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

Release __version__ = '0.1.4'

Paul Gierz

Dec 16, 2019

Contents:

CHAPTER 1

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

`scope` is a offline, script-based coupler for various simulations of the Earth System. It is written in Python and configured with YAML files.

- Free software: GNU General Public License v3
- Documentation: https://scope-coupler.readthedocs.io.

1.1 Features

- TODO

1.2 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER 2

Installation

2.1 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.

2.2 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 3

Usage

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

```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

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"
```

(continues on next page)

(continued from previous page)

```
            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:
```

To use scope in a project:

```
import scope
```

CHAPTER 4

scope

4.1 scope package

4.1.1 Submodules

4.1.2 scope.cli module

Console script for scope.

scope.cli.**yaml_file_to_dict**(*filepath*)
Given a yaml file, returns a corresponding dictionary.

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

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

Returns A dictionary representation of the yaml file.

Return type dict

4.1.3 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'

4.1.4 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.

Here, 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.
- `Preprocess` – a class to extract and combine various NetCDF files for further processing.
- `Regrid` – a class to easily regrid from one model to another, depending on the specifications in the `scope_config.yaml`

class scope.scope.**Preprocess**(*config*, *whos_turn*)

Bases: *scope.scope.Scope*

Subclass of `Scope` which enables preprocessing of models via `cdo`. Use the `preprocess` method after building a `Precprocess` 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 "preprocess". Here is the order the program will execute in, given the following configuration:

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

post_preprocess:
    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_preprocess['program'] list of arguments --flag value1 --different_flag␣
↪value2
$ <... python call to preprocess method ...>
$ ./post_preprocess['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
```

(continues on next page)

(continued from previous page)

```
    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*)

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 mergetime` command to concatenate all files found which should be sent to particular model.

_construct_filelist(*var_dict*)

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 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.nc`, e.g. `echam_temp2_file_for_scope.nc` 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

preprocess()

class scope.scope.**Regrid**(*config*, *whos_turn*)

Bases: *scope.scope.Scope*

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 "preprocess". Here is the order the program will execute in, given the following configuration:

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

post_preprocess:
    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_preprocess['program'] list of arguments --flag value1 --different_flag␣
↪value2
$ <... python call to preprocess method ...>
$ ./post_preprocess['program'] A B C --different_flag value 3
```

_calculate_weights(*Model*, *Type*, *Interp*)

regrid()

regrid_one_var(*Model*, *Type*, *Interp*, *Variable*)

class scope.scope.**Scope**(*config*, *whos_turn*)

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 "preprocess". Here is the order the program will execute in, given the following configuration:

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

post_preprocess:
    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_preprocess['program'] list of arguments --flag value1 --different_flag␣
↪value2
$ <... python call to preprocess method ...>
$ ./post_preprocess['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*)

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=None*)

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

Parameters **has_openMP** (*bool*) – Default is None. 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

scope.scope.**determine_cdo_openMP**()

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

4.1.5 Module contents

Top-level package for SCOPE.

CHAPTER 5

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:

5.1 Types of Contributions

5.1.1 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.

5.1.2 Fix Bugs

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

5.1.3 Implement Features

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

5.1.4 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.

5.1.5 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 :)

5.2 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.

5.3 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.

5.4 Tips

To run a subset of tests:

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

5.5 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 6

Credits

- Paul Gierz <pgierz@awi.de>

CHAPTER 7

History

7.1 0.1.3 (2019-12-4)

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

7.2 0.1.0 (2019-11-13)

- First release on PyPI.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

Symbols

A

B

C

D

G

M

N

P

R

S

Y