pyCMBS Documentation

Release 1.0.0-dev

Alexander Loew

May 11, 2015

CHAPTER 1

Introduction

pyCMBS is a simple and flexible toolkit to perform quick analysis of geospatial data, visualize it as well as to perform benchmarking of climate model simulations. Its a very lightweight and powerful tool, easy to install and to use. pyCMBS comes with full python power included.

1.1 Why to use?

Why and when should you use pyCMBS?

Geospatial data analysis ...

- you work with geospatial data
- you work a lot with netCDF files
- you analyze timeseries
- you do geostatistical analysis
- you want to generate nice looking maps
- you are bored in re-coding you own scripts, but want to increase your scientific throughput using a powerfull tool just using a few lines of code?

Model benchmarking ... the easy way!

- you are widely using models or are developing models?
- you want to know how good your model performs
- you search for a flexible tool for automated model performance metrics, comparing e.g. different models or model versions against control simulations or observational datasets

Some of the features pyCMBS provides for model benchmarking are

- automated and efficient pre-processing and regridding of model data and observations in a standardized way
- standard, modular and flexible libraries for plots and diagnostics
- automatic report generation with different levels of detail
- comparison against up to four different observational datasets at the same time
- user friendly configuration of plots and processing
- state-of-the-art model skill scores

CHAPTER 2

General concept

pyCMBS consists of two parts

1. core part: provides a core library for data analysis and plotting. This can be used to write powerful data analysis scripts.
2. benchmarking framework: easy and efficient framework for benchmarking multimodel output.

2.1 Coding principles

pyCMBS is written purely in Python (2.7). The coding is based largely on the concept of the test driven development. The code has the following characteristics

- thorough testing using unittests
- readable python code, following PEP8 convention
- object oriented programming, flexibility and modularity
- pyCMBS is an open source project and is hosted on github

CHAPTER 3

Installation

Below we describe how you properly install pyCMBS in your environment. Currently, there are three different ways to install pyCMBS

1. NOT RECOMMENDED AT THE MOMENT Easy installation using *pip* (note: not checked for a while be careful!)
2. Source code installation from code repository (currently recommended)
3. Source code installation from tarball

All approaches are detailed below.

Special informations for users working at the Max-Planck-Institute for Meteorology are provided in a `installation_mpi`.

3.1 Operating systems

pyCMBS is developed purely in python and is expected to be in general independent from a special operating system. The current version was however only developed on Linux and no tests at all were made on other operating systems.

3.2 Dependencies

pyCMBS was built to have only a small number of external dependencies. However, there is a minimum number of dependencies existing which are a required for using pyCMBS sucessfully.

Core python packages [obligatory]

- python 2.7.x
- matplotlib >v1.3.x
- numpy
- scipy >0.11

Note that you should really ensure that your matplotlib installation is greater than version 1.3, as otherwise some features will not work. Standard packages like e.g. shipped as standard with Ubuntu have smaller version numbers (e.g. v1.1.1).

Compatability with python 3.x is not supported yet

Obligatory additional dependencies

For file I/O of netCDF files, a single library is supported at the moment:

- netCDF4 library is used as default for file I/O [recommended]

For an efficient data pre-processing the climate data operators are used. The core CDO's and the corresponding python wrapper is required.

- climate data operators (cdo) [obligatory]
- cdo python interface [obligatory, can be easily installed using pip]

Recommended dependencies

For plotting projected data (map plots), pyCMBS currently supports two different plotting backends. These are

- cartopy [recommended]
- matplotlib basemap [optional]
- yaml [required if you want to use plugins or do benchmarking]

3.3 Installation of dependencies

For convenience, we have put together an installation procedure to install all required dependencies (except the ones needed for cartopy). Please find it *here* for your convenience. An `installation_checklist` summarizing the required packages is available as well.

