`
- Run `conda build package -c conda-forge -c esmvalgroup` to build the conda package
- If the build was successful, upload the package to the esmvalgroup conda channel, e.g. `anaconda upload --user esmvalgroup /path/to/conda/conda-bld/noarch/esmvalcore-2.0.0b6-py_0.tar.bz2`.

Part VI

ESMValTool Core API Reference

ESMValCore is mostly used as a commandline tool. However, it is also possibly to use (parts of) ESMValTool as a library. This section documents the public API of ESMValCore.

CHAPTER

TWENTY

CMOR FUNCTIONS

CMOR module.

20.1 Checking compliance

Module for checking iris cubes against their CMOR definitions.

Classes

CMORCheck(cube, var_info[, frequency, ...])	Class used to check the CMOR-compliance of the data.
CheckLevels(value)	Level of strictness of the checks.

Exceptions

CMORCheckError	Exception raised when a cube does not pass the CMORCheck.

Functions

cmor_check(cube, cmor_table, mip, ...)	Check if cube conforms to variable's CMOR definiton.
cmor_check_data(cube, cmor_table, mip, ...)	Check if data conforms to variable's CMOR definiton.
cmor_check_metadata(cube, cmor_table, mip, ...)	Check if metadata conforms to variable's CMOR definiton.

class esmvalcore.cmor.check.**CMORCheck**(*cube*, *var_info*, *frequency=None*, *fail_on_error=False*, *check_level=<CheckLevels.DEFAULT: 3>*, *automatic_fixes=False*)

Bases: object

Class used to check the CMOR-compliance of the data.

It can also fix some minor errors and does some minor data homogeneization:

Parameters

- **cube** (*iris.cube.Cube:*) – Iris cube to check.
- **var_info** (*variables_info.VariableInfo*) – Variable info to check.
- **frequency** (*str*) – Expected frequency for the data.

- **fail_on_error** (*bool*) – If true, CMORCheck stops on the first error. If false, it collects all possible errors before stopping.
- **automatic_fixes** (*bool*) – If True, CMORCheck will try to apply automatic fixes for any detected error, if possible.
- **check_level** (CheckLevels) – Level of strictness of the checks.

Methods

check_data([logger])	Check the cube data.
check_metadata([logger])	Check the cube metadata.
has_debug_messages()	Check if there are reported debug messages.
has_errors()	Check if there are reported errors.
has_warnings()	Check if there are reported warnings.
report(level, message, *args)	Generic method to report a message from the checker
report_critical(message, *args)	Report an error.
report_debug_message(message, *args)	Report a debug message.
report_debug_messages()	Report detected debug messages to the given logger.
report_error(message, *args)	Report a normal error.
report_errors()	Report detected errors.
report_warning(message, *args)	Report a warning level error.
report_warnings()	Report detected warnings to the given logger.

frequency
Expected frequency for the data.

Type str

check_data(*logger=None*)
Check the cube data.

Performs all the tests that require to have the data in memory. Assumes that metadata is correct, so you must call check_metadata prior to this.

It will also report some warnings in case of minor errors.

Parameters **logger** (*logging.Logger*) – Given logger.

Raises ***CMORCheckError*** – If errors are found. If fail_on_error attribute is set to True, raises as soon as an error is detected. If set to False, it perform all checks and then raises.

check_metadata(*logger=None*)
Check the cube metadata.

Perform all the tests that do not require to have the data in memory.

It will also report some warnings in case of minor errors and homogenize some data:

- Equivalent calendars will all default to the same name.
- Time units will be set to days since 1850-01-01

Parameters **logger** (*logging.Logger*) – Given logger.

Raises ***CMORCheckError*** – If errors are found. If fail_on_error attribute is set to True, raises as soon as an error is detected. If set to False, it perform all checks and then raises.

has_debug_messages()
Check if there are reported debug messages.

Returns True if there are pending debug messages, False otherwise.

Return type bool

has_errors()
Check if there are reported errors.

Returns True if there are pending errors, False otherwise.

Return type bool

has_warnings()
Check if there are reported warnings.

Returns True if there are pending warnings, False otherwise.

Return type bool

report(*level*, *message*, **args*)
Generic method to report a message from the checker

Parameters

- **level** (CheckLevels) – Message level
- **message** (*str*) – Message to report
- **args** – String format args for the message

Raises ***CMORCheckError*** – If fail on error is set, it is thrown when registering an error message

report_critical(*message*, **args*)
Report an error.

If fail_on_error is set to True, raises automatically. If fail_on_error is set to False, stores it for later reports.

Parameters

- **message** (*str: unicode*) – Message for the error.
- ***args** – arguments to format the message string.

report_debug_message(*message*, **args*)
Report a debug message.

Parameters

- **message** (*str: unicode*) – Message for the debug logger.
- ***args** – arguments to format the message string

report_debug_messages()
Report detected debug messages to the given logger.

Parameters **logger** (*logging.Logger*) – Given logger.

report_error(*message*, **args*)
Report a normal error.

Parameters

- **message** (*str: unicode*) – Message for the error.
- ***args** – arguments to format the message string.

report_errors()
Report detected errors.

Raises *CMORCheckError* – If any errors were reported before calling this method.

report_warning(*message*, **args*)
Report a warning level error.

Parameters

- **message** (*str: unicode*) – Message for the warning.
- ***args** – arguments to format the message string.

report_warnings()
Report detected warnings to the given logger.

Parameters **logger** (*logging.Logger*) – Given logger

exception esmvalcore.cmor.check.**CMORCheckError**
Bases: Exception

Exception raised when a cube does not pass the CMORCheck.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class esmvalcore.cmor.check.**CheckLevels**(*value*)
Bases: enum.IntEnum

Level of strictness of the checks.

Attributes

DEBUG	int([x]) -> integer
STRICT	int([x]) -> integer
DEFAULT	int([x]) -> integer
RELAXED	int([x]) -> integer
IGNORE	int([x]) -> integer

- DEBUG

Type Report any debug message that the checker wants to communicate.

- STRICT

Type Fail if there are warnings regarding compliance of CMOR standards.

- DEFAULT

Type Fail if cubes present any discrepancy with CMOR standards.

- RELAXED

Type Fail if cubes present severe discrepancies with CMOR standards.

- IGNORE

Type Do not fail for any discrepancy with CMOR standards.

DEBUG = 1

DEFAULT = 3

IGNORE = 5

RELAXED = 4

STRICT = 2

esmvalcore.cmor.check.**cmor_check**(*cube*, *cmor_table*, *mip*, *short_name*, *frequency*, *check_level*)

Check if cube conforms to variable's CMOR definiton.

Equivalent to calling cmor_check_metadata and cmor_check_data consecutively.

Parameters

- **cube** (*iris.cube.Cube*) – Data cube to check.
- **cmor_table** (*basestring*) – CMOR definitions to use.
- **mip** – Variable's mip.
- **short_name** (*basestring*) – Variable's short name.
- **frequency** (*basestring*) – Data frequency.
- **check_level** (*enum.IntEnum*) – Level of strictness of the checks.

esmvalcore.cmor.check.**cmor_check_data**(*cube*, *cmor_table*, *mip*, *short_name*, *frequency*, *check_level=<CheckLevels.DEFAULT: 3>*)

Check if data conforms to variable's CMOR definiton.

The checks performed at this step require the data in memory.

Parameters

- **cube** (*iris.cube.Cube*) – Data cube to check.
- **cmor_table** (*basestring*) – CMOR definitions to use.
- **mip** – Variable's mip.
- **short_name** (*basestring*) – Variable's short name
- **frequency** (*basestring*) – Data frequency
- **check_level** (CheckLevels) – Level of strictness of the checks.

esmvalcore.cmor.check.**cmor_check_metadata**(*cube*, *cmor_table*, *mip*, *short_name*, *frequency*, *check_level=<CheckLevels.DEFAULT: 3>*)

Check if metadata conforms to variable's CMOR definiton.

None of the checks at this step will force the cube to load the data.

