SCOPE: A Script Based Coupler for Simulations of the Earth System Documentation

Release 0.2.0

Paul Gierz

Mar 13, 2023

Contents:

1 Installation **3**
- 1.1 Stable release . . . 3
- 1.2 From sources . . . 3

2 Using SCOPE **5**
- 2.1 Example: Configuration Files for `SCOPE` . . . 6
- 2.2 Example: `PISM` to `ECHAM6` . . . 7
- 2.3 Command line interface . . . 7
- 2.4 Python Library Usage . . . 8

3 scope **9**
- 3.1 scope package . . . 9

4 Contributing **17**
- 4.1 Types of Contributions . . . 17
- 4.2 Get Started! . . . 18
- 4.3 Pull Request Guidelines . . . 19
- 4.4 Tips . . . 19
- 4.5 Deploying . . . 19

5 Credits **21**

6 History **23**
- 6.1 0.1.4 (2019-12-12) . . . 23
- 6.2 0.1.3 (2019-12-4) . . . 23
- 6.3 0.1.0 (2019-11-13) . . . 23

7 Indices and tables **25**

Python Module Index **27**

Index **29**

Welcome to the `scope` documentation. `scope` is a two-in-one command line utility as well as a Python library designed to facilitate coupling and communication between various Earth system models. The minimal quickstart:

```
$ pip install scope-coupler
$ scope --help
$ scope preprocess ${CONFIG_FILE} echam
$ scope regrid ${CONFIG_FILE} pism
```

The commands listed above would install the scope coupler; show you what it can do, gather relevant files from the atmosphere model `echam`, and regrid them onto a `pism` ice sheet grid.

However, `scope` is capable of much more than this. You can preprocess or postprocess data on either side of the communication, modify variable names and attributes, perform corrections due to resolution differences, and provide your own specific steps for each part of the coupling process.

`scope` is designed to run completely independently of the models being used, the run-time infrastructure available on the supercomputer, and, perhaps most importantly **requires 0 modifications to your model code**. To get started, have a look at the documentation below.

CHAPTER 1

Installation

1.1 Stable release

To install scope, run this command in your terminal:

```
$ pip install scope-coupler
```

This is the preferred method to install scope, as it will always install the most recent stable release.

> **Warning:** Since `scope` is still under active development, there is no "stable release" yet.

If you don't have pip installed, this Python installation guide can guide you through the process.

1.2 From sources

The sources for scope can be downloaded from AWI's Gitlab repo.

You can either clone the public repository:

```
$ git clone git://gitlab.awi.de/pgierz/scope
```

Or download the tarball:

```
$ curl  -OL https://gitlab.awi.de/pgierz/scope/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 2

Using SCOPE

`scope` uses configuration files, generally written in the `YAML` syntax, to define what you wantt it do. This configuration is divided into seperate sections. Here, we give a brief overview. A complete reference is currently being written.

The first section, simply titled `scope`, defines general parameters for the program:

```
1 scope:
2     couple_dir: "/path/to/directory/"
3     number openMP processes: 8
```

In the example above, we define two things for the program.

`couple_dir` Line 2

This entry defines where scope will save it's files. These files generally include remap weights to be reused each time the coupler is called; gathered output files for processing for the other model; and intermediate files that may be interesting for diagnosis.

`number openMP processes` Line 3

Since `scope` uses `cdo` in the background; you can use this to define how many processes you want to run `cdo` on. This generally speeds up regridding.

Next, there is a section which may optionally be defined, `template_replacements`. Here, you can store key/value pairs which will be replaced elsewhere in the configuration. As an example:

```
1 template_replacements:
2     EXP_ID: "PI_1x10"
3     DATE_PATTERN: "[0-9]{6}"
```

Now, any other time that `{{ EXP_ID }}` is used in the configuration, it will be replaced with `PI_1x10`. The syntax here is derived from the Jinja2 Python package used for templating.

Warning: I'm not sure what happens here if you try to use recursion in the template replacements!

You can also see in this section that you can define a `DATE_PATTERN`. Specific key/value pairs ending with the substring `PATTERN` are treated as a regular expression.

Next, you describe the models you wish to couple together.

