Caelus Python Library (CPL)

Release v0.1.1

Applied CCM

May 12, 2018

Contents

I User Manual **3**

1 Introduction **5**
- 1.1 Usage . . . 5
- 1.2 Contributing . . . 5

2 Installing Caelus Python Library (CPL) **7**
- 2.1 Installing CPL with Anaconda Python Distribution . . . 7
  - 2.1.1 Install Anaconda . . . 7
  - 2.1.2 Install CPL . . . 8
- 2.2 Alternate Installation – Virtualenv . . . 8
  - 2.2.1 Prepare system for virtual environment . . . 9
    - 2.2.1.1 Useful virtualenvwrapper commands . . . 9
  - 2.2.2 Install CPL . . . 9
- 2.3 Check installation . . . 10
- 2.4 Building documentation . . . 10
- 2.5 Running tests . . . 10

3 Configuring Caelus Python Library **11**
- 3.1 Checking current configuration . . . 12
- 3.2 CPL configuration reference . . . 13
  - 3.2.1 Core library configuration . . . 13
    - 3.2.1.1 Python environment options . . . 13
    - 3.2.1.2 System configuration . . . 14
    - 3.2.1.3 CPL logging options . . . 14
  - 3.2.2 CML version configuration . . . 15

4 Command-line Applications **17**
- 4.1 Common CLI options . . . 17
- 4.2 Available command-line applications . . . 17
  - 4.2.1 `caelus` – Common CPL actions . . . 17
    - 4.2.1.1 caelus cfg – Print CPL configuration . . . 18
    - 4.2.1.2 caelus env – write shell environment file . . . 18
    - 4.2.1.3 caelus clone – Clone a case directory . . . 19
    - 4.2.1.4 caelus tasks – run tasks from a file . . . 20
    - 4.2.1.5 caelus run – run a Caelus executable in the appropriate environment . . . 20
    - 4.2.1.6 caelus logs – process a Caelus solver log file from a run . . . 21
    - 4.2.1.7 caelus clean – clean a Caelus case directory . . . 22

4.2.2 caelus_tutorials – Run tutorials . . . 22

5 Caelus Tasks **25**
5.1 Quick tutorial . . . 25
5.2 Tasks reference . . . 26
5.2.1 run_command – Run CML executables . . . 26
5.2.2 copy_files – Copy files . . . 27
5.2.3 copy_tree – Recursively copy directories . . . 27
5.2.4 clean_case – Clean a case directory . . . 27
5.2.5 process_logs – Process solver outputs . . . 28
5.2.6 task_set – Group tasks . . . 28

II Developer Manual **29**

6 Caelus Python API **31**
6.1 caelus.config – Caelus Configuration Infrastructure . . . 31
6.1.1 Caelus Python Configuration . . . 31
6.1.2 Caelus CML Environment Manager . . . 33
6.2 caelus.utils – Basic utilities . . . 34
6.2.1 Struct Module . . . 35
6.2.2 Miscellaneous utilities . . . 36
6.3 caelus.run – CML Execution Utilities . . . 38
6.3.1 Caelus Tasks Manager . . . 38
6.3.2 CML Execution Utilities . . . 39
6.3.3 Job Scheduler Interface . . . 41
6.4 caelus.post – Post-processing utilities . . . 44
6.4.1 Caelus Log Analyzer . . . 45
6.4.2 Caelus Plotting Utilities . . . 47
6.5 caelus.scripts – CLI App Utilities . . . 49
6.5.1 Basic CLI Interface . . . 49

III Indices and tables **51**

Python Module Index **55**

Caelus Python Library is a companion package for interacting with Caelus CML open-source CFD package. The library provides utilities for pre and post-processing, as well as automating various aspects of the CFD simulation workflow. Written in Python, it provides a consistent user-interface across the three major operating systems Linux, Windows, and Mac OS X ensuring that the scripts written in one platform can be quickly copied and used on other platforms.

Like CML, CPL is also an open-source library released under the Apache License Version 2.0 license. See Apache License Version 2.0 for more details on use and distribution.

This documentation is split into two parts: a *user* and a *developer* manual. New users should start with the user manual that provides an overview of the features and capabilities currently available in CPL, the installation process and examples of usage. The developer manual documents the application programming interface (API) and is useful for users and developers looking to write their own python scripts to extend functionality or add features to the library. See *Introduction* for more details.

Part I

User Manual

CHAPTER 1

Introduction

The primary motivation for CPL is to provide a platform-agnostic capability to automate the CFD simulation workflow with Caelus CML package. The package is configurable to adapt to different user needs and system configurations and can interact with multiple CML versions simultaneous without the need to source *environment* files (e.g., using `caelus-bashrc` on Unix systems).

Some highlights of CPL include:

- The library is built using Python programming language and uses scientific python libraries (e.g., NumPy, Matplotlib). Capable of running on both Python 2.7 as well as Python 3.x versions.
- Uses YAML format for configuration files and input files. The YAML files can be read, manipulated, and written out to disk using libraries available in several programming languages, not just Python.
- Provides modules and python classes to work with Caelus case directories, process and plot logs, etc. The API is documented to allow users to build custom workflows that are currently not part of CPL.
- A YAML-based *task* workflow capable of automating the mesh, pre-process, solve, post-process workflow on both local workstations as well as high-performance computing (HPC) systems with job schedulers.

1.1 Usage

CPL is distributed under the terms Apache License Version 2.0 open-source license. Users can download the installers from Applied CCM's website, or access the Git repository hosted on BitBucket. Please follow *Installing Caelus Python Library (CPL)* for more details on how to install CPL and its dependencies within an existing Python installation on your system.

Please contact the developers with questions, issues, or bug reports.

1.2 Contributing

CPL is an open-source project and welcomes the contributions from the user community. Users wishing to contribute should submit pull requests to the public git repository.

CHAPTER 2

Installing Caelus Python Library (CPL)

CPL is a python package for use with Caelus CML simulation suite. Therefore, it is assumed that users have a properly functioning CML installation on their system. In addition to Caelus CML and python, it also requires several scientific python libraries:

- NumPy – Arrays, linear algebra
- Pandas – Data Analysis library
- Matplotlib – Plotting package

The quickest way to install CPL is to use the official installer provided by Applied CCM. Once installed, please proceed to *Check installation* to learn how to use CPL.

For users wishing to install CPL from the git repository, this user guide recommends the use of Anaconda Python Distribution. This distribution provides a comprehensive set of python packages necessary to get up and running with CPL. An alternate approach using Python *virtualenv* is described at the end of this section, but will require some Python expertise on the part of the user.

The default installation instructions use Python v2.7. However, CPL is designed to work with both Python v2.7 and Python v3.x versions.

2.1 Installing CPL with Anaconda Python Distribution

2.1.1 Install Anaconda

1. Download the Anaconda installer for your operating system.
2. Execute the downloaded file and follow the installation instructions. It is recommended that you install the default packages.
3. Update the anaconda environment according to installation instructions

Note: Make sure that you answer `yes` when the installer asks to add the installation location to your default PATH locations. Or else the following commands will not work. It might be necessary to open a new shell for the environment to be updated.

2.1.2 Install CPL

1. Obtain the CPL source from the public Git repository.

```
# Change to directory where you want to develop/store sources
git clone https://bitbucket.org/appliedccm/CPL
cd CPL
```

2. Create a custom conda environment

```
# Ensure working directory is CPL
conda env create -f etc/caelus2.yml
```

Note:

(a) Developers interested in developing CPL might want to install the development environment available in `etc/caelus2-dev.yml`. This installs additional packages like `sphinx` for document generation, and `pytest` for running the test suite.

(b) By default, the environment created is named `caelus2` when using `etc/caelus2.yml` and `caelus-dev` when using `etc/caelus2-dev.yml`. The user can change the name of the environment by using *-n <env_name>* option in the previous command.

(c) Users wishing to use Python 3.x should replace `etc/caelus2.yml` with `etc/caelus3.yml`. Both `caelus2` and `caelus3` environment can be used side by side for testing and development.

3. Activate the custom environment and install CPL within this environment

```
source activate caelus2
pip install .
```

For *editable* development versions of CPL use `pip install -e .` instead.

After completing this steps, please proceed to *Check installation* to test that your installation is working properly.

2.2 Alternate Installation – Virtualenv

This method is suitable for users who prefer to use the existing python installations in their system (e.g., from `apt-get` for Linux systems). A brief outline of the installation process is described here. Users are referred to the following documentation for more assistance:

1. Virtualenv
2. VirtualEnvWrapper

2.2.1 Prepare system for virtual environment

1. Install necessary packages

```
# Install necessary packages
pip install virtualenv virtualenvwrapper
```

Windows users must use `virtualenvwrapper-win` instead of the `virtualenvwrapper` mentioned above. Alternately, you might want to install these packages via `apt-get` or `yum`.

1. Update your *~/.bashrc* or *~/.profile* with the following lines:

```
export WORKON_HOME=~/ENVS/
source /usr/local/bin/virtualenvwrapper.sh
```

Adjust the location of `virtualenvwrapper.sh` file according to your system installation location.