Parameters

- **cube** (*iris.cube.Cube*) – Data cube to check.
- **cmor_table** (*basestring*) – CMOR definitions to use.
- **mip** – Variable's mip.
- **short_name** (*basestring*) – Variable's short name.
- **frequency** (*basestring*) – Data frequency.
- **check_level** (CheckLevels) – Level of strictness of the checks.

20.2 Automatically fixing issues

Apply automatic fixes for known errors in cmorized data

All functions in this module will work even if no fixes are available for the given dataset. Therefore is recommended to apply them to all variables to be sure that all known errors are fixed.

Functions

fix_data(cube, short_name, project, dataset, mip)	Fix cube data if fixes add present and check it anyway.
fix_file(file, short_name, project, dataset, ...)	Fix files before ESMValTool can load them.
fix_metadata(cubes, short_name, project, ...)	Fix cube metadata if fixes are required and check it anyway.

esmvalcore.cmor.fix.**fix_data**(*cube*, *short_name*, *project*, *dataset*, *mip*, *frequency=None*, *check_level=<CheckLevels.DEFAULT: 3>*)

Fix cube data if fixes add present and check it anyway.

This method assumes that metadata is already fixed and checked.

This method collects all the relevant fixes for a given variable, applies them and checks resulting cube (or the original if no fixes were needed) metadata to ensure that it complies with the standards of its project CMOR tables.

Parameters

- **cube** (*iris.cube.Cube*) – Cube to fix
- **short_name** (*str*) – Variable's short name
- **project** (*str*) –
- **dataset** (*str*) –
- **mip** (*str*) – Variable's MIP
- **frequency** (*str, optional*) – Variable's data frequency, if available
- **check_level** (CheckLevels) – Level of strictness of the checks. Set to default.

Returns Fixed and checked cube

Return type iris.cube.Cube

Raises ***CMORCheckError*** – If the checker detects errors in the data that it can not fix.

esmvalcore.cmor.fix.**fix_file**(*file*, *short_name*, *project*, *dataset*, *mip*, *output_dir*)

Fix files before ESMValTool can load them.

This fixes are only for issues that prevent iris from loading the cube or that cannot be fixed after the cube is loaded.

Original files are not overwritten.

Parameters

- **file** (*str*) – Path to the original file
- **short_name** (*str*) – Variable's short name
- **project** (*str*) –
- **dataset** (*str*) –
- **output_dir** (*str*) – Output directory for fixed files

Returns Path to the fixed file

Return type str

esmvalcore.cmor.fix.**fix_metadata**(*cubes*, *short_name*, *project*, *dataset*, *mip*, *frequency=None*, *check_level=<CheckLevels.DEFAULT: 3>*)

Fix cube metadata if fixes are required and check it anyway.

This method collects all the relevant fixes for a given variable, applies them and checks the resulting cube (or the original if no fixes were needed) metadata to ensure that it complies with the standards of its project CMOR tables.

Parameters

- **cubes** (*iris.cube.CubeList*) – Cubes to fix
- **short_name** (*str*) – Variable's short name
- **project** (*str*) –
- **dataset** (*str*) –
- **mip** (*str*) – Variable's MIP
- **frequency** (*str, optional*) – Variable's data frequency, if available
- **check_level** (CheckLevels) – Level of strictness of the checks. Set to default.

Returns Fixed and checked cube

Return type iris.cube.Cube

Raises ***CMORCheckError*** – If the checker detects errors in the metadata that it can not fix.

20.3 Functions for fixing issues

Functions for fixing specific issues with datasets.

Functions

add_plev_from_altitude(cube)	Add pressure level coordinate from altitude coordinate.
add_sigma_factory(cube)	Add factory for `atmosphere_sigma_coordinate`.

esmvalcore.cmor.fixes.**add_plev_from_altitude**(*cube*)

Add pressure level coordinate from altitude coordinate.

Parameters **cube** (*iris.cube.Cube*) – Input cube.

Raises **ValueError** – `cube` does not contain coordinate `altitude`.

esmvalcore.cmor.fixes.**add_sigma_factory**(*cube*)

Add factory for `atmosphere_sigma_coordinate`.

Parameters **cube** (*iris.cube.Cube*) – Input cube.

Raises **ValueError** – `cube` does not contain coordinate `atmosphere_sigma_coordinate`.

20.4 Using CMOR tables

CMOR information reader for ESMValTool.

Read variable information from CMOR 2 and CMOR 3 tables and make it easily available for the other components of ESMValTool

Classes

CMIP3Info(cmor_tables_path[, default, strict])	Class to read CMIP3-like data request.
CMIP5Info(cmor_tables_path[, default, strict])	Class to read CMIP5-like data request.
CMIP6Info(cmor_tables_path[, default, ...])	Class to read CMIP6-like data request.
CoordinateInfo(name)	Class to read and store coordinate information.
CustomInfo([cmor_tables_path])	Class to read custom var info for ESMVal.
JsonInfo()	Base class for the info classes.
TableInfo(*args, **kwargs)	Container class for storing a CMOR table.
VariableInfo(table_type, short_name)	Class to read and store variable information.

Data

CMOR_TABLES	dict of str, obj: CMOR info objects.

Functions

get_var_info(project, mip, short_name)	Get variable information.
read_cmor_tables([cfg_developer])	Read cmor tables required in the configuration.

class esmvalcore.cmor.table.**CMIP3Info**(*cmor_tables_path*, *default=None*, *strict=True*)

Bases: *esmvalcore.cmor.table.CMIP5Info*

Class to read CMIP3-like data request.

Parameters

- **cmor_tables_path** (*basestring*) – Path to the folder containing the Tables folder with the json files
- **default** (*object*) – Default table to look variables on if not found
- **strict** (*bool*) – If False, will look for a variable in other tables if it can not be found in the requested one

Methods

get_table(table)	Search and return the table info.
get_variable(table, short_name[, derived])	Search and return the variable info.

get_table(*table*)

Search and return the table info.

Parameters **table** (*basestring*) – Table name

Returns Return the TableInfo object for the requested table if found, returns None if not

Return type *TableInfo*

get_variable(*table*, *short_name*, *derived=False*)

Search and return the variable info.

Parameters

- **table** (*basestring*) – Table name
- **short_name** (*basestring*) – Variable's short name
- **derived** (*bool, optional*) – Variable is derived. Info retrieval is less strict

Returns Return the VariableInfo object for the requested variable if found, returns None if not

Return type *VariableInfo*

class esmvalcore.cmor.table.**CMIP5Info**(*cmor_tables_path*, *default=None*, *strict=True*)

Bases: object

Class to read CMIP5-like data request.

Parameters

- **cmor_tables_path** (*basestring*) – Path to the folder containing the Tables folder with the json files
- **default** (*object*) – Default table to look variables on if not found
- **strict** (*bool*) – If False, will look for a variable in other tables if it can not be found in the requested one

Methods

get_table(table)	Search and return the table info.
get_variable(table, short_name[, derived])	Search and return the variable info.

get_table(*table*)

Search and return the table info.

Parameters **table** (*basestring*) – Table name

Returns Return the TableInfo object for the requested table if found, returns None if not

Return type *TableInfo*

get_variable(*table*, *short_name*, *derived=False*)

Search and return the variable info.

Parameters

- **table** (*basestring*) – Table name
- **short_name** (*basestring*) – Variable's short name
- **derived** (*bool, optional*) – Variable is derived. Info retrieval is less strict

Returns Return the VariableInfo object for the requested variable if found, returns None if not

Return type *VariableInfo*

class esmvalcore.cmor.table.**CMIP6Info**(*cmor_tables_path*, *default=None*, *strict=True*, *default_table_prefix=''*)

Bases: object

Class to read CMIP6-like data request.

This uses CMOR 3 json format

Parameters

- **cmor_tables_path** (*basestring*) – Path to the folder containing the Tables folder with the json files
- **default** (*object*) – Default table to look variables on if not found
- **strict** (*bool*) – If False, will look for a variable in other tables if it can not be found in the requested one

Methods

get_table(table)	Search and return the table info.
get_variable(table_name, short_name[, derived])	Search and return the variable info.

get_table(*table*)
Search and return the table info.