```
model_name:
    type: physical_domain
    griddes: built-in CDO grid description, or path to a SCRIP formatted file
    outdata_dir: /some/path
    code table: build-in CDO code table, or path to a file with GRB-style code table
    send:
        ...send directives...
    receive:
        ...receive directives...
    pre_step:
        ...description of pre step...
    post_step:
        ...description of post step...
```

In the generalised example above, we define:

model_name A model to couple, in this case, `model_name`. Usually, this would be more specific, e.g. `echam`, `openifs`, `pism`, `fesom`.

Inside the `model_name` configuration, we again have:

type This describes the *type* of the model; e.g. atmosphere, ice, ocean.

griddes Here, you must specify which grid description to use for this model. This is the default for this model.

outdata_dir This defines where scope will, by default, look for files for this particular model. However, you can override this on a case by case basis. See the send directives for more information.

code_table Since `scope` is built on top of `cdo`, which supports `grb` files, here you can specify which code table to use in order to detect variable names when converting from `grb` to `netcdf`.

send This configuration contains send directives for other coupling partners. More on this in the next section.

receive This configuration is used to receive information from other models.

pre_step Programs run before a particular step. Can be configured for each step separately, e.g. `pre_preprocess`, or `pre_regrid`.

post_step Programs run after a particular step

2.1 Example: Configuration Files for SCOPE

A complete example configuration file is provided under `examples/scope_config.yaml`:

```
template_replacements:
    EXP_ID: "PI_1x10"
    DATE_PATTERN: "[0-9]{6}"

scope:
    couple_dir: "/work/ollie/pgierz/scope_tests/couple/"
    number openMP processes: 8

```

(continues on next page)

(continued from previous page)

```
echam:
    type: atmosphere
    griddes: T63
    outdata_dir: "/work/ollie/pgierz/scope_tests/outdata/echam/"
    code table: "echam6"
    pre_preprocess:
        program: "echo \"hello from pre_preprocess. Do you know: $(( 7 * 6 )) is the answer!\""
    send:
        ice:
            temp2:
                files:
                    pattern: "{{ EXP_ID }}_echam6_echam_{{ DATE_PATTERN }}.grb"
                    take:
                        newest: 12
                code table: "echam6"
            aprl:
                files:
                    dir: "/work/ollie/pgierz/scope_tests/outdata/echam/"
                    pattern: "{{ EXP_ID }}_echam6_echam_{{ DATE_PATTERN }}.grb"
                    take:
                        newest: 12
                code table: "/work/ollie/pgierz/scope_tests/outdata/echam/PI_1x10_185001.01_echam.codes"
            aprc:
                files:
                    dir: "/work/ollie/pgierz/scope_tests/outdata/echam/"
                    pattern: "{{ EXP_ID }}_echam6_echam_{{ DATE_PATTERN }}.grb"
                    take:
                        newest: 12

pism:
    type: ice
    griddes: ice.griddes
    recieve:
        atmosphere:
            temp2:
                interp: bil
                transformation:
                    - expr: "air_temp=temp2-273.15"
        ocean:
    send:
        atmosphere:
        ocean:
```

2.2 Example: PISM to ECHAM6

2.3 Command line interface

`scope` comes with a command line interface. For a very quick introduction:

```
$ scope --help
```

This will print usage information.

Any `scope` commands you normally would run in a batch job can also be individually targeted via command line arguments. In principle, the command structure is always the same, namely:

```
$ scope <command> ${CONFIG} ${WHOS_TURN}
```

This allows you to run one specific part of `scope` for a particular configuration assuming a particular model is currently doing something. As an example, this could take the form of:

```
$ scope preprocess ~/Code/scope/examples/scope_config.yaml echam
```

This would cause `scope` to run the prepare steps described for `echam`; in this particular case gathering output files, extracting variables, and placing the resulting file into the couple folder described in the configuration file. Note that also and pre- and post-processing hooks defined in the configuration file will also be run at this point.

All available commands are printed via `scope --help`.

Currently, the following commands are implemented:

- `preprocess`
- `regrid`

2.4 Python Library Usage

While the command line interface is nice for users who never want to actually touch `scope` code; we also support the ability to use scope functions in your own `Python` programs. this section describes how to use `scope` from a script.

To use scope in a project:

```
import scope
```

Consider having a look at the developer API for more detailed usage.

CHAPTER 3

scope

3.1 scope package

3.1.1 Submodules

3.1.2 scope.cli module

Console script for scope.

`scope.cli.`**`yaml_file_to_dict`**(*filepath: str*) → dict

Given a scope configuration yaml file, returns a corresponding dictionary.

If you do not give an extension, tries again after appending one:

- `.yml`
- `.yaml`
- `.YML`
- `.YAML`

Note that this function also uses *~jinja2* to replace any templated variables found in the under the top-level key `template_replacements`. This key is then deleted from the remainder of the dictionary.

Parameters **`filepath`** (*`str`*) – Where to get the YAML file from

Returns A dictionary representation of the yaml file.

Return type dict

Raises `OSError` if the file cannot be found.

3.1.3 scope.models module

Not sure what to do with this stuff yet. . .

class scope.models.**Component**
Bases: *scope.models.SimObj*

NAME = 'Generic Component Object'

class scope.models.**Model**
Bases: *scope.models.SimObj*

NAME = 'Generic Model Object'

class scope.models.**SimObj**
Bases: object

after_run()

before_recieve()

before_send()

recieve()

send()

NAME = 'Generic Sim Object'

3.1.4 scope.scope module

Here, the scope library is described. This allows you to use specific parts of scope from other programs.

scope consists of several main classes. Note that most of them provide Python access to cdo calls via Python's built-in subprocess module. Without a correctly installed cdo, many of these functions/classes will not work.

We provide a quick summary, but please look at the documentation for each function and class for more complete information. The following functions are defined:

- determine_cdo_openMP – using cdo --version, determines if you have openMP support.

The following classes are defined here:

- Scope – an abstract base class useful for starting other classes from. This provides a way to determine if cdo has openMP support or not by parsing cdo --version. Additionally, it has a nested class which gives you decorators to put around methods for enabling arbitrary shell calls before and after the method is executed, which can be configured via the Scope.config dictionary.
- Send – a class to extract and combine various NetCDF files for further processing.
- Recieve – a class to easily regrid from one model to another, depending on the specifications in the scope_config.yaml
- – * .

  .

  .

class scope.scope.**Recieve**(*config: dict*, *whos_turn: str*)
Bases: *scope.scope.Scope*

Parameters

- **config** (*dict*) – A dictionary (normally recieved from a YAML file) describing the scope configuration. An example dictionary is included in the root directory under examples/scope_config.yaml
- **whos_turn** (*str*) – An explicit model name telling you which model is currently interfacing with scope e.g. echam or pism.

Warning: This function has a filesystem side-effect: it generates the couple folder defined in `config["scope"]["couple_dir"]`. If you don't have permissions to create this folder, the object initialization will fail...

Some design features are listed below:

- **"pre" and "post" hooks**

Any appropriately decorated method of a `scope` object has a hook to call a script with specific arguments and flags before and after the main scope method call. Best explained by an example. Assume your Scope subclass has a method "send". Here is the order the program will execute in, given the following configuration:

```
pre_send:
    program: /some/path/to/an/executable
    args:
        - list
        - of
        - arguments
    flags:
        - "--flag value1"
        - "--different_flag value2"

post_send:
    program: /some/other/path
    args:
        - A
        - B
        - C
    flags:
        - "--different_flag value3"
```

Given this configuration, an idealized system call would look like the example shown below. Note however that the Python program calls the shell and immediately destroys it again, so any variables exported to the environment (probably) don't survive:

```
$ ./pre_send['program'] list of arguments --flag value1 --different_flag value2
$ <... python call to send method ...>
$ ./post_send['program'] A B C --different_flag value 3
```

_calculate_weights(*model*, *type_*, *interp*)

_combine_tmp_variable_files(*target_file*, *source_files*)

recieve()

regrid_one_var(*model*, *type_*, *interp*, *variable*, *target_file*)

regrid_recieve_from(*model*, *type_*, *interp*, *variable*, *target_file*, *recv_from*)

run_cdos(*model*, *type_*, *variable*, *target_file*, *cdo_commands*)

class scope.scope.**Scope**(*config: dict*, *whos_turn: str*)

Bases: object

Base class for various Scope objects. Other classes should extend this one.

Parameters

- **config** (*dict*) – A dictionary (normally recieved from a YAML file) describing the `scope` configuration. An example dictionary is included in the root directory under `examples/scope_config.yaml`

- **whos_turn** (*str*) – An explicit model name telling you which model is currently interfacing with `scope` e.g. `echam` or `pism`.

Warning: This function has a filesystem side-effect: it generates the couple folder defined in `config["scope"]["couple_dir"]`. If you don't have permissions to create this folder, the object initialization will fail...

Some design features are listed below:

- **"pre" and "post" hooks**

Any appropriately decorated method of a `scope` object has a hook to call a script with specific arguments and flags before and after the main scope method call. Best explained by an example. Assume your Scope subclass has a method "send". Here is the order the program will execute in, given the following configuration:

```
pre_send:
    program: /some/path/to/an/executable
    args:
        - list
        - of
        - arguments
    flags:
        - "--flag value1"
        - "--different_flag value2"

post_send:
    program: /some/other/path
    args:
        - A
        - B
        - C
    flags:
        - "--different_flag value3"
```

Given this configuration, an idealized system call would look like the example shown below. Note however that the Python program calls the shell and immediately destroys it again, so any variables exported to the environment (probably) don't survive:

```
$ ./pre_send['program'] list of arguments --flag value1 --different_flag value2
$ <... python call to send method ...>
$ ./post_send['program'] A B C --different_flag value 3
```

class ScopeDecorators

Bases: `object`

Contains decorators you can use on class methods

static _wrap_hook(*self*, *meth*, *pre_or_post*)

classmethod post_hook(*meth*)

classmethod pre_hook(*meth*)

Based upon the `self.config`, runs a specific system command

Using the method name, you can define

get_cdo_prefix(*has_openMP: bool = False*)

Return a string with an appropriate `cdo` prefix for using OpenMP with the `-P` flag.

Parameters **has_openMP** (*bool*) – Default is `False`. You can explicitly override the ability of `cdo` to use the `-P` flag. If set to `True`, the config must have an entry under

config[scope][number openMP processes] defining how many openMP processes to use (should be an int)

Returns A string which should be used for the cdo call, either with or without -P X, where X is the number of openMP processes to use.

Return type str

class scope.scope.**Send**(*config: dict*, *whos_turn: str*)

Bases: *scope.scope.Scope*

Subclass of Scope which enables sending of models via cdo. Use the send method after building a Precprocess object.

Parameters

- **config** (*dict*) – A dictionary (normally recieved from a YAML file) describing the scope configuration. An example dictionary is included in the root directory under examples/scope_config.yaml
- **whos_turn** (*str*) – An explicit model name telling you which model is currently interfacing with scope e.g. echam or pism.

Warning: This function has a filesystem side-effect: it generates the couple folder defined in config["scope"]["couple_dir"]. If you don't have permissions to create this folder, the object initialization will fail...

Some design features are listed below:

- **"pre" and "post" hooks**

Any appropriately decorated method of a scope object has a hook to call a script with specific arguments and flags before and after the main scope method call. Best explained by an example. Assume your Scope subclass has a method "send". Here is the order the program will execute in, given the following configuration:

```
pre_send:
    program: /some/path/to/an/executable
    args:
        - list
        - of
        - arguments
    flags:
        - "--flag value1"
        - "--different_flag value2"

post_send:
    program: /some/other/path
    args:
        - A
        - B
        - C
    flags:
        - "--different_flag value3"
```

Given this configuration, an idealized system call would look like the example shown below. Note however that the Python program calls the shell and immediately destroys it again, so any variables exported to the environment (probably) don't survive:

```
$ ./pre_send['program'] list of arguments --flag value1 --different_flag value2
$ <... python call to send method ...>
$ ./post_send['program'] A B C --different_flag value 3
```

_all_senders()
A generator giving tuples of the *reciever_type* (e.g. ice, atmosphere, ocean, solid earth), and the *configuration for the reciever type*, including variables and corresponding specifications for which files to use and how to process them.

Example

Here is an example for the reciever specification dictionary. See the documentation regarding `scope` configuration for further information:

```
temp2:
    files:
        pattern: "{{ EXP_ID }}_echam6_echam_{{ DATE_PATTERN }}.grb"
        take:
            newest: 12
    code_table: "echam6"
aprl:
    files:
        dir: "/work/ollie/pgierz/scope_tests/outdata/echam/"
        pattern: "{{ EXP_ID }}_echam6_echam_{{ DATE_PATTERN }}.grb"
        take:
            newest: 12
    code_table: "/work/ollie/pgierz/scope_tests/outdata/echam/PI_1x10_185001.
↪01_echam.codes"
```

Yields *tuple of (str, dict)* – The first element of the tuple, `reciever_type`, is a string describing what sort of model should get this data; e.g. "ice", "atmosphere"

The second element, `reciever_spec`, is a dictionary describing which files should be used.

_combine_tmp_variable_files(*reciever_type*, *files_to_combine*)
Combines all files in the couple directory for a particular reciever type.

Depending on the configuration, this method combines all files found in the `couple_dir` which may have been further processed by `scope` to a file `<sender_type>_file_for_<reciever_type>.nc`

Parameters **reciever_type** (*str*) – Which reciever the model is sending to, e.g. ice, ocean, atmosphere

Returns

Return type None

Notes

This executes a `cdo merge` command to concatenate all files found which should be sent to particular model.

_construct_filelist(*var_dict*)
Constructs a file list to use for further processing based on user specifications.

Parameters **var_dict** (*dict*) – Configuration dictionary for how to handle one specific variable.

Returns A list of files for further processing.

Return type file_list

Example

The variable configuration dictionary can have the following top-level **keys**:

- **`files` may contain:**
  - a `filepattern` in regex to look for
  - `take` which files or timesteps to take, either specific, or `newest`/`latest` followed by an integer.
  - `dir` a directory where to look for the files. Note that if this is not provided, the default is to fall back to the top level `outdata_dir` for the currently sending model.

`_make_tmp_files_for_variable`(*varname*, *var_dict*)

Generates temporary files for further processing with `scope`.

Given a variable name and a description dictionary of how it should be extracted and processed, this method makes a temporary file, `<sender_name>_<varname>_file_for_scope.dat`, e.g. `echam_temp2_file_for_scope.dat` in the `couple_dir`.

Parameters

- **`varname`** (*str*) – Variable name as that should be selected from the files
- **`var_dict`** (*dict*) – A configuration dictionary describing how the variable should be extracted. An example is given in `_construct_filelist`.

Notes

In addition to the dictionary description of `files`, further information may be added with the following top-level keys:

- `code_table` describing which `GRIB` code numbers correspond to which variables. If not given, the fallback value is the value of `code_table` in the sender configuration.

Converts any input file to `nc` via *cdo*. Runs both `select` and `settable`.

Returns

Return type None

`_rename_send_as_variables`(*variable_name*, *variable_dict*)

`_run_cdo_for_variable`(*variable_name*, *variable_dict*)

`send`()

Selects and combines variables from various file into one single file for futher processing.

- `<sender_type>_file_for_<reciever_type>` (e.g. `atmosphere_file_for_ice.nc`)

Parameters **`None`** –

Returns

Return type None

`scope.scope.`**`determine_cdo_openMP`**() → bool

Checks if the `cdo` version being used supports `OpenMP`; useful to check if you need a `-P` flag or not.

Parameters **None** –

Returns True if OpenMP is listed in the Features of cdo, otherwise False

Return type bool

scope.scope.**determine_fileextension**(*f: str*) → str

scope.scope.**get_newest_n_timesteps**(*f: str*, *take: int*) → str

Given a file, takes the newest n timesteps for further processing.

Parameters

- **f** (*str*) – The file to use.
- **take** (*int*) – Number of timesteps to take (newest will be taken, i.e. from the end of the file). Please use a positive value!

Returns A string with the path to the new file

Return type str

scope.scope.**get_oldest_n_timesteps**(*f: str*, *take: int*) → str

Given a file, takes the oldest n timesteps for further processing.

Parameters

- **f** (*str*) – The file to use.
- **take** (*int*) – Number of timesteps to take (oldest will be taken, i.e. from the beginning of the file).

Returns A string with the path to the new file

Return type str

scope.scope.**rename_with_suggested_fileext**(*f: str*) → None

Renames a file with the suggested file extension

scope.scope.**suggest_fileext**(*f: str*) → str

Given a file, uses CDO to determine which file extension is should have, and gives back an appropriate string that can be used for renaming.

3.1.5 Module contents

Top-level package for SCOPE.

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/pgierz/scope/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

4.1.4 Write Documentation

SCOPE: A Script Based Coupler for Simulations of the Earth System could always use more documentation, whether as part of the official SCOPE: A Script Based Coupler for Simulations of the Earth System docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/pgierz/scope/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up *scope* for local development.

1. Fork the *scope* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/scope.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv scope
$ cd scope/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 scope tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/pgierz/scope/pull_requests and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_scope
```