2.2.1.1 Useful virtualenvwrapper commands

- `mkvirtualenv` - Create a new virtual environment
- `workon` - Activate a previously created virtualenv, or switch between environments.
- `deactivate` - Deactive the current virtual environment
- `rmvirtualenv` - Delete an existing virtual environment
- `lsvirtualenv` - List existing virtual environments

2.2.2 Install CPL

1. Obtain the CPL source from the public Git repository.

```
# Change to directory where you want to develop/store sources
git clone https://bitbucket.org/appliedccm/CPL
cd CPL
```

2. Create a virtual environment with all dependencies for CPL

```
# Create a caelus Python 2.7 environment
mkvirtualenv -a $(pwd) -r requirements.txt caelus2
```

3. Activate virtual environment and install CPL into it

```
# Ensure that we are in the right environment
workon caelus2
pip install . # Install CPL within this environment
```

Note:

1. Use `--system-site-packages` with the `mkvirtualenv` command to reuse python modules installed in the system (e.g., via `apt-get`) instead of reinstalling packages locally within the environment.
2. Use `mkvirtualenv --python=PYTHON_EXE` to customize the python interpreter used by the virutal environment instead of the default python found in your path.

2.3 Check installation

After installing CPL, please open a command line terminal and execute **`caelus -h`** to check if the installation process was completed succesfully. Note that users who didn't use the installer provided by Applied CCM might need to activate their *environment* before the `caelus` command is available on their path. If everything was installed and configured successfully, users should see a detailed help message summarizing the usage of **`caelus`**. At this stage, you can either learn about building documentation and executing unit tests (provided with CPL) in the next sections or skip to *Configuring Caelus Python Library* to learn how to configure and use CPL.

2.4 Building documentation

A local version of this documentation can be built using sphinx. See *Install CPL* for more details on installing the developer environment and sources.

```
# Change working directory to CPL
cd docs/

# Build HTML documentation
make html
# View in browser
open build/html/index.html

# Build PDF documentation
make latexpdf
open build/latex/CPL.pdf
```

2.5 Running tests

The unit tests are written using py.test. To run the tests executing **`py.test tests`** from the top-level CPL directory. Note that this will require the user to have initialized the environment using `etc/caelus2-dev.yml` (or `etc/caelus3-dev.yml` for the Python v3.x version).

CHAPTER 3

Configuring Caelus Python Library

CPL provides a YAML-based configuration utility that can be used to customize the library depending on the operating system and user's specific needs. A good example is to provide non-standard install locations for the Caelus CML executables, as well as using different versions of CML with CPL simultaneously.

The use of configuration file is optional, CPL provides defaults that should work on most systems and will attempt to auto-detect CML installations on standard paths. On Linux/OS X systems, CPL will look at `~/Caelus/caelus-VERSION` to determine the installed CML versions and use the `VERSION` tag to determine the latest version to use. On Window systems, the default search path is `C:\Caelus`.

Upon invocation, CPL will search and load configuration files from the following locations, if available. The files are loaded in sequence shown below and options found in succeeding files will overwrite configuration options found in preceeding files.

1. Default configuration supplied with CPL;
2. The system-wide configuration in file pointed by environment variable `CAELUSRC_SYSTEM` if it exists;
3. The per-user configuration file, if available. On Linux/OS X, this is the file `~/.caelus/caelus.yaml`, and `%APPDATA%/caelus/caelus.yaml` on Windows systems;
4. The per-user configuration file pointed by the environment variable `CAELUSRC` if it exists;
5. The file `caelus.yaml` in the current working directory, if it exists.

While CPL provides a way to auto-discovered installed CML versions, often it will be necessary to provide at least a system-wide or per-user configuration file to allow CPL to use the right CML executables present in your system. A sample CPL configuration is shown below `download caelus.yaml`:

```
# -*- mode: yaml -*-
#
# Sample CPL configuration file
#

# Root CPL configuration node
caelus:
  # Control logging of CPL library
  logging:
```

```
    log_to_file: true
    log_file: ~/Caelus/cpl.log

  # Configuration for Caelus CML
  caelus_cml:
    # Pick the development version of CML available; use "latest" to choose the
    # latest version available.
    default: "7.04"

    # Versions that can be used with CPL
    versions:
      - version: "6.10"
        path: ~/Caelus/caelus-6.10

      - version: "7.04"
        path: ~/Caelus/caelus-7.04

      - version: "dev-clang"
        path: ~/Caelus/caelus-contributors   # Use latest git repository
        mpi_path: /usr/local/openmpi         # Use system OpenMPI
        build_option: "linux64clang++DPOpt"  # Use the LLVM version

      - version: "dev-gcc"
        path: ~/Caelus/caelus-contributors   # Use latest git repository
        mpi_path: /usr/local/openmpi         # Use system OpenMPI
        build_option: "linux64gcc++DPOpt"    # Use the GCC version
```

The above configuration would be suitable as as a system-wide or per-user configuration stored in the home directory, and the user can override specific options used for particular runs by using, for example, the following `caelus.yaml` within the case directory:

```
# Local CPL settings for this working directory
caelus:
  logging:
    log_file: cpl_dev.log  # Change log file to a local file

  caelus_cml:
    default: "dev-gcc"     # Use the latest dev version for this run
```

Note that only options that are being overridden need to be specified. Other options are populated from the system-wide or per-user configuration file if they exist.

3.1 Checking current configuration

To aid debugging and troubleshooting, CPL provides a command **`caelus cfg`** to dump the configuration used by the library based on all available configuration files. A sample usage is shown here:

```
$ caelus -v cfg
DEBUG: Loaded configuration from files = ['/home/caelus/.caelus/caelus.yaml']
INFO: Caelus Python Library (CPL) v0.1.0
# -*- mode: yaml -*-
#
# Caelus Python Library (CPL) v0.1.0
#
# Auto-generated on: 2018-04-21 17:03:35 (UTC)
```

```
#

caelus:
  cpl:
    python_env_type: conda
    python_env_name: caelus
    conda_settings:
      conda_bin: ~/anaconda/bin
  system:
    job_scheduler: local_mpi
    always_use_scheduler: false
    scheduler_defaults:
      join_outputs: true
      shell: /bin/bash
      mail_opts: NONE
  logging:
    log_to_file: true
    log_file: null
  caelus_cml:
    default: latest
    versions: []
```

The **final** configuration after parsing all available configuration files is shown in the output. If the user provides `-v` (verbose) flag, then the command also prints out all the configuration files that were detected and read during the initialization process. Users can also use *`caelus cfg`* to create a configuration file with all the current settings using the `-f` option. Please see *caelus* command documentation for details.

3.2 CPL configuration reference

CPL configuration files are in YAML format and must contain at least one node *`caelus`*. Two other optional nodes can be present in the file, *`caelus_scripts`* and *`caelus_user`* whose purpose is described below.

`caelus`
: The root YAML node containing the core CPL configuration object. This node contains all configuration options used internally by the library.

`caelus_scripts`
: An optional node used to store configuration for CPL CLI apps.

`caelus_user`
: An optional node node reserved for user scripts and applications that will be built upon CPL.

Note: In the following sections, the configuration parameters are documented in the format `root_note.sub_node.config_parameter`. Please see the sample configuration file above for the exact nesting structure used for *`caelus.logging.log_file`*.

3.2.1 Core library configuration

3.2.1.1 Python environment options

`caelus.cpl`
: This section contains options to configure the python environment (either Anaconda/Conda environment or

virtualenv settings).

caelus.cpl.python_env_type
Type of python environment. Currently this can be either `conda` or `virtualenv`.

caelus.cpl.python_env_name
The name of the Python environment for use with CPL, e.g., `caelus2` or `caelus-dev`.

caelus.cpl.conda_settings
Extra information for Conda installation on your system.

3.2.1.2 System configuration

caelus.system
This section provides CPL with necessary information on the system settings, particularly the queue configuration on HPC systems.

caelus.system.job_scheduler
The type of job-scheduler available on the system and used by CPL when executing CML executables on the system. By default, all parallel jobs will use the job scheduler, user can configure even serial jobs (e.g., mesh generation, domain decomposition and reconstruction) be submitted on queues.

Name	Description
`local_mpi`	No scheduler, submit locally
`slurm`	Use SLURM commands to submit jobs

caelus.system.always_use_scheduler
A Boolean flag indicating whether even serial jobs (e.g., mesh generation) should use the queue system. This flag is useful when the user intends to generate large meshes and requires access to the high-memory compute nodes on the HPC system.

caelus.system.scheduler_defaults
This section contains options that are used by default when submitting jobs to an HPC queue system.

Option	Description
`queue`	Default queue for submitting jobs
`account`	Account for charging core hours
`stdout`	Default file pattern for redirecting standard output
`stdout`	Default file pattern for redirecting standard error
`join_outputs`	Join `stdout` and `stderr` (queue specific)
`mail_options`	A string indicating mail options for queue
`email_address`	Address where notifications should be sent
`time_limit`	Wall clock time limit

Note: Currently, these options accept strings and are specific to the queue system (e.g., SLURM or PBS Torque). So the user must consult their queue system manuals for appropriate values to these options.