Parameters **table** (*basestring*) – Table name

Returns Return the TableInfo object for the requested table if found, returns None if not

Return type *TableInfo*

get_variable(*table_name*, *short_name*, *derived=False*)
Search and return the variable info.

Parameters

- **table_name** (*basestring*) – Table name
- **short_name** (*basestring*) – Variable's short name
- **derived** (*bool, optional*) – Variable is derived. Info retrieval is less strict

Returns Return the VariableInfo object for the requested variable if found, returns None if not

Return type *VariableInfo*

esmvalcore.cmor.table.**CMOR_TABLES = {'CMIP3': <esmvalcore.cmor.table.CMIP3Info object>, 'C**
CMOR info objects.

Type dict of str, obj

class esmvalcore.cmor.table.**CoordinateInfo**(*name*)
Bases: *esmvalcore.cmor.table.JsonInfo*

Class to read and store coordinate information. **Methods**

read_json(json_data)	Read coordinate information from json.

Attributes

axis	Axis
value	Coordinate value
standard_name	Standard name
long_name	Long name
out_name	Out name
var_name	Short name
units	Units
stored_direction	Direction in which the coordinate increases

continues on next page

Table 15 – continued from previous page

requested	Values requested
valid_min	Minimum allowed value
valid_max	Maximum allowed value
must_have_bounds	Whether bounds are required on this dimension

axis
Axis

long_name
Long name

must_have_bounds
Whether bounds are required on this dimension

out_name
Out name

This is the name of the variable in the file

read_json(*json_data*)
Read coordinate information from json.

Non-present options will be set to empty

Parameters **json_data** (*dict*) – dictionary created by the json reader containing coordinate information

requested
Values requested

standard_name
Standard name

stored_direction
Direction in which the coordinate increases

units
Units

valid_max
Maximum allowed value

valid_min
Minimum allowed value

value
Coordinate value

var_name
Short name

class esmvalcore.cmor.table.**CustomInfo**(*cmor_tables_path=None*)
Bases: *esmvalcore.cmor.table.CMIP5Info*

Class to read custom var info for ESMVal.

Parameters **cmor_tables_path** (*basestring or None*) – Full path to the table or name for the table if it is present in ESMValTool repository

Methods

get_table(table)	Search and return the table info.
get_variable(table, short_name[, derived])	Search and return the variable info.

get_table (*table*)
Search and return the table info.

Parameters **table** (*basestring*) – Table name

Returns Return the TableInfo object for the requested table if found, returns None if not

Return type *TableInfo*

get_variable (*table*, *short_name*, *derived=False*)
Search and return the variable info.

Parameters

- **table** (*basestring*) – Table name
- **short_name** (*basestring*) – Variable's short name
- **derived** (*bool, optional*) – Variable is derived. Info retrieval is less strict

Returns Return the VariableInfo object for the requested variable if found, returns None if not

Return type *VariableInfo*

class esmvalcore.cmor.table.**JsonInfo**
Bases: object

Base class for the info classes.

Provides common utility methods to read json variables

class esmvalcore.cmor.table.**TableInfo** (**args*, ***kwargs*)
Bases: dict

Container class for storing a CMOR table. **Methods**

clear()	
copy()	
fromkeys([value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
popitem()	2-tuple; but raise KeyError if D is empty.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
values()	

clear () → None. Remove all items from D.

copy () → a shallow copy of D

fromkeys(*value=None*, */*)
Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)
Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

setdefault(*key*, *default=None*, */*)
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class esmvalcore.cmor.table.**VariableInfo**(*table_type*, *short_name*)
Bases: *esmvalcore.cmor.table.JsonInfo*

Class to read and store variable information. **Methods**

copy()	Return a shallow copy of VariableInfo.
read_json(json_data, default_freq)	Read variable information from json.

Attributes

modeling_realm	Modeling realm
short_name	Short name
standard_name	Standard name
long_name	Long name
units	Data units
valid_min	Minimum admitted value
valid_max	Maximum admitted value
frequency	Data frequency
positive	Increasing direction
dimensions	List of dimensions
coordinates	Coordinates

coordinates
Coordinates

This is a dict with the names of the dimensions as keys and CoordinateInfo objects as values.

copy()
Return a shallow copy of VariableInfo.

Returns Shallow copy of this object

Return type *VariableInfo*

dimensions
List of dimensions

frequency
Data frequency

long_name
Long name

modeling_realm
Modeling realm

positive
Increasing direction

read_json(*json_data*, *default_freq*)
Read variable information from json.

Non-present options will be set to empty

Parameters

- **json_data** (*dict*) – dictionary created by the json reader containing variable information
- **default_freq** (*str*) – Default frequency to use if it is not defined at variable level

short_name
Short name

standard_name
Standard name

units
Data units

valid_max
Maximum admitted value

valid_min
Minimum admitted value

esmvalcore.cmor.table.**get_var_info**(*project*, *mip*, *short_name*)
Get variable information.

Parameters

- **project** (*str*) – Dataset's project.
- **mip** (*str*) – Variable's cmor table.
- **short_name** (*str*) – Variable's short name.

esmvalcore.cmor.table.**read_cmor_tables**(*cfg_developer=None*)
Read cmor tables required in the configuration.

Parameters **cfg_developer** (*dict of str*) – Parsed config-developer file

CHAPTER

TWENTYONE

PREPROCESSOR FUNCTIONS

By default, preprocessor functions are applied in the order in which they are listed here.

Preprocessor module.

Functions

download(files, dest_folder)	Download files that are not available locally
fix_file(file, short_name, project, dataset, ...)	Fix files before ESMValTool can load them.
load(file[, callback])	Load iris cubes from files.
derive(cubes, short_name, long_name, units)	Derive variable.
fix_metadata(cubes, short_name, project, ...)	Fix cube metadata if fixes are required and check it anyway.
concatenate(cubes)	Concatenate all cubes after fixing metadata.
cmor_check_metadata(cube, cmor_table, mip, ...)	Check if metadata conforms to variable's CMOR definiton.
extract_time(cube, start_year, start_month, ...)	Extract a time range from a cube.
extract_season(cube, season)	Slice cube to get only the data belonging to a specific season.
extract_month(cube, month)	Slice cube to get only the data belonging to a specific month.
fix_data(cube, short_name, project, dataset, mip)	Fix cube data if fixes add present and check it anyway.
extract_levels(cube, levels, scheme[, ...])	Perform vertical interpolation.
weighting_landsea_fraction(cube, ...)	Weight fields using land or sea fraction.
mask_landsea(cube, fx_variables, mask_out[, ...])	Mask out either land mass or sea (oceans, seas and lakes).
mask_glaciated(cube, mask_out)	Mask out glaciated areas.
mask_landseaice(cube, fx_variables, mask_out)	Mask out either landsea (combined) or ice.
regrid(cube, target_grid, scheme[, ...])	Perform horizontal regridding.
extract_point(cube, latitude, longitude, scheme)	Extract a point, with interpolation
mask_fillvalues(products, threshold_fraction)	Compute and apply a multi-dataset fillvalues mask.
mask_above_threshold(cube, threshold)	Mask above a specific threshold value.
mask_below_threshold(cube, threshold)	Mask below a specific threshold value.
mask_inside_range(cube, minimum, maximum)	Mask inside a specific threshold range.
mask_outside_range(cube, minimum, maximum)	Mask outside a specific threshold range.
clip(cube[, minimum, maximum])	Clip values at a specified minimum and/or maximum value
extract_region(cube, start_longitude, ...)	Extract a region from a cube.
extract_shape(cube, shapefile[, method, ...])	Extract a region defined by a shapefile.
extract_volume(cube, z_min, z_max)	Subset a cube based on a range of values in the z-coordinate.