3.4 Quick installation from scratch for experts

If you have not yet any of the above dependencies installed and are working on a Debian like operating system (e.g. Ubunutu), the easiest way to install pyCMBS is by executing the installation sequence which can be found in the file *.travis.yml* in the root directory of the source code.

The developers are using automatic code checking and builds and the installation sequence in the file *.travis.yml* is used to setup for each build a working installation.

3.5 Detailed installation instructions for pyCMBS

In the following, we will summarize the different approaches to install pyCMBS.

3.5.1 Installation using *pip* (the easiest way ... theoretically)

NOT RECOMMENDED AT THE MOMENT AS NOT TESTED!!! USE INSTALLATION FROM GITHIB REPOSITORY INSTEAD!!!

Using pip is in general the easiest way to install pyCMBS if you dont want to develop by your own.

NOTICE however that this installation procedure was not tested thoroughly with present version. Please be therefore careful.

The installation with *pip* was tested without the cartopy plotting backend so far.

If you have *pip* not yet installed on your machine, then the first step is to install pip .

Once *pip* is installed, the installation of pyCMBS is as simple as a two-liner:

```
pip install numpy
pip install pycmbs
```

Note that the *numpy* module needs to be installed separately before as otherwise the installation fails. The command *pip install pycmbs* will then install all remaining dependencies.

Check if everything is working, like described *below*.

3.5.2 Installation of stable version from a *tarball*

You can install specific version of pycmbs from a *tarball* from the download site . All dependencies must have been installed before.

After you have obtained the pyCMBS code from a *tarball*, then first extract the archive into some new directory:

```
mkdir temp_dir
cd temp_dir
tar -xvf pycmbs-vx.x.x.tar.gz
```

This expands the tar file and you obtain a subdirectory, called *pycmbs-vx.x.x*

To install the package first change to the directory and then install the package using the standard python setup tools as:

```
cd pycmbs-vx.x.x
python setup.py install
```

This will install the package in your python environment. Check successful installation, like described *below*.

3.5.3 github repository (for developers)

Assuming that you want to contribute to the development of the pyCMBS, follow the instructions to create your fork. All dependencies must have been installed and configured properly (check section Installation of dependencies above). To retrieve the code into your development environment execute the following command (replace your_user_name with the user name you registered with github):

```
git clone https://github.com/your_user_name/pycmbs.git
```

Some sub-modules are written in cython to speed up processing. These modules need to be compiled prior to the final installation. This is done by just executing the following command:

```
# compile cython code
sh compile_extensions.py
```

Final installation

Now you have in principle two options. You either decide that the code should be installed in the python dist-packages directory, then you do:

```
# do installation
python setup.py install
```

or if you want to hack the code, it is highly recommended to simply set the right PYTHONPATH environment variable:

```
# or as an alternative for developers, just set the PYTHONPATH
# environment variable to the pycmbs root directory and also adapt
# you systempath (PATH) such that includes the pycmbs rootdirectory
export PYTHONPATH=/my/path/to/pycmbs:$PYTHONPATH
```

Check successful installation, like described *below*.

3.6 Installation of dependencies

Please find here a working installation procedure, which was tested under Ubuntu 32-bit. It installs all pyCMBS dependencies, except the ones needed for cartopy and installs pyCMBS itself.:

```
#!/usr/bin/env bash

#
# This file provides an installing procedure for pyCMBS WITHOUT Cartopy support
#    it was tested for ubuntu32
#

# update package database
apt-get update

#####################################################################
# DEPENDENCIES
#####################################################################

# the -qq option installs silent using defaults
apt-get -qq install texlive-latex-base texlive-latex-extra texlive-latex-recommended
apt-get -qq install python-pip python-dev
apt-get -qq install cdo libhdf5-openmpi-dev libnetcdf-dev libopenmpi-dev
apt-get -qq install python-numpy
apt-get -qq install cython
C_INCLUDE_PATH=/usr/include/mpi pip install netCDF4

# apt-get -qq install python-matplotlib  # this gives the system default package, which is currently
# it is highly recommended to use matplotlib > 1.3
apt-get -qq install libfreetype6-dev libpng-dev  # required for matplotlib
sudo easy_install -U distribute
sudo pip install https://downloads.sourceforge.net/project/matplotlib/matplotlib/matplotlib-1.3.1/mat
apt-get -qq install python-mpltoolkits.basemap
apt-get -qq install python-mpltoolkits.basemap-data

apt-get -qq install python-scipy
pip install pyshp

#####################################################################
# pycmbs
#####################################################################
pip install --upgrade pycmbs

#####################################################################
# test environment
#####################################################################
pip install nose

echo "Now you can run the unittests as follows:"
echo "    cd /usr/local/lib/python2.7/dist-packages/pycmbs/tests"
echo "    nosetests"
```