3.2.1.3 CPL logging options

caelus.logging
This section of the configuration file controls the logging options for the CPL library. By default, CPL only outputs messages to the standard output. Users can optionally save all messages from CPL into a log file of

their choice. This is useful for tracking and troubleshooting, or providing additional information regarding bugs observed by the user.

Internally, CPL uses the logging module. For brevity, messages output to console are usually at log levels `INFO` or higher. However, all messages `DEBUG` and above are captured in log files.

caelus.logging.log_to_file
A Boolean value indicating whether CPL should output messages to the log file. The default value is `false`. If set to `true`, then the log messages will also be saved to the file indicated by *log_file* as well as output to the console.

caelus.logging.log_file
Filename where the log messages are saved if *log_to_file* evaluates to `True`.

3.2.2 CML version configuration

caelus.caelus_cml
The primary purpose of CPL is to interact with CML executables and utilities. This section informs CPL of the various CML installations available on a system and the desired *version* used by CPL when invoking CML executables.

caelus.caelus_cml.default
A string parameter indicating default version used when invoking CML executables. It must be one of the *version* entries provided in the file. Alternately, the user can specify `latest` to indicate that the latest version must be used. If users rely on auto-discovery of Caelus versions in default install locations, then it is recommended that this value be `latest` so that CPL picks the latest CML version. For example, with the following configuration, CPL will choose version `7.04` when attempting to execute programs like `pisoSolver`.

```
caelus:
  caelus_cml:
    default: "latest"

    versions:
      - version: "6.10"
        path: ~/Caelus/caelus-6.10

      - version: "7.04"
        path: ~/Caelus/caelus-7.04
```

caelus.caelus_cml.versions
A list of configuration mapping listing various versions available for use with CPL. It is recommended that the users only provide *version* and *path* entries, the remaining entries are optional. CPL will auto-detect remaining parmeters.

caelus.caelus_cml.versions.version
A unique string identifier that is used to tag this specific instance of CML installation. Typically, this is the version number of the Caelus CML release, e.g., `7.04`. However, as indicated in the example CPL configuration file, users can use any unique tag to identify a specific version. If is identifier does not follow the conventional version number format, then it is recommended that the user provide a specific version in *caelus.caelus_cml.default* instead of using `latest`.

caelus.caelus_cml.versions.path
The path to the Caelus install. This is equivalent to the directory pointed by the `CAELUS_PROJECT_DIR` environment variable, e.g., `/home/caelus_user/projects/caelus/caelus-7.04`.

caelus.caelus_cml.versions.build_option
A string parameter identifying the Caelus build, if multiple builds are present within a CML install, to be used

with CPL. This is an **expert** only option used by developers who are testing multiple compilers and build options. It is recommended that the normal users let CPL autodetect the build option.

caelus.caelus_cml.versions.mpi_root
Path to the MPI installation used to compile Caelus for parallel execution. By default, CPL expects the MPI library to be present within the project directory.

caelus.caelus_cml.versions.mpi_bin_path
Directory containing MPI binaries used for **mpiexec** when executing in parallel mode. If absent, CPL will assume that the binaries are located within the subdirectory `bin` in the path pointed by *mpi_root*.

caelus.caelus_cml.versions.mpi_lib_path
Directory containing MPI libraries used for **mpiexec** when executing in parallel mode. If absent, CPL will assume that the libraries are located within the subdirectory `lib` in the path pointed by *mpi_root*.

CHAPTER 4

Command-line Applications

CPL provides command-line interface (CLI) to several frequently used workflows without having to write custom python scripts to access features within the library. These CLI apps are described in detail in the following sections.

4.1 Common CLI options

All CPL command-line applications support a few common options. These options are described below:

-h, --help
: Print a brief help message that describes the purpose of the application and what options are available when interacting with the application.

--version
: Print the CPL version number and exit. Useful for submitting bug-reports, etc.

-v, --verbose
: Increase the verbosity of messages printed to the standard output. Use `-vv` and `-vvv` to progressively increase verbosity of output.

--no-log
: Disable logging messages from the script to a log file.

--cli-logs `log_file`
: Customize the filename used to capture log messages during execution. This overrides the configuration parameter *log_file* provided in the user configuration files.

4.2 Available command-line applications

4.2.1 `caelus` – Common CPL actions

New in version 0.0.2.

The *caelus* command provides various sub-commands that can be used to perform common tasks using the CPL library. Currently the following sub-commands (or actions) are available through the **caelus** script.

Action	Purpose
cfg	Print CPL configuration to stdout or file
env	Generate an environment file for sourcing within bash or csh shell
clone	Clone a case directory
tasks	Automatic execution of workflow from a YAML file
run	Run a CML executable in the appropriate environment
logs	Parse a solver log file and extract data for analysis
clean	Clean a case directory after execution

Note: The script also supports the *common options* documented in the previous section. Care must be take to include the common options before the subcommand, i.e.,

```
# Correct usage
caelus -vvv cfg -f caelus.yaml

# The following will generate an error
# caelus cfg -vvv # ERROR
```

4.2.1.1 caelus cfg – Print CPL configuration

Print out the configuration dictionary currently in use by CPL. This will be a combination of all the options loaded from the configuration files described in *configuration* section. By default, the command prints the YAML-formatted dictionary to the standard output. The output can be redirected to a file by using the *caelus cfg -f* option. This is useful for debugging.

```
$ caelus cfg -h
usage: caelus cfg [-h] [-f CONFIG_FILE] [-b]

Dump CPL configuration

optional arguments:
  -h, --help            show this help message and exit
  -f CONFIG_FILE, --config-file CONFIG_FILE
                        Write to file instead of standard output
  -b, --no-backup       Overwrite existing config without saving a backup
```

-f output_file, **--config-file** output_file
: Save the current CPL configuration to the output_file instead of printing to standard output.

-b, --no-backup
: By default, when using the *caelus cfg -f* CPL will create a backup of any existing configuration file before writing a new file. This option overrides the behavior and will not create backups of existing configurations before overwriting the file.

4.2.1.2 caelus env – write shell environment file

Write a shell environment file to be sourced/called by the platform specific shell. This will be a combination of all the options loaded from the configuration files described in *configuration* section. The output can be redirected to a directory by using the *caelus env -d* option. This is useful for legacy workflows.

```
$ caelus env -h
usage: caelus env [-h] [-d WRITE_DIR]

Write environment variables that can be sourced into the SHELL environment

optional arguments:
  -h, --help            show this help message and exit
  -d WRITE_DIR, --write-dir WRITE_DIR
                        Path where the environment files are written
```

-d write_dir, **--write_dir** write_dir
Save the environment file to the write_dir instead of the current working directory

4.2.1.3 caelus clone – Clone a case directory

caelus clone takes two mandatory parameters, the source template case directory, and name of the new case that is created. By default, the new case directory is created in the current working directory and must not already exist. CPL will not attempt to overwrite existing directories during clone.

```
$ caelus clone -h
usage: caelus clone [-h] [-m] [-z] [-s] [-e EXTRA_PATTERNS] [-d BASE_DIR]
                    template_dir case_name

Clone a case directory into a new folder.

positional arguments:
  template_dir          Valid Caelus case directory to clone.
  case_name             Name of the new case directory.

optional arguments:
  -h, --help            show this help message and exit
  -m, --skip-mesh       skip mesh directory while cloning
  -z, --skip-zero       skip 0 directory while cloning
  -s, --skip-scripts    skip scripts while cloning
  -e EXTRA_PATTERNS, --extra-patterns EXTRA_PATTERNS
                        shell wildcard patterns matching additional files to
                        ignore
  -d BASE_DIR, --base-dir BASE_DIR
                        directory where the new case directory is created
```

-m, --skip-mesh
Do not copy the constant/polyMesh directory when cloning. The default behavior is to copy the mesh along with the case directory.

-z, --skip-zero
Do not copy the 0 directory during clone. The default behavior copies time t=0 directory.

-s, --skip-scripts
Do not copy any python or YAML scripts during clone.

-e pattern, **--extra-patterns** pattern
A shell-wildcard pattern used to skip additional files that might exist in the source directory that must be skipped while cloning the case directory. This option can be repeated multiple times to provide more than one pattern.

```
# Skip all bash files and text files in the source directory
caelus clone -e "*.sh" -e "*.txt" old_case_dir new_case_dir
```

-d basedir, **--base-dir** basedir
: By default, the new case directory is created in the current working directory. This option allows the user to modify the behavior and create the new case in a different location. Useful for use within scripts.