continues on next page

Table 1 – continued from previous page

extract_trajectory(cube, latitudes, longitudes)	Extract data along a trajectory.
extract_transect(cube[, latitude, longitude])	Extract data along a line of constant latitude or longitude.
detrend(cube[, dimension, method])	Detrend data along a given dimension.
multi_model_statistics(products, span, ...)	Compute multi-model statistics.
extract_named_regions(cube, regions)	Extract a specific named region.
depth_integration(cube)	Determine the total sum over the vertical component.
area_statistics(cube, operator[, fx_variables])	Apply a statistical operator in the horizontal direction.
volume_statistics(cube, operator[, fx_variables])	Apply a statistical operation over a volume.
amplitude(cube, coords)	Calculate amplitude of cycles by aggregating over coordinates.
zonal_statistics(cube, operator)	Compute zonal statistics.
meridional_statistics(cube, operator)	Compute meridional statistics.
daily_statistics(cube[, operator])	Compute daily statistics.
monthly_statistics(cube[, operator])	Compute monthly statistics.
seasonal_statistics(cube[, operator])	Compute seasonal statistics.
annual_statistics(cube[, operator])	Compute annual statistics.
decadal_statistics(cube[, operator])	Compute decadal statistics.
climate_statistics(cube[, operator, period])	Compute climate statistics with the specified granularity.
anomalies(cube, period[, reference, standardize])	Compute anomalies using a mean with the specified granularity.
regrid_time(cube, frequency)	Align time axis for cubes so they can be subtracted.
timeseries_filter(cube, window, span[, ...])	Apply a timeseries filter.
cmor_check_data(cube, cmor_table, mip, ...)	Check if data conforms to variable's CMOR definiton.
convert_units(cube, units)	Convert the units of a cube to new ones.
save(cubes, filename[, optimize_access, ...])	Save iris cubes to file.
cleanup(files[, remove])	Clean up after running the preprocessor.

esmvalcore.preprocessor.**amplitude**(*cube*, *coords*)

Calculate amplitude of cycles by aggregating over coordinates.

Note: The amplitude is calculated as *peak-to-peak* amplitude (difference between maximum and minimum value of the signal). Other amplitude types are currently not supported.

Parameters

- **cube** (*iris.cube.Cube*) – Input data.
- **coords** (*str or list of str*) – Coordinates over which is aggregated. For example, use `'year'` to extract the annual cycle amplitude for each year in the data or `['day_of_year', 'year']` to extract the diurnal cycle amplitude for each individual day in the data. If the coordinates are not found in `cube`, try to add it via `iris.coord_categorisation` (at the moment, this only works for the temporal coordinates `day_of_month`, `day_of_year`, `hour`, `month`, `month_fullname`, `month_number`, `season`, `season_number`, `season_year`, `weekday`, `weekday_fullname`, `weekday_number` or `year`.

Returns Amplitudes.

Return type iris.cube.Cube

Raises **iris.exceptions.CoordinateNotFoundError** – A coordinate is not found in `cube` and cannot be added via `iris.coord_categorisation`.

`esmvalcore.preprocessor.`**annual_statistics**(*cube*, *operator='mean'*)

Compute annual statistics.

Note that this function does not weight the annual mean if uneven time periods are present. Ie, all data inside the year are treated equally.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **operator** (*str, optional*) – Select operator to apply. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max'

Returns Annual statistics cube

Return type iris.cube.Cube

`esmvalcore.preprocessor.`**anomalies**(*cube*, *period*, *reference=None*, *standardize=False*)

Compute anomalies using a mean with the specified granularity.

Computes anomalies based on daily, monthly, seasonal or yearly means for the full available period

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **period** (*str*) – Period to compute the statistic over. Available periods: 'full', 'season', 'seasonal', 'monthly', 'month', 'mon', 'daily', 'day'
- **reference** (*list int, optional, default: None*) – Period of time to use a reference, as needed for the 'extract_time' preprocessor function If None, all available data is used as a reference
- **standardize** (*bool, optional*) – If True standardized anomalies are calculated

Returns Anomalies cube

Return type iris.cube.Cube

`esmvalcore.preprocessor.`**area_statistics**(*cube*, *operator*, *fx_variables=None*)

Apply a statistical operator in the horizontal direction.

The average in the horizontal direction. We assume that the horizontal directions are ['longitude', 'latutude'].

This function can be used to apply several different operations in the horizontal plane: mean, standard deviation, median variance, minimum and maximum. These options are specified using the *operator* argument and the following key word arguments:

mean	Area weighted mean.
median	Median (not area weighted)
std_dev	Standard Deviation (not area weighted)
sum	Area weighted sum.
variance	Variance (not area weighted)
min:	Minimum value
max	Maximum value

Parameters

- **cube** (*iris.cube.Cube*) – Input cube.

- **operator** (*str*) – The operation, options: mean, median, min, max, std_dev, sum, variance
- **fx_variables** (*dict*) – dictionary of field:filename for the fx_variables

Returns collapsed cube.

Return type iris.cube.Cube

Raises

- **iris.exceptions.CoordinateMultiDimError** – Exception for latitude axis with dim > 2.
- **ValueError** – if input data cube has different shape than grid area weights

esmvalcore.preprocessor.**cleanup**(*files*, *remove=None*)

Clean up after running the preprocessor.

esmvalcore.preprocessor.**climate_statistics**(*cube*, *operator='mean'*, *period='full'*)

Compute climate statistics with the specified granularity.

Computes statistics for the whole dataset. It is possible to get them for the full period or with the data grouped by day, month or season

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **operator** (*str, optional*) – Select operator to apply. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max'
- **period** (*str, optional*) – Period to compute the statistic over. Available periods: 'full', 'season', 'seasonal', 'monthly', 'month', 'mon', 'daily', 'day'

Returns Monthly statistics cube

Return type iris.cube.Cube

esmvalcore.preprocessor.**clip**(*cube*, *minimum=None*, *maximum=None*)

Clip values at a specified minimum and/or maximum value

Values lower than minimum are set to minimum and values higher than maximum are set to maximum.

Parameters

- **cube** (*iris.cube.Cube*) – iris cube to be clipped
- **minimum** (*float*) – lower threshold to be applied on input cube data.
- **maximum** (*float*) – upper threshold to be applied on input cube data.

Returns clipped cube.

Return type iris.cube.Cube

esmvalcore.preprocessor.**cmor_check_data**(*cube*, *cmor_table*, *mip*, *short_name*, *frequency*, *check_level=<CheckLevels.DEFAULT: 3>*)

Check if data conforms to variable's CMOR definiton.

The checks performed at this step require the data in memory.

Parameters

- **cube** (*iris.cube.Cube*) – Data cube to check.
- **cmor_table** (*basestring*) – CMOR definitions to use.

- **mip** – Variable's mip.
- **short_name** (*basestring*) – Variable's short name
- **frequency** (*basestring*) – Data frequency
- **check_level** (CheckLevels) – Level of strictness of the checks.

esmvalcore.preprocessor.**cmor_check_metadata**(*cube*, *cmor_table*, *mip*, *short_name*, *frequency*, *check_level=<CheckLevels.DEFAULT: 3>*)

Check if metadata conforms to variable's CMOR definiton.

None of the checks at this step will force the cube to load the data.

Parameters

- **cube** (*iris.cube.Cube*) – Data cube to check.
- **cmor_table** (*basestring*) – CMOR definitions to use.
- **mip** – Variable's mip.
- **short_name** (*basestring*) – Variable's short name.
- **frequency** (*basestring*) – Data frequency.
- **check_level** (CheckLevels) – Level of strictness of the checks.

esmvalcore.preprocessor.**concatenate**(*cubes*)

Concatenate all cubes after fixing metadata.

esmvalcore.preprocessor.**convert_units**(*cube*, *units*)

Convert the units of a cube to new ones.

This converts units of a cube.

Parameters

- **cube** (*iris.cube.Cube*) – input cube
- **units** (*str*) – new units in udunits form

Returns converted cube.

Return type iris.cube.Cube

esmvalcore.preprocessor.**daily_statistics**(*cube*, *operator='mean'*)

Compute daily statistics.