3.7 Final check of installation

Check that installation worked properly by going through the following checklist. In case of problems, please refer to the *troublesolver* .

Is the pyCMBS python module loaded properly?:

```
python -c "from pycmbs import *; print('Welcome to pyCMBS')"
```

This should give you a short welcome message, but no error messages.

Is the benchmarking script working properly?:

```
pycmbs-benchmarking.py
```

This will you give a short message like:

```
*******************************************
* WELCOME to pycmbs.py                    *
* Happy benchmarking ...                  *
*******************************************
```

and will end with an error message that the configuration file is not found (this is o.k.)

If you see the above, the installation has worked! Congratulations!

3. Check also the proper installation of the cdo's and the cdo.py interface, as this is a prerequesite of beeing able to properly work with pyCMBS:

```
python -c "from cdo import *; cdo=Cdo(); print 'If you see this, everything went right ... have fun w
```

Again, this should give you a short welcome message. Any error message is a bad sign. In that case, please check your installation again. Have a look at the *troublesolver*.

pycmbs init

3.8 Running tests

pyCMBS code comes with a rich suite of test routines. We follow the concept of unittests using the nosetests tools. Tests should be always executed in the following cases:

- after installation
- before comitting code to the repository
- before merging a branch in the master branch

Tests can be simply executed using the *Makefile* in the main installation directory as:

```
make tests
```

As an alternative you can also check the coverage of tests in the code using:

```
make coverage
```

which gives you a report on test coverage in */coverage/index.html*.

3.9 Further information and trouble solving

pyCMBS makes use of a standard directory to look for observations. This directory is the Standard Evaluation Pool (SEP). The path to the SEP directory needs to be specified in the $SEP environment variable. In you .bashrc write:

```
export SEP=/path/to/directory
```

For users at MPI-M, the SEP variable needs to point to */pool/SEP*. It is however possible to specify also for each observation an individual path where the observation is located. Then the SEP evnironment variable is not required. To check whether SEP is set, type:

```
echo $SEP
```

3.9.1 Some hints for trouble solving

If your pyCMBS installation seems not to work properly, here are a few recommendations where to start searching.

Is python working properly?:

```
python -c "print 'Hello world'"
```

Does your PYTHONPATH environment variable contain the path to pyCMBS?:

```
echo $PYTHONPATH
```

This should give you the path where python is searching for modules. If it is empty you are most likely in trouble. Check if you have a valid python installation.

Is the script pycmb-benchmarking.py found in the system path?:

```
pycmbs-benchmarking.py
```

should give you a short Welcome Screen like described above. If this is not the case then either the overall pyCMBS installation is incomplete or Your systempath is not set appropriately. Type:

```
echo $PATH
```

and verify if the directory where pycmbs-benchmarking.py is located is listed in your PATH variable. If not, then you can try to change your PATH variable to make it working.

Further problems?

In case that these recommendations did not solve your problem, please feel free to ask a question or raise an issue on the pyCMBS development site.

CHAPTER 4

Getting started

There are several ways to get started with pyCMBS.