4.2.1.4 caelus tasks – run tasks from a file

Read and execute tasks from a YAML-formatted file. Task files could be considered recipes or workflows. By default, it reads `caelus_tasks.yaml` from the current directory. The behavior can be modified to read other file names and locations.

```
$ caelus tasks -h
usage: caelus tasks [-h] [-f FILE]

Run pre-defined tasks within a case directory read from a YAML-formatted file.

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  file containing tasks to execute (caelus_tasks.yaml)
```

-f task_file, **--file** task_file
: Execute the task file named `task_file` instead of caelus_tasks.yaml in current working directory

4.2.1.5 caelus run – run a Caelus executable in the appropriate environment

Run a single Caelus application. The application name is the one mandatory argument. Additional command arguments can be specified. The behavior can be modified to enble parallel execution of the application. By default, the application runs from the current directory. This behavior can be modified to specify the case directory. Note: when passing `cmd_args`, `--` is required between `run` and `cmd_name` so the cmd_args are parsed correctly. E.g. `caelus run -- renumberMesh "-overwrite"`

```
$ caelus run -h
usage: caelus run [-h] [-p] [-l LOG_FILE] [-d CASE_DIR]
                  cmd_name [cmd_args [cmd_args ...]]

Run a Caelus executable in the correct environment

positional arguments:
  cmd_name              name of the Caelus executable
  cmd_args              additional arguments passed to command

optional arguments:
  -h, --help            show this help message and exit
  -p, --parallel        run in parallel
  -l LOG_FILE, --log-file LOG_FILE
                        filename to redirect command output
  -d CASE_DIR, --case-dir CASE_DIR
                        path to the case directory
```

-p, --parallel
: Run the executable in parallel

-l log_file, **--log-file** log_file
: By default, a log file named `<application>.log` is created. This option allows the user to modify the behavior and create a differently named log file.

-d casedir, **--case-dir** casedir
By default, executables run from the current working directory. This option allows the user to modify the behavior and specify the path to the case directory.

4.2.1.6 caelus logs – process a Caelus solver log file from a run

Process a single Caelus solver log. The log file name is the one mandatory argument. Additional command arguments can be specified. By default, the log file is found in the current directory and the output is written to `logs` directory. The behavior can be modified to specify the case directory and output directory.

```
$ caelus logs -h
usage: caelus logs [-h] [-l LOGS_DIR] [-d CASE_DIR] [-p] [-f PLOT_FILE] [-w]
                   [-i INCLUDE_FIELDS | -e EXCLUDE_FIELDS]
                   log_file

Process logfiles for a Caelus run

positional arguments:
log_file              log file (e.g., simpleSolver.log)

optional arguments:
-h, --help            show this help message and exit
-l LOGS_DIR, --logs-dir LOGS_DIR
                      directory where logs are output (default: logs)
-d CASE_DIR, --case-dir CASE_DIR
                      path to the case directory
-p, --plot-residuals  generate residual time-history plots
-f PLOT_FILE, --plot-file PLOT_FILE
                      file where plot is saved
-w, --watch           Monitor residuals during a run
-i INCLUDE_FIELDS, --include-fields INCLUDE_FIELDS
                      plot residuals for given fields
-e EXCLUDE_FIELDS, --exclude-fields EXCLUDE_FIELDS
```

-l logs_dir, **--logs-dir** logs_dir
By default, the log files are output to `logs`. This option allows the user to modify the behavior and create a differently named log file output directory.

-d, **case_dir**, **--case-dir** case_dir
By default, the log file is found in the current working directory. This option allows the user to specify the path to the case directory where the log file exists.

-p, **--plot-residuals**
This option allows the user to plot and save the residuals to an image file.

-f plot_file, **--plot-file** plot_file
By default, plots are saved to `residuals.png` in the current working directory. This option allows the user to modify the behavior and specify a differently named plot file.

-w, **--watch**
This option allows the user to dynamically monitor residuals for a log file from a currently run.

-i include_fields, **--include-fields** include_fields
By default, all field equation residuals are plotted. This option can be used to only include specific fields in residual plot. Multiple fields can be provided to this option. For example,

```
# Plot pressure and momentum residuals from simpleSolver case log
caelus logs -p -i "p Ux Uy Uz" simpleSolver.log
```

-e exclude_fields, **--exclude-patterns** exclude fields
By default, all field equation residuals are plotted. This option can be used to exclude specific fields in residual plot. Multiple fields be provided to this option. For example,

```
# Exclude TKE and omega residuals from simpleSolver case log
caelus logs -p -e "k epsilon" simpleSolver.log
```

4.2.1.7 caelus clean – clean a Caelus case directory

Cleans files generated by a run. By default, this function will always preserve `system`, `constant`, and `0` directories as well as any YAML or python files. The behavior can be modified to presevere additional files and directories.

```
$ caelus clean -h
usage: caelus clean [-h] [-d CASE_DIR] [-m] [-z] [-p PRESERVE]

Clean a case directory

optional arguments:
    -h, --help          show this help message and exit
    -d CASE_DIR, --case-dir CASE_DIR
                        path to the case directory
    -m, --clean-mesh    remove polyMesh directory
    -z, --clean-zero    remove 0 directory
    -p PRESERVE, --preserve PRESERVE
                        shell wildcard patterns of extra files to preserve
```

-d, case_dir, --case-dir case_dir
By default, the case directory is the current working directory. This option allows the user to specify the path to the case directory.

-m, --clean-mesh
By default, the `polyMesh` directory is not removed. This option allows the user to modify the behavior and remove the `polyMesh` directory.

-z, --clean-zero
By default, the `0` files are not cleaned. This option allows the user to modify the behavior and remove the `0` directory.

-p preserve_pattern, **--preserve** preserve_pattern
A shell-wildcard patterns of files or directories that will not be cleaned.

4.2.2 caelus_tutorials – Run tutorials

This is a convenience command to automatically run tutorials provided within the Caelus CML distribution.

```
$ caelus_tutorials -h
usage: caelus_tutorials [-h] [--version] [-v] [--no-log | --cli-logs CLI_LOGS]
                        [-d BASE_DIR] [-c CLONE_DIR] [-f TASK_FILE]
                        [-i INCLUDE_PATTERNS | -e EXCLUDE_PATTERNS]

Run Caelus Tutorials

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -v, --verbose         increase verbosity of logging. Default: No
```

```
  --no-log              disable logging of script to file.
  --cli-logs CLI_LOGS   name of the log file (caelus_tutorials.log)
  -d BASE_DIR, --base-dir BASE_DIR
                        directory where tutorials are run
  -c CLONE_DIR, --clone-dir CLONE_DIR
                        copy tutorials from this directory
  --clean               clean tutorials from this directory
  -f TASK_FILE, --task-file TASK_FILE
                        task file containing tutorial actions
                        (run_tutorial.yaml)
  -i INCLUDE_PATTERNS, --include-patterns INCLUDE_PATTERNS
                        run tutorial case if it matches the shell wildcard
                        pattern
  -e EXCLUDE_PATTERNS, --exclude-patterns EXCLUDE_PATTERNS
                        exclude tutorials that match the shell wildcard
                        pattern

Caelus Python Library (CPL) v0.0.2
```

-f task_file, **--task-file** task_file
: The name of the task file used to execute the steps necessary to complete a tutorial. The default value is `run_tutorial.yaml`

-i pattern, **--include-patterns** pattern
: A shell wildcard pattern to match tutorial names that must be executed. This option can be used multiple times to match different patterns. For example,

```
# Run all simpleSolver cases and pisoSolver's cavity case
caelus_tutorials -i "*simpleSolver* -i "*cavity*"
```

This option is mutually exclusive to *caelus_tutorials -e*

-e pattern, **--exclude-patterns** pattern
: A shell wildcard pattern to match tutorial names that are skipped during the tutorial run. This option can be used multiple times to match different patterns. For example,

```
# Skip motorBikeSS and motorBikeLES cases
caelus_tutorials -e "*motorBike*"
```

This option is mutually exclusive to *caelus_tutorials -i*

CHAPTER 5

Caelus Tasks

CPL provides a *tasks* interface to automate various aspects of the CFD simulation workflow that can be executed by calling **`caelus tasks`** (see *tasks documentation*).

5.1 Quick tutorial

The *tasks* interface requires a list of tasks provided in a YAML-formatted file as shown below (`download`):

```
tasks:
  - clean_case:
      remove_zero: no
      remove_mesh: yes

  - run_command:
      cmd_name: blockMesh

  - run_command:
      cmd_name: pisoSolver

  - process_logs:
      log_file: pisoSolver.log
      plot_residuals: true
      residuals_plot_file: residuals.pdf
      residuals_fields: [Ux, Uy]
```

The file lists a set of actions to be executed sequentially by **`caelus tasks`**. The tasks can accept various options that can be used to further customize the workflow. A sample interaction is shown below

```
$ caelus -v tasks -f caelus_tasks.yaml
INFO: Caelus Python Library (CPL) v0.1.0
INFO: Caelus CML version: 7.04
INFO: Loaded tasks from: cavity/caelus_tasks.yaml
INFO: Begin executing tasks in cavity
```

```
INFO: Cleaning case directory: cavity
INFO: Executing command: blockMesh
INFO: Executing command: pisoSolver
INFO: Processing log file: pisoSolver.log
INFO: Saved figure: cavity/residuals.pdf
INFO: Residual time history saved to residuals.pdf
INFO: Successfully executed 4 tasks in cavity
INFO: All tasks executed successfully.
```

For a comprehensive list of task file examples, please consult the `run_tutorial.yaml` files in the `tutorials` directory of Caelus CML distribution. In particular, the `tutorials/incompressible/pimpleSolver/les/motorBike` case provides an example of a tasks workflow involving two different case directories.

5.2 Tasks reference

This section documents the various *tasks* available currently within CPL and the options that can be used to customize execution of those tasks.