Chunks time in daily periods and computes statistics over them;

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **operator** (*str, optional*) – Select operator to apply. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max'

Returns Daily statistics cube

Return type iris.cube.Cube

esmvalcore.preprocessor.**decadal_statistics**(*cube*, *operator='mean'*)

Compute decadal statistics.

Note that this function does not weight the decadal mean if uneven time periods are present. Ie, all data inside the decade are treated equally.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **operator** (*str, optional*) – Select operator to apply. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max'

Returns Decadal statistics cube

Return type iris.cube.Cube

esmvalcore.preprocessor.**depth_integration**(*cube*)

Determine the total sum over the vertical component.

Requires a 3D cube. The z-coordinate integration is calculated by taking the sum in the z direction of the cell contents multiplied by the cell thickness.

Parameters **cube** (*iris.cube.Cube*) – input cube.

Returns collapsed cube.

Return type iris.cube.Cube

esmvalcore.preprocessor.**derive**(*cubes*, *short_name*, *long_name*, *units*, *standard_name=None*)

Derive variable.

Parameters

- **cubes** (*iris.cube.CubeList*) – Includes all the needed variables for derivation defined in `get_required()`.
- **short_name** (*str*) – short_name
- **long_name** (*str*) – long_name
- **units** (*str*) – units
- **standard_name** (*str, optional*) – standard_name

Returns The new derived variable.

Return type iris.cube.Cube

esmvalcore.preprocessor.**detrend**(*cube*, *dimension='time'*, *method='linear'*)

Detrend data along a given dimension.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **dimension** (*str*) – Dimension to detrend
- **method** (*str*) – Method to detrend. Available: linear, constant. See documentation of 'scipy.signal.detrend' for details

Returns Detrended cube

Return type iris.cube.Cube

esmvalcore.preprocessor.**download**(*files*, *dest_folder*)

Download files that are not available locally

esmvalcore.preprocessor.**extract_levels**(*cube*, *levels*, *scheme*, *coordinate=None*)

Perform vertical interpolation.

Parameters

- **cube** (*cube*) – The source cube to be vertically interpolated.

- **levels** (*array*) – One or more target levels for the vertical interpolation. Assumed to be in the same S.I. units of the source cube vertical dimension coordinate.
- **scheme** (*str*) – The vertical interpolation scheme to use. Choose from 'linear', 'nearest', 'nearest_horizontal_extrapolate_vertical', 'linear_horizontal_extrapolate_vertical'.
- **coordinate** (*optional str*) – The coordinate to interpolate

Returns

Return type cube

See also:

regrid() Perform horizontal regridding.

esmvalcore.preprocessor.**extract_month**(*cube*, *month*)

Slice cube to get only the data belonging to a specific month.

Parameters

- **cube** (*iris.cube.Cube*) – Original data
- **month** (*int*) – Month to extract as a number from 1 to 12

Returns data cube for specified month.

Return type iris.cube.Cube

esmvalcore.preprocessor.**extract_named_regions**(*cube*, *regions*)

Extract a specific named region.

The region coordinate exist in certain CMIP datasets. This preprocessor allows a specific named regions to be extracted.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **regions** (*str, list*) – A region or list of regions to extract.

Returns collapsed cube.

Return type iris.cube.Cube

Raises

- **ValueError** – regions is not list or tuple or set.
- **ValueError** – region not included in cube.

esmvalcore.preprocessor.**extract_point**(*cube*, *latitude*, *longitude*, *scheme*)

Extract a point, with interpolation

Extracts a single latitude/longitude point from a cube, according to the interpolation scheme *scheme*.

Multiple points can also be extracted, by supplying an array of latitude and/or longitude coordinates. The resulting point cube will match the respective latitude and longitude coordinate to those of the input coordinates. If the input coordinate is a scalar, the dimension will be missing in the output cube (that is, it will be a scalar).

Parameters

- **cube** (*cube*) – The source cube to extract a point from.
- **longitude** (*latitude,*) – The latitude and longitude of the point.
- **scheme** (*str*) – The interpolation scheme. 'linear' or 'nearest'. No default.

Returns

- *Returns a cube with the extracted point(s), and with adjusted*
- *latitude and longitude coordinates (see above).*

Examples

With a cube that has the coordinates

- latitude: [1, 2, 3, 4]
- longitude: [1, 2, 3, 4]
- **data values: [[[1, 2, 3, 4], [5, 6, . . .], [. . .], [. . .],** ...]]]

```
>>> point = extract_point(cube, 2.5, 2.5, 'linear')
>>> point.data
array([ 8.5, 24.5, 40.5, 56.5])
```

Extraction of multiple points at once, with a nearest matching scheme. The values for 0.1 will result in masked values, since this lies outside the cube grid.

```
>>> point = extract_point(cube, [1.4, 2.1], [0.1, 1.1],
...                       'nearest')
>>> point.data.shape
(4, 2, 2)
>>> # x, y, z indices of masked values
>>> np.where(~point.data.mask)
(array([0, 0, 1, 1, 2, 2, 3, 3]), array([0, 1, 0, 1, 0, 1, 0, 1]),
array([1, 1, 1, 1, 1, 1, 1, 1]))
>>> point.data[~point.data.mask].data
array([ 1,  5, 17, 21, 33, 37, 49, 53])
```

esmvalcore.preprocessor.**extract_region**(*cube*, *start_longitude*, *end_longitude*, *start_latitude*, *end_latitude*)

Extract a region from a cube.

Function that subsets a cube on a box (start_longitude, end_longitude, start_latitude, end_latitude) This function is a restriction of masked_cube_lonlat().

Parameters

- **cube** (*iris.cube.Cube*) – input data cube.
- **start_longitude** (*float*) – Western boundary longitude.
- **end_longitude** (*float*) – Eastern boundary longitude.
- **start_latitude** (*float*) – Southern Boundary latitude.
- **end_latitude** (*float*) – Northern Boundary Latitude.

Returns smaller cube.

Return type iris.cube.Cube

esmvalcore.preprocessor.**extract_season**(*cube*, *season*)

Slice cube to get only the data belonging to a specific season.

Parameters

- **cube** (*iris.cube.Cube*) – Original data

- **season** (*str*) – Season to extract. Available: DJF, MAM, JJA, SON

Returns data cube for specified season.

Return type iris.cube.Cube

esmvalcore.preprocessor.**extract_shape**(*cube*, *shapefile*, *method='contains'*, *crop=True*, *decomposed=False*)

Extract a region defined by a shapefile.

Note that this function does not work for shapes crossing the prime meridian or poles.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **shapefile** (*str*) – A shapefile defining the region(s) to extract.
- **method** (*str, optional*) – Select all points contained by the shape or select a single representative point. Choose either 'contains' or 'representative'. If 'contains' is used, but not a single grid point is contained by the shape, a representative point will selected.
- **crop** (*bool, optional*) – Crop the resulting cube using *extract_region()*. Note that data on irregular grids will not be cropped.
- **decomposed** (*bool, optional*) – Whether or not to retain the sub shapes of the shapefile in the output. If this is set to True, the output cube has a dimension for the sub shapes.

Returns Cube containing the extracted region.

Return type iris.cube.Cube

See also:

extract_region() Extract a region from a cube.

esmvalcore.preprocessor.**extract_time**(*cube*, *start_year*, *start_month*, *start_day*, *end_year*, *end_month*, *end_day*)

Extract a time range from a cube.

Given a time range passed in as a series of years, months and days, it returns a time-extracted cube with data only within the specified time range.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **start_year** (*int*) – start year
- **start_month** (*int*) – start month
- **start_day** (*int*) – start day
- **end_year** (*int*) – end year
- **end_month** (*int*) – end month
- **end_day** (*int*) – end day

Returns Sliced cube.

Return type iris.cube.Cube

Raises **ValueError** – if time ranges are outside the cube time limits

esmvalcore.preprocessor.**extract_trajectory**(*cube*, *latitudes*, *longitudes*, *number_points=2*)

Extract data along a trajectory.

latitudes and longitudes are the pairs of coordinates for two points. number_points is the number of points between the two points.

This version uses the expensive interpolate method, but it may be necceasiry for irregular grids.