1. pyCMBS gallery
2. a first pycmbs session with example data is provided in an ipython notebook in the *demo* folder.
3. examples provided together with the source code
4. in code documentation. Use the python help() to view the docstrings.

CHAPTER 5

pyCMBS gallery

In the following, examples will be given that introduce different features of pyCMBS.

5.1 Basic plotting

For plotting you have a rich suite of keyword parameters. Please use the python help() system to get full documentation from docstrings. Some more illustrations of options are provided:

```
map_plot(air,show_timeseries=True, use_basemap=True,title='show_timeseries=True')
map_plot(air,show_zonal=True, use_basemap=True,title='show_zonal=True')
map_plot(air,show_histogram=True, use_basemap=True,title='show_histogram=True')
```

And a few more details on customizing your map ...:

```
map_plot(air, use_basemap=True, title='vmin=-30.,vmax=30.,cmap_data=RdBu_r', vmin=-30., vmax=30., cma
map_plot(air, contours=True, use_basemap=True, levels=np.arange(-50.,50.,10.), title='contours=True',
map_plot(air, show_stat=True, use_basemap=True,title='show_stat=True',ax=ax3)
map_plot(air, show_stat=True, stat_type='median', use_basemap=True, title='show_stat=True,stat_type='
```

5.2 Basic data analysis

5.2.1 Temporal mean

Calculating the temporal mean field of a variable is as simple as:

```
air.timmean()
```

5.2.2 Spatial mean

The (area weighted) spatial mean is obtained as:

```
air.fldmean()
```

5.2.3 Masking an area

You probably want to work only on particular regions. The following script shows you how to easily to this.

5.2.4 Temporal slicing

If you want to perform a temporal subsetting of the data, this can be done as follows:

```
# temporal subsetting using existing start/stop dates
import datetime.datetime

start_date = datetime(2001,05,01)
stop_date = datetime(2010,04,15)
air.apply_temporal_subsetting(start_date, stop_date):
```

5.3 Working with multiple datasets

5.3.1 Simple arithmetic operations

Model benchmarking

pyCMBS provides a flexible tool to compare output from models with other models as well as with observational data.

6.1 Your first benchmarking

The main input for model benchmarking is model data and observational datasets as well as a user specified configuration. After running the benchmarking, you get a report (PDF format) as well as a lot of figures and statistics which contain usefull information. The benchmarking is based ona modular approach and allows the user to customize the results by activating and deactivating specific components.

The next steps will guide you through a benchmarking session to get you started.

1. set up a working directory and change to it:

```
# set up a working directory somewhere on your machine
mkdir -p ~/temp/my_first_benchmarking
cd ~/temp/my_first_benchmarking
```

2. set up an initial configuration:

```
# simply run the pycmbs-benchmarking.py script
# if properly installed, you should be able to just run it from the console
pycmbs-benchmarking.py init
```

This gives you:

```
$ ls
configuration  template.cfg
```

If you do this for the first time, then it is recommended that you make yourself familiar with the content of the *configuration* directory. This contains

- INI files for each variable which specify the plot and processing configuration
- .json files which specify interface routines for data conversion

3. adapt the configuration file (xxx.cfg) to your needs. The configuration file is the center part where you specify

   (a) which variables shall be diagnosed

   (b) which models shall be analysed

   Details about the configuration file are specified below.

4. Do it! Run the benchmarking now by executing:

```
    pycmbs.py your_config_script.cfg
```

6.2 Benchmarking components

The benchmarking part of pyCMBS is based on a modular system of diagnostics. Currently the focus is on the comparison of climate mean states between observations and models. Below are examples of the currently implemented components which the user can combine to generate a specific report for a certain variable.

6.2.1 Map season

Creates a figure with seasonal or monthly climatological means of the investigated variables.

6.2.2 Map difference

Create a plot of temporal mean fields of observations and models and their absolute and relative differences.