- The task file must be in YAML format, and must contain one entry `tasks` that is a list of tasks to be executed.
- The tasks are executed sequentially in the order provided until an error is encountered or all tasks are executed successfully.
- The tasks must be invoked from within a valid Caelus case directory (see `task_set` for an exception). All filenames in the task file are interpreted relative to the execution directory where the command is invoked.

5.2.1 run_command – Run CML executables

This *task type* is used to execute a Caelus CML executable (e.g., **`blockMesh`** or **`pimpleSolver`**). CPL will ensure that the appropriate version of CML is selected and the runtime enviornment is setup properly prior to executing the task. The task must provide one mandatory parameter *`run_command.cmd_name`* that is the name of the CML executable. Several other options are available and are documented below. Example:

```
- run_command:
    cmd_name: potentialSolver
    cmd_args: "-initialiseUBCs -noFunctionObjects"
    parallel: true
```

`run_command.cmd_name`
: The name of the CML executable. This option is mandatory.

`run_command.cmd_args`
: Extra arguments that must be passed to the CML executable. It is recommended that arguments be enclosed in a double-quoted string. Default value is an empty string.

`run_command.log_file`
: The filename where the output of the command is redirected. By default, it is the CML executable name with the `.log` extension appended to it. The user can change this to any valid filename of their choice using this option.

`run_command.parallel`
: A Boolean flag indicating whether the executable is to be run in parallel mode. The default value is `False`. If `parallel` is True, then the default options for job scheduler are used from CPL configuration file, but can be overriden with additional options to `run_command`.

run_command.num_ranks
The number of MPI ranks for a parallel run.

run_command.mpi_extra_args
Extra arguments to be passed to **mpiexec** command (e.g., `hostfile` options). As with *cmd_args*, enclose the options within quotes.

5.2.2 copy_files – Copy files

This task copies files in a platform-agnostic manner.

copy_files.src
A unix-style file pattern that is used to match the pattern of files to be copied. The path to the files must be relative to the execution directory, but can exist in other directories as long as the relative paths are provided correctly. If the pattern matches multiple files, then *copy_files.dest* must be a directory.

copy_files.dest
The destination where the files are to be copied.

5.2.3 copy_tree – Recursively copy directories

This task takes an existing directory (`src`) and copies it to the destination. Internally, this task uses copytree function to copy the directory, please refer to Python docs for more details.

> **Warning:** If the destination directory already exists, the directory is deleted before copying the contents of the source directory. Currently, this task does not provide a way to copy only non-existent files to the destination. Use with caution.

copy_tree.src
The source directory that must be recursively copied.

copy_tree.dest
The pathname for the new directory to be created.

copy_tree.ignore_patterns
A list of Unix-style file patterns used to ignore files present in source directory when copying it to destination. A good example of this is to prevent copying the contents of `polyMesh` when copying the contents of `constant` from one case directory to another.

copy_tree.preserve_symlinks
A Boolean flag indicating whether symbolic links are preserved when copying. Linux and Mac OSX only.

5.2.4 clean_case – Clean a case directory

Use this task to clean up a case directory after a run. By default, this task will preserve all YAML and python files found in the case directory as well as the `0/` directory. For example,

```
- clean_case:
    remove_zero: yes
    remove_mesh: no
    preserve: [ "0.org" ]
```

clean_case.remove_zero
Boolean flag indicating whether the `0/` directory should be removed. The default value is `False`.

clean_case.remove_mesh
Boolean flag indicating whether the `constant/polyMesh` directory should be removed. The default value is `False`.

clean_case.preserve
A list of Unix-style file patterns that match files that should be preserved within the case directory.

5.2.5 process_logs – Process solver outputs

This task takes one mandatory argument *log_file* that contains the outputs from a CFD run. The time-histories of the residuals are extracted and output to files that can be loaded by **gnuplot**, or loaded in python using loadtxt command or using Pandas library. Users can also plot the residuals by using the *plot_residuals* option. For example,

```
- process_logs:
  log_file: pimpleSolver.log
  log_directory: pimpleSolver_logs

- process_logs:
  log_file: simpleSolver.log
  plot_residuals: yes
  residuals_plot_file: residuals.pdf
  residuals_fields: [Ux, Uy, p]
```

process_logs.log_file
The filename containing the solver residual ouputs. This parameter is mandatory.

process_logs.logs_directory
The directory where the processed residual time-history outputs are stored. Default: `logs` within the execution directory.

process_logs.plot_residuals
Boolean flag indicating whether a plot of the convergence time-history is generated. Default value is `False`.

process_logs.residuals_plot_file
The file where the plot is saved. Default value is `residuals.png`. The user can use an appropriate extension (e.g., `.png`, `.pdf`, `.jpg`) to change the image format of the plot generated.

process_logs.residual_fields
A list of fields that are plotted. If not provided, all fields available are plotted.

process_logs.plot_continuity_errors
A Boolean flag indicating whether time-history of continuity errors are plotted along with residuals.

5.2.6 task_set – Group tasks

A `task_set` groups a sub-set of tasks that can be executed in a different case directory. `Download` an example.

task_set.case_dir
The path to a valid Caelus case directory where the sub-tasks are to be executed. This parameter is mandatory.

task_set.name
A unique name to identify this task group.

task_set.tasks
The list of sub-tasks. This list can contain any of the tasks that have been documented above.

Part II

Developer Manual

CHAPTER 6

Caelus Python API

6.1 caelus.config – Caelus Configuration Infrastructure

`caelus.config` performs the following tasks:

- Configure the behavior of the Caelus python library using YAML based configuration files.
- Provide an interface to Caelus CML installations and also aid in automated discovery of installed Caelus versions.

get_config	Get the configuration object
reload_config	Reset the configuration object
reset_default_config	Reset to default configuration
cml_get_version	Get the CML environment for the version requested
cml_get_latest_version	Get the CML environment for the latest version available.
CMLEnv	CML Environment Interface.

6.1.1 Caelus Python Configuration

The *config* module provides functions and classes for loading user configuration via YAML files and a central location to configure the behavior of the Caelus python library. The user configuration is stored in a dictionary format within the *CaelusCfg* and can be modified during runtime by user scripts. Access to the configuration object is by calling the *get_config()* method defined within this module which returns a fully populated instance of the configuration dictionary. This module also sets up logging (to both console as well as log files) during the initialization phase.

class caelus.config.config.**CaelusCfg**(**args*, ***kwds*)

Bases: *caelus.utils.struct.Struct*

Caelus Configuration Object

A (key, value) dictionary containing all the configuration data parsed from the user configuration files. It is recommended that users obtain an instance of this class via the *get_config()* function instead of directly

instantiating this class.

Initialize an ordered dictionary. The signature is the same as regular dictionaries, but keyword arguments are not recommended because their insertion order is arbitrary.

yaml_decoder
alias of StructYAMLLoader

yaml_encoder
alias of StructYAMLDumper

write_config(*fh=<open file '<stdout>', mode 'w'>*)
Write configuration to file or standard output.

Parameters **fh** (*handle*) – An open file handle

caelus.config.config.**configure_logging**(*log_cfg=None*)
Configure python logging.

If log_cfg is None, then the basic configuration of python logging module is used.

See Python Logging Documentation for more information.

Parameters **log_cfg** – Instance of *CaelusCfg*

caelus.config.config.**get_appdata_dir**()
Return the path to the Windows APPDATA directory

caelus.config.config.**get_caelus_root**()
Get Caelus root directory

In Unix-y systems this returns ${HOME}/Caelus and on Windows it returns C:\Caelus.

Returns Path to Caelus root directory

Return type path

caelus.config.config.**get_config**(*base_cfg=None*, *init_logging=False*)
Get the configuration object

On the first call, initializes the configuration object by parsing all available configuration files. Successive invocations return the same object that can be mutated by the user. The config dictionary can be reset by invoking *reload_config()*.

Parameters

- **base_cfg** (CaelusCfg) – A base configuration object that is updated
- **init_logging** (*bool*) – If True, initializes logging

Returns The configuration dictionary

Return type *CaelusCfg*

caelus.config.config.**get_cpl_root**()
Return the root path for CPL

caelus.config.config.**get_default_config**()
Return a fresh instance of the default configuration

This function does not read the caelus.yaml files on the system, and returns the configurations shipped with CPL.

Returns The default configuration

Return type *CaelusCfg*

caelus.config.config.**rcfiles_loaded**()
Return a list of the configuration files that were loaded

caelus.config.config.**reload_config**(*base_cfg=None*)
Reset the configuration object

Forces reloading of all the available configuration files and resets the modifications made by user scripts.

See also: *reset_default_config()*

Parameters **base_cfg** – A CMLEnv object to use instead of default

Returns The configuration dictionary

Return type *CaelusCfg*

caelus.config.config.**reset_default_config**()
Reset to default configuration

Resets to library default configuration. Unlike *reload_config()*, this function does not load the configuration files.

Returns The configuration dictionary

Return type *CaelusCfg*

caelus.config.config.**search_cfg_files**()
Search locations and return all possible configuration files.