If only two latitude and longitude coordinates are given, extract_trajectory will produce a cube will extrapolate along a line between those two points, and will add *number_points* points between the two corners.

If more than two points are provided, then extract_trajectory will produce a cube which has extrapolated the data of the cube to those points, and *number_points* is not needed.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **latitudes** (*list*) – list of latitude coordinates (floats).
- **longitudes** (*list*) – list of longitude coordinates (floats).
- **number_points** (*int*) – number of points to extrapolate (optional).

Returns collapsed cube.

Return type iris.cube.Cube

Raises **ValueError** – if latitude and longitude have different dimensions.

esmvalcore.preprocessor.**extract_transect**(*cube*, *latitude=None*, *longitude=None*)

Extract data along a line of constant latitude or longitude.

Both arguments, latitude and longitude, are treated identically. Either argument can be a single float, or a pair of floats, or can be left empty. The single float indicates the latitude or longitude along which the transect should be extracted. A pair of floats indicate the range that the transect should be extracted along the secondairy axis.

For instance *'extract_transect(cube, longitude=-28)'* will produce a transect along 28 West.

Also, *'extract_transect(cube, longitude=-28, latitude=[-50, 50])'* will produce a transect along 28 West between 50 south and 50 North.

This function is not yet implemented for irregular arrays - instead try the extract_trajectory function, but note that it is currently very slow. Alternatively, use the regrid preprocessor to regrid along a regular grid and then extract the transect.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **latitude** (*None, float or [float, float], optional*) – transect latiude or range.
- **longitude** (*None, float or [float, float], optional*) – transect longitude or range.

Returns collapsed cube.

Return type iris.cube.Cube

Raises

- **ValueError** – slice extraction not implemented for irregular grids.
- **ValueError** – latitude and longitude are both floats or lists; not allowed to slice on both axes at the same time.

esmvalcore.preprocessor.**extract_volume**(*cube*, *z_min*, *z_max*)

Subset a cube based on a range of values in the z-coordinate.

Function that subsets a cube on a box (z_min, z_max) This function is a restriction of masked_cube_lonlat(); Note that this requires the requested z-coordinate range to be the same sign as the iris cube. ie, if the cube has z-coordinate as negative, then z_min and z_max need to be negative numbers.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **z_min** (*float*) – minimum depth to extract.
- **z_max** (*float*) – maximum depth to extract.

Returns z-coord extracted cube.

Return type iris.cube.Cube

esmvalcore.preprocessor.**fix_data**(*cube*, *short_name*, *project*, *dataset*, *mip*, *frequency=None*, *check_level=<CheckLevels.DEFAULT: 3>*)

Fix cube data if fixes add present and check it anyway.

This method assumes that metadata is already fixed and checked.

This method collects all the relevant fixes for a given variable, applies them and checks resulting cube (or the original if no fixes were needed) metadata to ensure that it complies with the standards of its project CMOR tables.

Parameters

- **cube** (*iris.cube.Cube*) – Cube to fix
- **short_name** (*str*) – Variable's short name
- **project** (*str*) –
- **dataset** (*str*) –
- **mip** (*str*) – Variable's MIP
- **frequency** (*str, optional*) – Variable's data frequency, if available
- **check_level** (CheckLevels) – Level of strictness of the checks. Set to default.

Returns Fixed and checked cube

Return type iris.cube.Cube

Raises ***CMORCheckError*** – If the checker detects errors in the data that it can not fix.

esmvalcore.preprocessor.**fix_file**(*file*, *short_name*, *project*, *dataset*, *mip*, *output_dir*)

Fix files before ESMValTool can load them.

This fixes are only for issues that prevent iris from loading the cube or that cannot be fixed after the cube is loaded.

Original files are not overwritten.

Parameters

- **file** (*str*) – Path to the original file
- **short_name** (*str*) – Variable's short name
- **project** (*str*) –
- **dataset** (*str*) –

- **output_dir** (*str*) – Output directory for fixed files

Returns Path to the fixed file

Return type str

esmvalcore.preprocessor.**fix_metadata**(*cubes*, *short_name*, *project*, *dataset*, *mip*, *frequency=None*, *check_level=<CheckLevels.DEFAULT: 3>*)

Fix cube metadata if fixes are required and check it anyway.

This method collects all the relevant fixes for a given variable, applies them and checks the resulting cube (or the original if no fixes were needed) metadata to ensure that it complies with the standards of its project CMOR tables.

Parameters

- **cubes** (*iris.cube.CubeList*) – Cubes to fix
- **short_name** (*str*) – Variable's short name
- **project** (*str*) –
- **dataset** (*str*) –
- **mip** (*str*) – Variable's MIP
- **frequency** (*str, optional*) – Variable's data frequency, if available
- **check_level** (CheckLevels) – Level of strictness of the checks. Set to default.

Returns Fixed and checked cube

Return type iris.cube.Cube

Raises ***CMORCheckError*** – If the checker detects errors in the metadata that it can not fix.

esmvalcore.preprocessor.**load**(*file*, *callback=None*)

Load iris cubes from files.

esmvalcore.preprocessor.**mask_above_threshold**(*cube*, *threshold*)

Mask above a specific threshold value.

Takes a value 'threshold' and masks off anything that is above it in the cube data. Values equal to the threshold are not masked.

Parameters

- **cube** (*iris.cube.Cube*) – iris cube to be thresholded.
- **threshold** (*float*) – threshold to be applied on input cube data.

Returns thresholded cube.

Return type iris.cube.Cube

esmvalcore.preprocessor.**mask_below_threshold**(*cube*, *threshold*)

Mask below a specific threshold value.

Takes a value 'threshold' and masks off anything that is below it in the cube data. Values equal to the threshold are not masked.

Parameters

- **cube** (*iris.cube.Cube*) – iris cube to be thresholded
- **threshold** (*float*) – threshold to be applied on input cube data.

Returns thresholded cube.

Return type iris.cube.Cube

esmvalcore.preprocessor.**mask_fillvalues**(*products*, *threshold_fraction*, *min_value=None*, *time_window=1*)

Compute and apply a multi-dataset fillvalues mask.

Construct the mask that fills a certain time window with missing values if the number of values in that specific window is less than a given fractional threshold. This function is the extension of _get_fillvalues_mask and performs the combination of missing values masks from each model (of multimodels) into a single fillvalues mask to be applied to each model.

Parameters

- **products** (*iris.cube.Cube*) – data products to be masked.
- **threshold_fraction** (*float*) – fractional threshold to be used as argument for Aggregator. Must be between 0 and 1.
- **min_value** (*float*) – minimum value threshold; default None If default, no thresholding applied so the full mask will be selected.
- **time_window** (*float*) – time window to compute missing data counts; default set to 1.

Returns Masked iris cubes.

Return type iris.cube.Cube

Raises **NotImplementedError** – Implementation missing for data with higher dimensionality than 4.

esmvalcore.preprocessor.**mask_glaciated**(*cube*, *mask_out*)

Mask out glaciated areas.

It applies a Natural Earth mask. Note that for computational reasons only the 10 largest polygons are used for masking.

Parameters

- **cube** (*iris.cube.Cube*) – data cube to be masked.
- **mask_out** (*str*) – "glaciated" to mask out glaciated areas

Returns Returns the masked iris cube.

Return type iris.cube.Cube

Raises **ValueError** – Error raised if masking on irregular grids is attempted or if mask_out has a wrong value.

esmvalcore.preprocessor.**mask_inside_range**(*cube*, *minimum*, *maximum*)

Mask inside a specific threshold range.

Takes a MINIMUM and a MAXIMUM value for the range, and masks off anything that's between the two in the cube data.

Parameters

- **cube** (*iris.cube.Cube*) – iris cube to be thresholded
- **minimum** (*float*) – lower threshold to be applied on input cube data.
- **maximum** (*float*) – upper threshold to be applied on input cube data.

Returns thresholded cube.