6.2.3 Hovmoeller plot

Generates a standard Hovmoeller plot (= time-latitude plot) of the variable.

6.2.4 Pattern correlation

The correlation between the spatial patterns of investigated variables between models and observations can be vizualized in differnt ways. For each month or season the spatial correlation coefficient is estimated and vizualized as a timeline.

6.2.5 Portraet diagram

The *Portraet diagram* was proposed by Gleckler et al. (2008). It is an efficient way to vizualize the relative rank of different models for comparisons against different observations. While the original Portraet diagram supports only two different observations, pyCMBS supports up to four different datasets for each variable.

6.2.6 Global Mean Plot

Global mean timeseries of a specific variable. The global mean values are estimated using proper area weighting. The plot object also allows to automatically plot climatology mean values.

CHAPTER 7

Benchmarking configuration

7.1 Plot configuration details

For each variable, a configuration file with the extension *.ini* is used to specify. The INI files are expected to be located in a directory which is specified in the main configuration file (.cfg). The plot configuration file specifies for each variable

- which diagnostics should be applied to a certain variable
- how plots for a particular diagnostic should look like (e.g. colorbars, limits ...)
- which observational datasets should be used

An INI file has two major parts:

1. Global plot options
2. Observation specific plot options (for each used observational dataset)

7.1.1 Global plot options

The global plot options have the following structure (example below):

```
[OPTIONS]
map_difference =  True
map_seasons    =  True
map_season_difference = False
reichler_plot  =  True
gleckler_plot   =  True
hovmoeller_plot   =  False
regional_analysis = True
global_mean    = True
vmin           =  0.
vmax           =  8.
dmin           =  -1.
dmax           =  1.
units          =  $mm/day$
label          =  Daily evaporation
cticks         = [0.,2.,4.,6.,8.,10.]
nclasses       = 8
preprocess     = True
interpolation  = conservative
targetgrid     = t63grid
```

```
projection      = robin
region_file     = /home/m300028/shared/data/CMIP5/evap/evspsbl/merged/dummy_mask2.nc
region_file_varname = regmask
```

***map_difference* [True,False]** use diagnostic to plot difference between models and observations

***map_seasons* [True,False]** use diagnostic to plot climatological monthly mean or seasonal mean maps of models and observations

***map_season_difference* [True,False]** same as map_seasons, but for difference between models and observations.

***reichler_plot* [True,False]** Summarize error skill score for this variable at the end of the section for this variable.

***gleckler_plot* [True,False]** Use this variable in the *Portraet Diagram* at the end of the report.

***hovmoeller_plot* [True,False]** Generate a hovmoeller plot for the variable, for both observations and models.

***regional_analysis* [True,False]** Perform regional analysis (statistics and correlation) of observations and models per variable.

region_file Name of netCDF file which contains the rasterized region IDs (user needs to ensure that the same geometry as the target grid is provided)

region_file_varname name of variable in *region_file*, which shall be read to identify regions; note that the data is interpreted as integer values.

global_mean generate a global mean plot for this variable (see XXXX)

vmin minimum plotting limit for data

vmax maximum plotting limit for data

dmin minimum plotting limit for difference plot

dmax maximum plotting limit for difference plot

units string to specify units of the variable. This is used for automatic labeling of plots. Note that all text can be used which can also be used for labelling in matplotlib. In particular the usage of $ is usefull to render text using latex (e.g. $frac{a}{b}$ will plot you the a/b in a nice way).

label label text to be used for the variable

cticks tick labels for colormap

7.1.2 Observation specific plot options

Below the global options, one can include an arbitrary number of observations.