The following locations are searched:

- The path pointed by CAELUSRC_SYSTEM
- The user's home directory ~/.caelus/caelus.yaml on Unix-like systems, and %APPDATA%/caelus/caelus.yaml on Windows systems.
- The path pointed by CAELUSRC, if defined.
- The file caelus.yaml in the current working directory

Returns List of configuration files available

6.1.2 Caelus CML Environment Manager

cmlenv serves as a replacement for Caelus/OpenFOAM bashrc files, providing ways to discover installed versions as well as interact with the installed Caelus CML versions. By default, cmlenv attempts to locate installed Caelus versions in standard locations: ~/Caelus/caelus-VERSION on Unix-like systems, and in C:Caelus in Windows systems. Users can override the default behavior and point to non-standard locations by customizing their Caelus Python configuration file.

class caelus.config.cmlenv.**CMLEnv**(*cfg*)
Bases: object

CML Environment Interface.

This class provides an interface to an installed Caelus CML version.

Parameters **cfg** (CaelusCfg) – The CML configuration object

bin_dir
Return the bin directory for executable

build_dir
Return the build platform directory

environ
Return an environment for running Caelus CML binaries

lib_dir
Return the bin directory for executable

mpi_bindir
Return the MPI executables path for this installation

mpi_dir
Return the MPI directory for this installation

mpi_libdir
Return the MPI library path for this installation

project_dir
Return the project directory path

Typically `~/Caelus/caelus-VERSION`

root
Return the root path for the Caelus install

Typically on Linux/OSX this is the `~/Caelus` directory.

version
Return the Caelus version

caelus.config.cmlenv.**cml_get_latest_version**()
Get the CML environment for the latest version available.

Returns The environment object

Return type *CMLEnv*

caelus.config.cmlenv.**cml_get_version**(*version=None*)
Get the CML environment for the version requested

If version is None, then it returns the version set as default in the configuration file.

Parameters **version** (*str*) – Version string

Returns The environment object

Return type *CMLEnv*

caelus.config.cmlenv.**discover_versions**(*root=None*)
Discover Caelus versions if no configuration is provided.

If no root directory is provided, then the function attempts to search in path provided by *get_caelus_root()*.

Parameters **root** (*path*) – Absolute path to root directory to be searched

6.2 caelus.utils – Basic utilities

Collection of low-level utilities that are accessed by other packages within CPL, and other code snippets that do not fit elsewhere within CPL. The modules present within utils package must only depend on external libraries or other modules within util, they must not import modules from other packages within CPL.

Struct	Dictionary that supports both key and attribute access.

Continued on next page

Table 6.2 – continued from previous page

osutils	Miscellaneous utilities

6.2.1 Struct Module

Implements *Struct*.

class caelus.utils.struct.**Struct**(**args*, ***kwds*)

Bases: collections.OrderedDict, _abcoll.MutableMapping

Dictionary that supports both key and attribute access.

Struct is inspired by Matlab `struct` data structure that is intended to support both key and attribute access. It has the following features:

1. Preserves ordering of members as initialized
2. Provides attribute and dictionary-style lookups
3. Read/write YAML formatted data

Initialize an ordered dictionary. The signature is the same as regular dictionaries, but keyword arguments are not recommended because their insertion order is arbitrary.

yaml_decoder

alias of StructYAMLLoader

yaml_encoder

alias of StructYAMLDumper

classmethod from_yaml(*stream*)

Initialize mapping from a YAML string.

Parameters **stream** – A string or valid file handle

Returns YAML data as a python object

Return type *Struct*

classmethod load_yaml(*filename*)

Load a YAML file

Parameters **filename** (*str*) – Absolute path to YAML file

Returns YAML data as python object

Return type *Struct*

merge(**args*)

Recursively update dictionary

Merge entries from maps provided such that new entries are added and existing entries are updated.

to_yaml(*stream=None*, *default_flow_style=False*, ***kwargs*)

Convert mapping to YAML format.

Parameters

- **stream** (*file*) – A file handle where YAML is output
- **default_flow_style** (*bool*) –
  - False - pretty printing
  - True - No pretty printing

class caelus.utils.struct.**StructMeta**
Bases: abc.ABCMeta

YAML interface registration

Simplify the registration of custom yaml loader/dumper classes for Struct class hierarchy.

caelus.utils.struct.**gen_yaml_decoder**(*cls*)
Generate a custom YAML decoder with non-default mapping class

Parameters **cls** – Class used for mapping

caelus.utils.struct.**gen_yaml_encoder**(*cls*)
Generate a custom YAML encoder with non-default mapping class

Parameters **cls** – Class used for mapping

caelus.utils.struct.**merge**(*a*, *b*, **args*)
Recursively merge mappings and return consolidated dict.

Accepts a variable number of dictionary mappings and returns a new dictionary that contains the merged entries from all dictionaries. Note that the update occurs left to right, i.e., entries from later dictionaries overwrite entries from preceeding ones.

Returns The consolidated map

Return type dict

6.2.2 Miscellaneous utilities

This module implements functions that are utilized throughout CPL. They mostly provide a higher-level interface to various os.path functions to make it easier to perform some tasks.

set_work_dir	A with-block to execute code in a given directory.
ensure_directory	Check if directory exists, if not, create it.
abspath	Return the absolute path of the directory.
ostype	String indicating the operating system type
timestamp	Return a formatted timestamp for embedding in files

caelus.utils.osutils.**abspath**(*pname*)
Return the absolute path of the directory.

This function expands the user home directory as well as any shell variables found in the path provided and returns an absolute path.

Parameters **pname** (*path*) – Pathname to be expanded

Returns Absolute path after all substitutions

Return type path

caelus.utils.osutils.**backup_file**(*fname*, *time_format=None*, *time_zone=<UTC>*)
Given a filename return a timestamp based backup filename

Parameters

- **time_format** – A time formatter suitable for strftime
- **time_zone** – Time zone used to generate timestamp (Default: UTC)

Returns A timestamped filename suitable for creating backups

Return type str

caelus.utils.osutils.**clean_directory**(*dirname*, *preserve_patterns=None*)
Utility function to remove files and directories from a given directory.

User can specify a list of filename patterns to preserve with the `preserve_patterns` argument. These patterns can contain shell wildcards to glob multiple files.

Parameters

- **dirname** (*path*) – Absolute path to the directory whose entries are purged.
- **preserve_patterns** (*list*) – A list of shell wildcard patterns

caelus.utils.osutils.**copy_tree**(*srcdir*, *destdir*, *symlinks=False*, *ignore_func=None*)
Enchanced version of shutil.copytree

- removes the output directory if it already exists.

Parameters

- **srcdir** (*path*) – path to source directory to be copied.
- **destdir** (*path*) – path (or new name) of destination directory.
- **symlinks** (*bool*) – as in shutil.copytree
- **ignore_func** (*func*) – as in shutil.copytree

caelus.utils.osutils.**ensure_directory**(*dname*)
Check if directory exists, if not, create it.

Parameters **dname** (*path*) – Directory name to check for

Returns Absolute path to the directory

Return type Path

caelus.utils.osutils.**ostype**()
String indicating the operating system type

Returns One of [“linux”, “darwin”, “windows”]

Return type str

caelus.utils.osutils.**remove_files_dirs**(*paths*, *basedir=None*)
Remove files and/or directories

Parameters

- **paths** (*list*) – A list of file paths to delete (no patterns allowed)
- **basedir** (*path*) – Base directory to search

caelus.utils.osutils.**set_work_dir**(**args*, ***kwds*)
A with-block to execute code in a given directory.

Parameters

- **dname** (*path*) – Path to the working directory.
- **create** (*bool*) – If true, directory is created prior to execution

Returns Absolute path to the execution directory

Return type path

Example

```
>>> with osutils.set_work_dir("results_dir", create=True) as wdir:
...     with open(os.path.join(wdir, "results.dat"), 'w') as fh:
...         fh.write("Data")
```

caelus.utils.osutils.**timestamp**(*time_format=None*, *time_zone=<UTC>*)
Return a formatted timestamp for embedding in files

Parameters

- **time_format** – A time formatter suitable for strftime
- **time_zone** – Time zone used to generate timestamp (Default: UTC)

Returns A formatted time string

Return type str

caelus.utils.osutils.**user_home_dir**()
Return the absolute path of the user's home directory

caelus.utils.osutils.**username**()
Return the username of the current user

6.3 caelus.run – CML Execution Utilities

6.3.1 Caelus Tasks Manager

class caelus.run.tasks.**Tasks**
Bases: object

Caelus Tasks.

Tasks provides a simple automated workflow interface that provides various pre-defined actions via a YAML file interface.

The tasks are defined as methods with a cmd_ prefix and are automaticaly converted to task names. Users can create additional tasks by subclassing and adding additional methods with cmd_ prefix. These methods accept one argument options, a dictionary containing parameters provided by the user for that particular task.

cmd_clean_case(*options*)
Clean a case directory

cmd_copy_files(*options*)
Copy given file(s) to the destination.

cmd_copy_tree(*options*)
Recursively copy a given directory to the destination.

cmd_exec_tasks(*options*)
Execute another task file

cmd_process_logs(*options*)
Process logs for a case

cmd_run_command(*options*)
Execute a Caelus CML binary.

cmd_task_set(*options*)
A subset of tasks for grouping

classmethod load(*task_file='caelus_tasks.yaml'*, *task_node='tasks'*)
Load tasks from a YAML file.