Return type iris.cube.Cube

esmvalcore.preprocessor.**mask_landsea**(*cube*, *fx_variables*, *mask_out*, *always_use_ne_mask=False*)

Mask out either land mass or sea (oceans, seas and lakes).

It uses dedicated fx files (sftlf or sftof) or, in their absence, it applies a Natural Earth mask (land or ocean contours). Note that the Natural Earth masks have different resolutions: 10m for land, and 50m for seas; these are more than enough for ESMValTool puprpose.

Parameters

- **cube** (*iris.cube.Cube*) – data cube to be masked.
- **fx_variables** (*dict*) – dict: keys: fx variables, values: full paths to fx files.
- **mask_out** (*str*) – either "land" to mask out land mass or "sea" to mask out seas.
- **always_use_ne_mask** (*bool, optional (default: False)*) – always apply Natural Earths mask, regardless if fx files are available or not.

Returns Returns the masked iris cube.

Return type iris.cube.Cube

Raises **ValueError** – Error raised if masking on irregular grids is attempted. Irregular grids are not currently supported for masking with Natural Earth shapefile masks.

esmvalcore.preprocessor.**mask_landseaice**(*cube*, *fx_variables*, *mask_out*)

Mask out either landsea (combined) or ice.

Function that masks out either landsea (land and seas) or ice (Antarctica and Greenland and some wee glaciers). It uses dedicated fx files (sftgif).

Parameters

- **cube** (*iris.cube.Cube*) – data cube to be masked.
- **fx_variables** (*dict*) – dict: keys: fx variables, values: full paths to fx files.
- **mask_out** (*str*) – either "landsea" to mask out landsea or "ice" to mask out ice.

Returns Returns the masked iris cube with either land or ice masked out.

Return type iris.cube.Cube

Raises

- **ValueError** – Error raised if fx mask and data have different dimensions.
- **ValueError** – Error raised if fx files list is empty.

esmvalcore.preprocessor.**mask_outside_range**(*cube*, *minimum*, *maximum*)

Mask outside a specific threshold range.

Takes a MINIMUM and a MAXIMUM value for the range, and masks off anything that's outside the two in the cube data.

Parameters

- **cube** (*iris.cube.Cube*) – iris cube to be thresholded
- **minimum** (*float*) – lower threshold to be applied on input cube data.
- **maximum** (*float*) – upper threshold to be applied on input cube data.

Returns thresholded cube.

Return type iris.cube.Cube

esmvalcore.preprocessor.**meridional_statistics**(*cube*, *operator*)
Compute meridional statistics.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **operator** (*str, optional*) – Select operator to apply. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max'.

Returns Meridional statistics cube.

Return type iris.cube.Cube

Raises **ValueError** – Error raised if computation on irregular grids is attempted. Zonal statistics not yet implemented for irregular grids.

esmvalcore.preprocessor.**monthly_statistics**(*cube*, *operator='mean'*)
Compute monthly statistics.

Chunks time in monthly periods and computes statistics over them;

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **operator** (*str, optional*) – Select operator to apply. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max'

Returns Monthly statistics cube

Return type iris.cube.Cube

esmvalcore.preprocessor.**multi_model_statistics**(*products*, *span*, *statistics*, *output_products=None*)
Compute multi-model statistics.

Multimodel statistics computed along the time axis. Can be computed across a common overlap in time (set span: overlap) or across the full length in time of each model (set span: full). Restrictive computation is also available by excluding any set of models that the user will not want to include in the statistics (set exclude: [excluded models list]).

Restrictions needed by the input data: - model datasets must have consistent shapes, - higher dimensional data is not supported (ie dims higher than four: time, vertical axis, two horizontal axes).

Parameters

- **products** (*list*) – list of data products or cubes to be used in multimodel stat computation; cube attribute of product is the data cube for computing the stats.
- **span** (*str*) – overlap or full; if overlap, statitsticss are computed on common time- span; if full, statistics are computed on full time spans, ignoring missing data.
- **output_products** (*dict*) – dictionary of output products.
- **statistics** (*str*) – statistical measure to be computed. Available options: mean, median, max, min, std, or pXX.YY (for percentile XX.YY; decimal part optional).

Returns list of data products or cubes containing the multimodel stats computed.

Return type list

Raises **ValueError** – If span is neither overlap nor full.

esmvalcore.preprocessor.**regrid**(*cube*, *target_grid*, *scheme*, *lat_offset=True*, *lon_offset=True*)
Perform horizontal regridding.

Parameters

- **cube** (*cube*) – The source cube to be regridded.
- **target_grid** (*cube or str*) – The cube that specifies the target or reference grid for the regridding operation. Alternatively, a string cell specification may be provided, of the form 'MxN', which specifies the extent of the cell, longitude by latitude (degrees) for a global, regular target grid.
- **scheme** (*str*) – The regridding scheme to perform, choose from 'linear', 'linear_extrapolate', 'nearest', 'area_weighted', 'unstructured_nearest'.
- **lat_offset** (*bool*) – Offset the grid centers of the latitude coordinate w.r.t. the pole by half a grid step. This argument is ignored if *target_grid* is a cube or file.
- **lon_offset** (*bool*) – Offset the grid centers of the longitude coordinate w.r.t. Greenwich meridian by half a grid step. This argument is ignored if *target_grid* is a cube or file.

Returns

Return type cube

See also:

extract_levels() Perform vertical regridding.

esmvalcore.preprocessor.**regrid_time**(*cube*, *frequency*)

Align time axis for cubes so they can be subtracted.

Operations on time units, time points and auxiliary coordinates so that any cube from cubes can be subtracted from any other cube from cubes. Currently this function supports yearly (frequency=yr), monthly (frequency=mon), daily (frequency=day), 6-hourly (frequency=6hr), 3-hourly (frequency=3hr) and hourly (frequency=1hr) data time frequencies.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **frequency** (*str*) – data frequency: mon, day, 1hr, 3hr or 6hr

Returns cube with converted time axis and units.

Return type iris.cube.Cube

esmvalcore.preprocessor.**save**(*cubes*, *filename*, *optimize_access=''*, *compress=False*, ***kwargs*)

Save iris cubes to file.

Parameters

- **cubes** (*iterable of iris.cube.Cube*) – Data cubes to be saved
- **filename** (*str*) – Name of target file
- **optimize_access** (*str*) – Set internal NetCDF chunking to favour a reading scheme

  Values can be map or timeseries, which improve performance when reading the file one map or time series at a time. Users can also provide a coordinate or a list of coordinates. In that case the better performance will be avhieved by loading all the values in that coordinate at a time
- **compress** (*bool, optional*) – Use NetCDF internal compression.

Returns filename

Return type str

esmvalcore.preprocessor.**seasonal_statistics**(*cube*, *operator='mean'*)
Compute seasonal statistics.

Chunks time in 3-month periods and computes statistics over them;

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **operator** (*str, optional*) – Select operator to apply. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max'

Returns Seasonal statistic cube

Return type iris.cube.Cube

esmvalcore.preprocessor.**timeseries_filter**(*cube*, *window*, *span*, *filter_type='lowpass'*, *filter_stats='sum'*)
Apply a timeseries filter.

Method borrowed from iris example

Apply each filter using the rolling_window method used with the weights keyword argument. A weighted sum is required because the magnitude of the weights are just as important as their relative sizes.

See also the iris rolling window

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **window** (*int*) – The length of the filter window (in units of cube time coordinate).
- **span** (*int*) – Number of months/days (depending on data frequency) on which weights should be computed e.g. 2-yearly: span = 24 (2 x 12 months). Span should have same units as cube time coordinate.
- **filter_type** (*str, optional*) – Type of filter to be applied; default 'lowpass'. Available types: 'lowpass'.
- **filter_stats** (*str, optional*) – Type of statistic to aggregate on the rolling window; default 'sum'. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max'

Returns cube time-filtered using 'rolling_window'.

Return type iris.cube.Cube

:raises iris.exceptions.CoordinateNotFoundError:: Cube does not have time coordinate. :raises NotImplementedError:: If filter_type is not implemented.

esmvalcore.preprocessor.**volume_statistics**(*cube*, *operator*, *fx_variables=None*)
Apply a statistical operation over a volume.