Each observation is specified by a block of configuration parameters, like in the following example.:

```
[CLARASAL]
obs_file =  #get_data_pool_directory() + 'data_sources/CMSAF/CLARA-SAL/DATA/SAL_all_t63.nc'#
obs_var  =  sal
scale_data = 0.01
gleckler_position = 2
add_to_report = True
valid_mask = land
```

[Observation_Identifier] [str] unique identified for the observation. Will be used e.g. in plots as labels

obs_file [str] name of observation file. Here the user can either specify a full path name to a file or, like shown in the example above, execute a python command that is used to construct the filename. In the above example, the hash (#) is used to identify a python command. If the value of obs_file starts and ends with a hash, then the string in between is executed like you would execute a python command. Here, the routine get_data_pool_directory() is called, which returns a path name and then the remaining path to the observational data file is appended.

obs_var [str] name of variable in observation file

scale_data [float] scaling factor to be applied on data of the file. This is e.g. usefull if the netCDF file does not contain an own scale_factor attribute or if you want to apply simple conversions (e.g. from kg/m**2 s to mm/day for precipitation). The data is multiplied by the scaling factor.

gleckler_position [int] [1,2,3,4] position of the observational dataset in the Portraet diagram. Up to four different datasets can be shown at once. The meaning if the numbers is as follows: 1=top, 2=bottom, 3=left, 4=right

add_to_report [str] add this observation to the report [True,False]

valid_mask [str] [land,ocean,global]; specifies if a mask shall be applied to the dataset and model. If 'land', then all ocean areas are masked if 'ocean', then all land areas are masked. For any other options, the whole globe is used.

CHAPTER 8

Customizing pyCMBS benchmarking environment

The model benchmarking framework can be easily customized and adapted to the user needs. In the following, we will cover the following topics:

- How to add new variables for an already existing model output format?
- How to add new observational datasets?
- How to integrate new variables in the analysis?
- How to use already existing external scripts together with the pyCMBS environment?
- How to add a new model format?

8.1 How to add new variables for an already existing model output format?

A particular model output format is represented in pyCMBS by its own class. For an already existing model class, one needs to implement a reader for each variable. In principle, this is just a small subroutine that has the logic implemented how to properly read the data. Typically this requires:

- generation of filename dependent on e.g. experiment ID
- reading data into a Data object

To implement a new variable reader, there are several ways. You can either implement one routine per variable or make use of generic I/O routines (see routine get_model_data_generic in modely.py)

8.2 How to add new observational datasets?

The integration of new observational datasets is very simple as long as the datasets you use follow some standard conventions:

- datasets are in netCDF format
- optional: datasets have metadata attributes in the netCDF file. pyCMBS is making automatically use of CF conventions like netCDF attributes like *scale_factor*, *add_offset*, *_FillValue*, *units*. In case these attributes are provided, they are automatically used
- lat/lon coordinates are provided either as vectors (for simple lat/lon projections) or as 2D fields (a (lat, lon) tuple for each grid cell).

- observations are stored in a single file (all timesteps included)

Steps to integrate a new observational dataset into pyCMBS are as follows:

- decide for the variable the observational dataset belongs to –> variable name; you can look in the configuration file (.cfg) to get the currently supported variable names
- modify the corresponding INI file
- let's say, that you have chosen *sis* (surface solar irradiance) as the variable and you have a new surface radiation dataset. Then the corresponding INI file would be *sis.ini*. The INI files can be found in the *configuration* folder.
- You can however also generate an own, new configuration folder, by simply typing *pycmbs.py init* in a fresh directory
- The content of the INI file is self explanatory. You have a global section which specifies how the analysis for this particular variable shall be made (e.g. which diagnostics and plots shall be generated). Below, you have for each observational dataset a section which specifies the details for each observation. Such a section looks e.g. like the following:

```
[CERES]
obs_file = #get_data_pool_directory() + 'data_sources/CERES/EBAF/ED_26r_SFC/DATA/CERES_EBAF-Surf
obs_var  = sfc_sw_down_all_mon
scale_data = 1.
gleckler_position = 2
add_to_report = True
valid_mask = global
```