If `exedir is None` then the execution directory is set to the directory where the tasks file is found.

Parameters **task_file** (*filename*) – Path to the YAML file

case_dir = None
Directory where the tasks are to be executed

env = None
Caelus environment used when executing tasks

task_file = None
File that was used to load tasks

tasks = None
List of tasks that must be performed

class caelus.run.tasks.**TasksMeta**(*name*, *bases*, *cdict*)
Bases: type

Process available tasks within each Tasks class.

TasksMeta is a metaclass that automates the process of creating a lookup table for tasks that have been implemented within the *Tasks* and any of its subclasses. Upon initialization of the class, it populates a class attribute `task_map` that contains a mapping between the task name (used in the tasks YAML file) and the corresponding method executed by the Tasks class executed.

6.3.2 CML Execution Utilities

caelus.run.core.**clean_casedir**(*casedir*, *preserve_extra=None*, *preserve_zero=True*, *purge_mesh=False*)
Clean a Caelus case directory.

Cleans files generated by a run. By default, this function will always preserve `system`, `constant`, and `0` directories as well as any YAML or python files. Additional files and directories can be preserved by using the `preserve_extra` option that accepts a list of shell wildcard patterns of files/directories that must be preserved.

Parameters

- **casedir** (*path*) – Absolute path to a case directory.
- **preserve_extra** (*list*) – List of shell wildcard patterns to preserve
- **purge_mesh** (*bool*) – If true, also removes mesh from constant/polyMesh
- **preserve_zero** (*bool*) – If False, removes the 0 directory

Raises IOError – `clean_casedir` will refuse to remove files from a directory that is not a valid Caelus case directory.

caelus.run.core.**clean_polymesh**(*casedir*, *region=None*, *preserve_patterns=None*)
Clean the polyMesh from the given case directory.

Parameters

- **casedir** (*path*) – Path to the case directory
- **region** (*str*) – Mesh region to delete

- **preserve_patterns** (*list*) – Shell wildcard patterns of files to preserve

caelus.run.core.**clone_case**(*casedir*, *template_dir*, *copy_polymesh=True*, *copy_zero=True*, *copy_scripts=True*, *extra_patterns=None*)

Clone a Caelus case directory.

Parameters

- **casedir** (*path*) – Absolute path to new case directory.
- **template_dir** (*path*) – Case directory to be cloned
- **copy_polymesh** (*bool*) – Copy contents of constant/polyMesh to new case
- **copy_zero** (*bool*) – Copy time=0 directory to new case
- **copy_scripts** (*bool*) – Copy python and YAML files
- **extra_patterns** (*list*) – List of shell wildcard patterns for copying

Returns Absolute path to the newly cloned directory

Return type path

Raises IOError – If either the `casedir` exists or if the `template_dir` does not exist or is not a valid Caelus case directory.

caelus.run.core.**find_caelus_recipe_dirs**(*basedir*, *action_file='caelus_tasks.yaml'*)

Return case directories that contain action files.

A case directory with action file is determined if the directory succeeds checks in `is_caelus_dir()` and also contains the action file specified by the user.

Parameters

- **basedir** (*path*) – Top-level directory to traverse
- **action_file** (*filename*) – Default is `caelus_tasks.yaml`

Yields Path to the case directory with action files

caelus.run.core.**find_case_dirs**(*basedir*)

Recursively search for case directories existing in a path.

Parameters **basedir** (*path*) – Top-level directory to traverse

Yields Absolute path to the case directory

caelus.run.core.**find_recipe_dirs**(*basedir*, *action_file='caelus_tasks.yaml'*)

Return directories that contain the action files

This behaves differently than *find_caelus_recipe_dirs()* in that it doesn't require a valid case directory. It assumes that the case directories are sub-directories and this task file acts on multiple directories.

Parameters

- **basedir** (*path*) – Top-level directory to traverse
- **action_file** (*filename*) – Default is `caelus_tasks.yaml`

Yields Path to the case directory with action files

caelus.run.core.**get_mpi_size**(*casedir*)

Determine the number of MPI ranks to run

caelus.run.core.**is_caelus_casedir**(*root=None*)

Check if the path provided looks like a case directory.

A directory is determined to be an OpenFOAM/Caelus case directory if the `system`, `constant`, and `system/controlDict` exist. No check is performed to determine whether the case directory will actually run or if a mesh is present.

Parameters **root** (*path*) – Top directory to start traversing (default: CWD)

6.3.3 Job Scheduler Interface

This module provides a unified interface to submitting serial, local-MPI parallel, and parallel jobs on high-performance computing (HPC) queues.

class caelus.run.hpc_queue.**HPCQueue**(*name*, *cml_env=None*, ***kwargs*)

Abstract base class for job submission interface

name
str – Job name

queue
str – Queue/partition where job is submitted

account
str – Account the job is charged to

num_nodes
int – Number of nodes requested

num_ranks
int – Number of MPI ranks

stdout
path – Filename where standard out is redirected

stderr
path – Filename where standard error is redirected

join_outputs
bool – Merge stdout/stderr to same file

mail_opts
str – Mail options (see specific queue implementation)

email_address
str – Email address for notifications

qos
str – Quality of service

time_limit
str – Wall clock time limit

shell
str – shell to use for scripts

mpi_extra_args
str – additional arguments for MPI

Parameters

- **name** (*str*) – Name of the job
- **cml_env** (CMLEnv) – Environment used for execution

static delete(*job_id*)
Delete a job from the queue

get_queue_settings()
Return a string with all the necessary queue options

static is_job_scheduler()
Is this a job scheduler

static is_parallel()
Flag indicating whether the queue type can support parallel runs

prepare_mpi_cmd()
Prepare the MPI invocation

process_run_env()
Populate the run variables for script

classmethod submit(*script_file*, *job_dependencies=None*, *extra_args=None*, *dep_type=None*)
Submit the job to the queue

update(*settings*)
Update queue settings from the given dictionary

write_script(*script_name=None*)
Write a submission script using the arguments provided

Parameters **script_name** (*path*) – Name of the script file

queue_name = '_ERROR_'
Identifier used for queue

script_body
The contents of the script submitted to scheduler

class caelus.run.hpc_queue.**PBSQueue**(*name*, *cml_env=None*, ***kwargs*)
PBS Queue Interface

Parameters

- **name** (*str*) – Name of the job
- **cml_env** (CMLEnv) – Environment used for execution

static delete(*job_id*)
Delete the PBS batch job using job ID

get_queue_settings()
Return all PBS options suitable for embedding in script

classmethod submit(*script_file*, *job_dependencies=None*, *extra_args=None*, *dep_type='afterok'*)
Submit a PBS job using qsub command

`job_dependencies` is a list of PBS job IDs. The submitted job will run depending the status of the dependencies.

`extra_args` is a dictionary of arguments passed to `qsub` command.

The job ID returned by this method can be used as an argument to delete method or as an entry in `job_dependencies` for a subsequent job submission.

Parameters

- **script_file** (*path*) – Script provided to sbatch command
- **job_dependencies** (*list*) – List of jobs to wait for

- **extra_args** (*dict*) – Extra SLURM arguments

Returns Job ID as a string

Return type str

class caelus.run.hpc_queue.**ParallelJob**(*name*, *cml_env=None*, ***kwargs*)
Interface to a parallel job

Parameters

- **name** (*str*) – Name of the job
- **cml_env** (CMLEnv) – Environment used for execution

static is_parallel()
Flag indicating whether the queue type can support parallel runs

prepare_mpi_cmd()
Prepare the MPI invocation

class caelus.run.hpc_queue.**SerialJob**(*name*, *cml_env=None*, ***kwargs*)
Interface to a serial job

Parameters

- **name** (*str*) – Name of the job
- **cml_env** (CMLEnv) – Environment used for execution

static delete(*job_id*)
Delete a job from the queue

get_queue_settings()
Return queue settings

static is_job_scheduler()
Flag indicating whether this is a job scheduler

static is_parallel()
Flag indicating whether the queue type can support parallel runs

prepare_mpi_cmd()
Prepare the MPI invocation

classmethod submit(*script_file*, *job_dependencies=None*, *extra_args=None*)
Submit the job to the queue

class caelus.run.hpc_queue.**SlurmQueue**(*name*, *cml_env=None*, ***kwargs*)
Interface to SLURM queue manager

Parameters

- **name** (*str*) – Name of the job
- **cml_env** (CMLEnv) – Environment used for execution

static delete(*job_id*)
Delete the SLURM batch job using job ID

get_queue_settings()
Return all SBATCH options suitable for embedding in script

prepare_srun_cmd()
Prepare the call to SLURM srun command

classmethod submit(*script_file*, *job_dependencies=None*, *extra_args=None*, *dep_type='afterok'*)
Submit to SLURM using sbatch command

`job_dependencies` is a list of SLURM job IDs. The submitted job will not run until after all the jobs provided in this list have been completed successfully.

`extra_args` is a dictionary of extra arguments to be passed to `sbatch` command. Note that this can override options provided in the script file as well as introduce additional options during submission.