The volume average is weighted according to the cell volume. Cell volume is calculated from iris's cartography tool multiplied by the cell thickness.

Parameters

- **cube** (*iris.cube.Cube*) – Input cube.
- **operator** (*str*) – The operation to apply to the cube, options are: 'mean'.
- **fx_variables** (*dict*) – dictionary of field:filename for the fx_variables

Returns collapsed cube.

Return type iris.cube.Cube

Raises **ValueError** – if input cube shape differs from grid volume cube shape.

esmvalcore.preprocessor.**weighting_landsea_fraction**(*cube*, *fx_variables*, *area_type*)

Weight fields using land or sea fraction.

This preprocessor function weights a field with its corresponding land or sea area fraction (value between 0 and 1). The application of this is important for most carbon cycle variables (and other land-surface outputs), which are e.g. reported in units of *kgC m-2*. This actually refers to 'per square meter of land/sea' and NOT 'per square meter of gridbox'. So in order to integrate these globally or regionally one has to both area-weight the quantity but also weight by the land/sea fraction.

Parameters

- **cube** (*iris.cube.Cube*) – Data cube to be weighted.
- **fx_variables** (*dict*) – Dictionary holding `var_name` (keys) and full paths (values) to the fx files as `str` or empty `list` (if not available).
- **area_type** (*str*) – Use land (`'land'`) or sea (`'sea'`) fraction for weighting.

Returns Land/sea fraction weighted cube.

Return type iris.cube.Cube

Raises

- **TypeError** – `area_type` is not `'land'` or `'sea'`.
- **ValueError** – Land/sea fraction variables `sftlf` or `sftof` not found or shape of them is not broadcastable to `cube`.

esmvalcore.preprocessor.**zonal_statistics**(*cube*, *operator*)

Compute zonal statistics.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **operator** (*str, optional*) – Select operator to apply. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max'.

Returns Zonal statistics cube.

Return type iris.cube.Cube

Raises **ValueError** – Error raised if computation on irregular grids is attempted. Zonal statistics not yet implemented for irregular grids.

Part VII

Changelog

CHAPTER

TWENTYTWO

V2.0.0

This release includes

22.1 Bug fixes

- Fixed derivation of co2s (#594) Manuel Schlund
- Padding while cropping needs to stay within sane bounds for shapefiles that span the whole Earth (#626) Valeriu Predoi
- Fix concatenation of a single cube (#655) Bouwe Andela
- Fix mask fx dict handling not to fail if empty list in values (#661) Valeriu Predoi
- Preserve metadata during anomalies computation when using iris cubes difference (#652) Valeriu Predoi
- Avoid crashing when there is directory 'esmvaltool' in the current working directory (#672) Valeriu Predoi
- Solve bug in ACCESS1 dataset fix for calendar. (#671) Peter Kalverla
- Fix the syntax for adding multiple ensemble members from the same dataset (#678) SarahAlidoost
- Fix bug that made preprocessor with fx files fail in rare cases (#670) Manuel Schlund
- Add support for string coordinates (#657) Javier Vegas-Regidor
- Fixed the shape extraction to account for wraparound shapefile coords (#319) Valeriu Predoi
- Fixed bug in time weights calculation (#695) Manuel Schlund
- Fix diagnostic filter (#713) Javier Vegas-Regidor

22.2 Documentation

- Add pandas as a requirement for building the documentation (#607) Bouwe Andela
- Document default order in which preprocessor functions are applied (#633) Bouwe Andela
- Add pointers about data loading and CF standards to documentation (#571) Valeriu Predoi
- Config file populated with site-specific data paths examples (#619) Valeriu Predoi
- Update Codacy badges (#643) Bouwe Andela
- Update copyright info on readthedocs (#668) Bouwe Andela
- Updated references to documentation (now docs.esmvaltool.org) (#675) Axel Lauer

- Add all European grants to Zenodo (#680) Bouwe Andela
- Update Sphinx to v3 or later (#683) Bouwe Andela
- Increase version to 2.0.0 and add release notes (#691) Bouwe Andela
- Update setup.py and README.md for use on PyPI (#693) Bouwe Andela
- Suggested Documentation changes (#690) Steve Smith

22.3 Improvements

- Reduce the size of conda package (#606) Bouwe Andela
- Add a few unit tests for DiagnosticTask (#613) Bouwe Andela
- Make ncl or R tests not fail if package not installed (#610) Valeriu Predoi
- Pin flake8<3.8.0 (#623) Valeriu Predoi
- Log warnings for likely errors in provenance record (#592) Bouwe Andela
- Unpin flake8 (#646) Bouwe Andela
- More flexible native6 default DRS (#645) Bouwe Andela
- Try to use the same python for running diagnostics as for esmvaltool (#656) Bouwe Andela
- Fix test for lower python version and add note on lxml (#659) Valeriu Predoi
- Added 1m deep average soil moisture variable (#664) bascrezee
- Update docker recipe (#603) Javier Vegas-Regidor
- Improve command line interface (#605) Javier Vegas-Regidor
- Remove utils directory (#697) Bouwe Andela
- Avoid pytest version that crashes (#707) Bouwe Andela
- Options arg in read_config_user_file now optional (#716) Javier Vegas-Regidor
- Produce a readable warning if ancestors are a string instead of a list. (#711) katjaweigel
- Pin Yamale to v2 (#718) Bouwe Andela
- Expanded cmor public API (#714) Manuel Schlund

22.4 Fixes for datasets

- Added various fixes for hybrid height coordinates (#562) Manuel Schlund
- Extended fix for cl-like variables of CESM2 models (#604) Manuel Schlund
- Added fix to convert "geopotential" to "geopotential height" for ERA5 (#640) Evgenia Galytska
- Do not fix longitude values if they are too far from valid range (#636) Javier Vegas-Regidor

22.5 Preprocessor

- Implemented concatenation of cubes with derived coordinates (#546) Manuel Schlund
- Fix derived variable ctotal calculation depending on project and standard name (#620) Valeriu Predoi
- State of the art FX variables handling without preprocessing (#557) Valeriu Predoi
- Add max, min and std operators to multimodel (#602) Javier Vegas-Regidor
- Added preprocessor to extract amplitude of cycles (#597) Manuel Schlund
- Overhaul concatenation and allow for correct concatenation of multiple overlapping datasets (#615) Valeriu Predoi
- Change volume stats to handle and output masked array result (#618) Valeriu Predoi
- Area_weights for cordex in area_statistics (#631) mwjury
- Accept cubes as input in multimodel (#637) sloosvel
- Make multimodel work correctly with yearly data (#677) Valeriu Predoi
- Optimize time weights in time preprocessor for climate statistics (#684) Valeriu Predoi
- Add percentiles to multi-model stats (#679) Peter Kalverla

CHAPTER

TWENTYTHREE

V2.0.0B9

This release includes

23.1 Bug fixes

- Cast dtype float32 to output from zonal and meridional area preprocessors (#581) Valeriu Predoi

23.2 Improvements

- Unpin on Python<3.8 for conda package (run) (#570) Valeriu Predoi
- Update pytest installation marker (#572) Bouwe Andela
- Remove vmrh2o (#573) Mattia Righi
- Restructure documentation (#575) Bouwe Andela
- Fix mask in land variables for CCSM4 (#579) Klaus Zimmermann
- Fix derive scripts wrt required method (#585) Klaus Zimmermann
- Check coordinates do not have repeated standard names (#558) Javier Vegas-Regidor
- Added derivation script for co2s (#587) Manuel Schlund
- Adapted custom co2s table to match CMIP6 version (#588) Manuel Schlund
- Increase version to v2.0.0b9 (#593) Bouwe Andela
- Add a method to save citation information (#402) SarahAlidoost

For older releases, see the release notes on https://github.com/ESMValGroup/ESMValCore/releases.

Part VIII

Indices and tables

- genindex
- search

PYTHON MODULE INDEX

e

INDEX

F

G

H

I

J

K

L

M

W

Z