The different entries have the following meaning

[id] [str] id of observational dataset. Will be used e.g also as labels for plots

obs_file [str] path to the netCDF file with the observations here you can either specify an *absolute* path or you can out a python code snipped between two hashes '#'. The latter approach is usefull, when you have some function that directs to a particular directory, like in the example given. Otherwise, the easiest way is to just put the *absolute path* to the netCDF file.

obs_var [str] specifies the name of the variable in the netCDF file

scale_data [float] a multiplicative scaling factor that is applied to the data when it is read

gleckler_position [int] [1...4]; This number specifies, where the observational dataset will be placed in the portraet diagram (???)

add_to_report [bool] [True,False], specifies if the observational dataset should be included in the report or not. This allows to have a lot of configurations in the INI file, but use only a few of them.

valid_mask [str] [land,ocean,global], specifies which area(s) are supposed to contain valid data. Other regions are automatically masked. Thus if you specify e.g. *land*, then the ocean will be masked. It is important, that you use a

Adding a new observation is as simple as copy/paste an already existing section and modify the entries like you need it. That's it ... well at least on a technicla point. If everything is working properly and if the diagnostics you want to apply for this observational dataset are usefull is a different question.

8.2.1 Recepies for handling problems

In 80% of the cases, *pycmbs* will handle your new data smoothly. However, it might happen that your file(s) are different from the files pyCMBS was tested so far with. For these cases the following steps might help to solve your problem:

Is the file o.k?

- Have a look at the file with other tools like e.g. ncview or panoply
- make also an *ncdump -h* to check the metadata of the file

Can *cdo's* work with the file?

The preprocessing capabilities of pyCMBS largely rely on the usage of the climate data operators (cdo). If the *cdo's* can not work with your file, then pyCMBS will most likely have also problems.

- check if *cdo's* can in general read the file: *cdo sinfo <filename>*
- check if grid of the file is recognized by trying to remap the file manually using *cdo remapcon,t63grid <infile> nothing.nc*

If one of the two tests above fail, then your file is missing some essential metadata or has a strange grid or grid description that is not automatically recognized. In these cases, it would be best, if you try to figure out, why the *cdo's* are not capable to work with your dataset. Try to pose your question to the cdo's help forum (don't forget to provide details about your file; e.g. by sending the results of *ncdump -h*)

8.3 How to integrate new variables in the analysis?

To add new variables in pyCMBS implies the following steps:

1. **Define I/O routine:** Implement for each model class that shall support the new variable a routine that allows to read the data. Let's say you have a variable *sis*, then you would need e.g. to implement a routine *get_sis()* for the CMIP5RAW model class. Note that there is already a routine which can be used for generic I/O.

2. **Register I/O routine**: After you have implemented the routine to read the data, you need to let the program know about it. All data is read using a routine called *get_data()*. This routine gets the information which subroutines to call from details provided in a configuration file. The configuration file is found in:

```
./configuration/model_data_routines.json
```

The file is a simple JSON dictionary. Make yourself a bit familar with the structure and it should not be a problem to implement your new routine there.

3. **Analysis script:** Now you have the analysis script that can be used to read the data. However, you still need to tell *pycmbs* how to make use of this new information. This you do by implementing an analysis routine in *analysis.py*. For most variables supported so far, this analysis routine is just a wrapper which is calling a very generic analysis routine that basically does everything you tell it to do. What to do is specified in the *INI files* for each variable. Note however, that you are free to do what you want and you can implement a new analysis routine which is doing right the thing you want it to do.

4. **Last step** is to tell *pycmbs* that the analysis script you implemented is existing. This is again done, by simply registering it in the following file:

```
./configuration/analysis_scripts.json
```

5. [optional] **Test environment** As a best practice, you should then also integrate a unittest for this new variable. Details are specified in the section *Testing the benchmarking framework (experts only)*.