4.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 5

Credits

- Paul Gierz <pgierz@awi.de>

CHAPTER 6

History

6.1 0.1.4 (2019-12-12)

- Includes examples directory and a test `scope_config.yaml`
- Fixes a small logging error in the command line interface
- Updates documentation of main module
- Adds pre and post hooks functionality
- Changes maximum line length in flake8 to 160 characters, black code style for 120 characters.

6.2 0.1.3 (2019-12-4)

- Automatically builds documentation from docstrings
- Most of `scope send` works

6.3 0.1.0 (2019-11-13)

- First release on PyPI.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

scope, 16
scope.cli, 9
scope.models, 9
scope.scope, 10

Index

Symbols

_all_senders() (*scope.scope.Send method*), 13
_calculate_weights() (*scope.scope.Recieve method*), 11
_combine_tmp_variable_files() (*scope.scope.Recieve method*), 11
_combine_tmp_variable_files() (*scope.scope.Send method*), 14
_construct_filelist() (*scope.scope.Send method*), 14
_make_tmp_files_for_variable() (*scope.scope.Send method*), 15
_rename_send_as_variables() (*scope.scope.Send method*), 15
_run_cdo_for_variable() (*scope.scope.Send method*), 15
_wrap_hook() (*scope.scope.Scope.ScopeDecorators static method*), 12

A

after_run() (*scope.models.SimObj method*), 10

B

before_recieve() (*scope.models.SimObj method*), 10
before_send() (*scope.models.SimObj method*), 10

C

Component (*class in scope.models*), 9

D

determine_cdo_openMP() (*in module scope.scope*), 15
determine_fileextension() (*in module scope.scope*), 16

G

get_cdo_prefix() (*scope.scope.Scope method*), 12
get_newest_n_timesteps() (*in module scope.scope*), 16
get_oldest_n_timesteps() (*in module scope.scope*), 16

M

Model (*class in scope.models*), 10

N

NAME (*scope.models.Component attribute*), 10
NAME (*scope.models.Model attribute*), 10
NAME (*scope.models.SimObj attribute*), 10

P

post_hook() (*scope.scope.Scope.ScopeDecorators class method*), 12
pre_hook() (*scope.scope.Scope.ScopeDecorators class method*), 12

R

Recieve (*class in scope.scope*), 10
recieve() (*scope.models.SimObj method*), 10
recieve() (*scope.scope.Recieve method*), 11
regrid_one_var() (*scope.scope.Recieve method*), 11
regrid_recieve_from() (*scope.scope.Recieve method*), 11
rename_with_suggested_fileext() (*in module scope.scope*), 16
run_cdos() (*scope.scope.Recieve method*), 11

S

Scope (*class in scope.scope*), 11
scope (*module*), 16
scope.cli (*module*), 9
scope.models (*module*), 9
scope.scope (*module*), 10
Scope.ScopeDecorators (*class in scope.scope*), 12
Send (*class in scope.scope*), 13

send() (*scope.models.SimObj method*), 10
send() (*scope.scope.Send method*), 15
SimObj (*class in scope.models*), 10
suggest_fileext() (*in module scope.scope*), 16

Y

yaml_file_to_dict() (*in module scope.cli*), 9