8.4 How to add a new model format?

Each model is represented by its own class which is herited from the *Model* class. Each model object/class needs to support reading of the variables that should be used for the analysis. Please look at the current code in *models.py* to see how the actual implementation can look like. Important is a coherent support of the routines which are importing data from files for a model specific file format.

CHAPTER 9

Testing the benchmarking framework (experts only)

To technically test the benchmarking framework an own framework for testing has been developed. This is based on nosetests and the code is located in the file *test_bench.py*. The framework requires test data to run. Both, model output and observations are required. Fore details see *Getting testdata for testing*

9.1 Getting testdata for testing

9.1.1 Getting model data

Tests were implemented so far for the model classes CMIP5RAWSINGLE and CMIP5RAW. Both rely on cmorized output in CMIP5 formats. To get testdata, results for a single model are sufficient. To extract the test data, the following steps are needed:

1. extract test data from MiKlip server archive using the *miklip_preprocessor*. Retrieve the required fields as follows:

```
python miklip_preprocessor.py -i /home/zmaw/m300028/cmip5_miklip -o /scratch/m300028 -e amip \\
                              -v rsds rsus pr tas sfcWind evspsbl hfls prw huss -cfg dummycfg
```

This will extract the data as described in benchmarking-cmip.

2. Choose the MPI-M folder and copy it to the testdata directory which is specified in *test_becnh.py*. by default, this is currently the directory *pycmbs_root/testdata/miklip*. Your structure should then look something like:

```
/testdata/miklip.cfg
         /MPI-M/
               /MPI-ESM-LR/
                          /amip/...
               /MPI-ESM-MR/...
```

3. get also observation data as specified in the INI configuration files (see *Getting observation data*)
4. execute the tests using *nosetests <https://nose.readthedocs.org/en/latest/>_*:

```
# change into test directory
cd <pycmbsdir>/benchmarking/tests/test_benchmarking
nosetests
```

This will execute tests, for both, CMIP5RAWSINGLE and CMIP5RAW classes. The variables tested are specified in *test_bench.py*

Note: The tests are only of technical character. Thus they only indicate that data I/O and benchmarking environment is working in principle well. A proper testing is nevertheless required to ensure that the results are valid (e.g. data scaling, valid areas ...)

9.1.2 Getting observation data

Observational data can come from various sources which depend on the input datasets specified in the INI configuration files. In general, the user can store the observational data in two different ways:

1. store data at custom places: provide a custom filename in the INI file
2. on a central data pool: use routine *get_data_pool_directory()* to specify the file location

The latter has the advantage, that when working on different file systems, the filenames don't need to be adapted, but only the name of the root directory where the data is stored on different servers. For examples see the code in the INI files provided.

CHAPTER 10

References

10.1 Project website

- pyCMBS on github (public development project)

10.2 Publications where pyCMBS has been used

- Brovkin, V. et al., 2013. Evaluation of vegetation cover and land-surface albedo in MPI-ESM CMIP5 simulations. Journal of Advances in Modeling Earth Systems, 5(1), pp.48-57. doi:10.1029/2012MS000169
- Hagemann, S., Loew, A. & Andersson, A., 2013. Combined evaluation of MPI-ESM land surface water and energy fluxes. Journal of Advances in Modeling Earth Systems, 5(2), pp.259-286. doi:10.1029/2012MS000173
- Loew, A. et al., 2013. Potential and limitations of multidecadal satellite soil moisture observations for selected climate model evaluation studies. Hydrology and Earth System Sciences, 17(9), pp.3523-3542. doi:10.5194/hess-17-3523-2013
- Loew, A., 2013. Terrestrial satellite records for climate studies: how long is long enough? A test case for the Sahel. Theoretical and Applied Climatology. doi:10.1007/s00704-013-0880-6

10.3 Projects where pyCMBS has been used

- ESA GlobAlbedo
- EvaCliMod (German Weather Service)
- ESA Climate Change Initiative (European Space Agency)

CHAPTER 11

Indices and tables

- genindex
- modindex
- search