`dep_type` can be one of: after, afterok, afternotok afterany

The job ID returned by this method can be used as an argument to delete method or as an entry in `job_dependencies` for a subsequent job submission.

Parameters

- **script_file** (*path*) – Script provided to sbatch command
- **job_dependencies** (*list*) – List of jobs to wait for
- **extra_args** (*dict*) – Extra SLURM arguments
- **dep_type** (*str*) – Dependency type

Returns Job ID as a string

Return type str

caelus.run.hpc_queue.**caelus_execute**(*cmd*, *env=None*, *stdout=<open file '<stdout>', mode 'w'>*, *stderr=<open file '<stderr>', mode 'w'>*)
Execute a CML command with the right environment setup

A wrapper around subprocess.Popen to set up the correct environment before invoing the CML executable.

The command can either be a string or a list of arguments as appropriate for Caelus executables.

Examples

caelus_execute("blockMesh -help")

Parameters

- **cmd** (*str or list*) – The command to be executed
- **env** (CMLEnv) – An instance representing the CML installation (default: latest)
- **stdout** – A file handle where standard output is redirected
- **stderr** – A file handle where standard error is redirected

Returns The task instance

Return type subprocess.Popen

caelus.run.hpc_queue.**get_job_scheduler**(*queue_type=None*)
Return an instance of the job scheduler

6.4 caelus.post – Post-processing utilities

Provides log analysis and plotting utilities

SolverLog	Caelus solver log file interface.
CaelusPlot	Caelus Data Plotting Interface

6.4.1 Caelus Log Analyzer

This module provides utilities to parse and extract information from solver outputs (log files) that can be used to monitor and analyze the convergence of runs. It implements the *SolverLog* class that can be used to access time histories of residuals for various fields of interest.

Example

```
>>> logs = SolverLog()
>>> print ("Available fields: ", logs.fields)
>>> ux_residuals = logs.residual("Ux")
```

The actual extraction of the logs is performed by *LogProcessor* which uses regular expressions to match lines of interest and convert them into tabular files suitable for loading with `numpy.loadtxt` or `pandas.read_table`.

class caelus.post.logs.**LogProcessor**(*logfile*, *case_dir=None*, *logs_dir='logs'*)

Bases: object

Process the log file and extract information for analysis.

This is a low-level utility to parse log files and extract information using regular expressions from the log file. Users should interact with solver output using the *SolverLog* class.

Parameters

- **logfile** (*str*) – Name of the Caelus log file
- **casedir** (*path*) – Path to the case directory (default: cwd)
- **logs_dir** (*path*) – Relative path to the directory where logs are written

add_rule(*regexp*, *actions*)

Add a user-defined rule for processing

Parameters

- **regexp** (*str*) – A string that can be compiled into a regexp
- **action** (*func*) – A coroutine that can consume matching patterns

bounding_processor(**args*, ***kwargs*)

Process the bounding lines

completion_processor(**args*, ***kwargs*)

Process End line indicating solver completion

continuity_processor(**args*, ***kwargs*)

Process continuity error lines from log file

convergence_processor(**args*, ***kwargs*)

Process convergence information (steady solvers only)

courant_processor(**args*, ***kwargs*)

Process Courant Number lines

exec_time_processor(**args*, ***kwargs*)
Process execution/clock time lines

extend_rule(*line_type*, *actions*)
Extend a pre-defined regexp with extra functions

The default action for LogProcessor is to output processed lines into files. Additional actions on pre-defined lines (e.g., "time") can be hooked via this method.

Parameters

- **line_type** (*str*) – Pre-defined line type
- **actions** (*list*) – A list of coroutines that receive the matching lines

residual_processor(**args*, ***kwargs*)
Process a residual line and output data to the relevant file.

time_processor(**args*, ***kwargs*)
Processor for the Time line in log files

watch_file(*target=None*, *wait_time=0.1*)
Process a log file for an in-progress run.

This method takes one parameter, `target`, a coroutine that is called at the end of every timestep. See *LogWatcher* for an example of using target to plot residuals for monitoring the run.

Parameters

- **target** (*coroutine*) – A consumer acting on the data
- **wait_time** (*seconds*) – Wait time between checking the log file for updates

bound_files = None
Open file handles for bounding outputs

case_dir = None
Absolute path to the case directory

converged = None
Flag indicating convergence message in logs

converged_time = None
Timestep when the steady state solver converged

current_state
Return the current state of the logs processor

logfile = None
User-supplied log file (relative to case directory)

logs_dir = None
Absolute path to the directory containing processed logs

res_files = None
Open file handles for the residual outputs

solve_completed = None
Flag indicating solver completion in logs (if End is found)

subiter_map = None
(variable, subIteration) pairs tracking the number of predictor subIterations for each flow variable

time = None
Track the latest time that was processed by the utility

time_str = None
Time as a string (for output)

class caelus.post.logs.**SolverLog**(*case_dir=None*, *logs_dir='logs'*, *force_reload=False*, *logfile=None*)
Bases: object

Caelus solver log file interface.

SolverLog extracts information from solver outputs and allows interaction with the log data as numpy.ndarray or pandas.Dataframe objects.

Parameters

- **case_dir** (*path*) – Absolute path to case directory
- **logs_dir** (*path*) – Path to logs directory relative to case_dir
- **force_reload** (*bool*) – If True, force reread of the log file even if the logs were processed previously.
- **logfile** (*file*) – If force_reload, then log file to process

Raises RuntimeError – An error is raised if no logs directory is available and the user has not provided a logfile that can be processed on the fly during initialization.

bounding_var(*field*)
Return the bounding information for a field

continuity_errors()
Return the time history of continuity errors

residual(*field*, *all_cols=False*)
Return the residual time-history for a field

6.4.2 Caelus Plotting Utilities

This module provides the capability to plot various quantities of interest using matplotlib through *CaelusPlot*.

class caelus.post.plots.**CaelusPlot**(*casedir=None*, *plotdir='results'*)
Bases: object

Caelus Data Plotting Interface

Currently implemented:

- Plot residual time history
- Plot convergence of forces and force coeffcients

Parameters

- **casedir** (*path*) – Path to the case directory
- **plotdir** (*path*) – Directory where figures are saved

plot_force_coeffs_hist(*plotfile=None*, *dpi=300*, ***kwargs*)
Plot force coefficients

Parameters

- **func_object** (*str*) – The function object used in controlDict
- **plotfile** – File to save plot (e.g., residuals.png)

- **dpi** – Resolution for saving plots (default=300)

plot_forces_hist(*plotfile=None*, *dpi=300*, ***kwargs*)
Plot forces

Parameters

- **func_object** (*str*) – The function object used in controlDict
- **plotfile** – File to save plot (e.g., residuals.png)
- **dpi** – Resolution for saving plots (default=300)

plot_residuals_hist(*plotfile=None*, *dpi=300*, ***kwargs*)
Plot time-history of residuals for a Caelus run

Parameters

- **fields** (*list*) – Plot residuals only for the fields in this list
- **plotfile** – File to save plot (e.g., residuals.png)
- **dpi** – Resolution for saving plots (default=300)

casedir = None
Path to the case directory

plot_continuity_errors = None
Flag indicating whether continuity errors are plotted along with residuals

plotdir = None
Path to plots output directory

solver_log = None
Instance of *SolverLog*

class caelus.post.plots.**LogWatcher**(*logfile*, *case_dir=None*)
Bases: object

Real-time log monitoring utility

Parameters

- **logfile** (*str*) – Name of the Caelus log file
- **casedir** (*path*) – Path to the case directory (default: cwd)

continuity_processor(**args*, ***kwargs*)
Capture continuity errors for plot updates

plot_residuals(**args*, ***kwargs*)
Update plot for residuals

residual_processor(**args*, ***kwargs*)
Capture residuals for plot updates

skip_field(*field*)
Helper function to determine if field must be processed

time_processor(**args*, ***kwargs*)
Capture time array

plot_fields = None
List of fields to plot. If None, plots all available fields

skip_fields = None
List of fields to skip. If None, plots all available fields

time_array = None
Time array used for plotting data

class caelus.post.plots.**PlotsMeta**
Bases: type

Provide interactive and non-interactive versions of plot methods.

This metaclass automatically wraps methods starting with _plot such that these methods can be used in both interactive and non-interactive modes. Non-interactive modes are automatically enabled if the user provides a file name to save the resulting figure.

caelus.post.plots.**make_plot_method**(*func*)
Make a wrapper plot method

caelus.post.plots.**mpl_settings**(**args*, ***kwds*)
Temporarily switch matplotlib settings for a plot

6.5 caelus.scripts – CLI App Utilities

6.5.1 Basic CLI Interface

Defines the base classes that are used to build the CLI scripts.

class caelus.scripts.core.**CaelusScriptBase**(*name=None*, *args=None*)
Bases: object

Base class for all Caelus CLI applications.

Defines the common functionality for simple scripts and scripts with sub-commands that are used to access functionality from the library without writing additional python scripts.

Parameters

- **name** (*str*) – Custom name used in messages
- **args** (*str*) – Pass arguments instead of using sys.argv

cli_options()
Setup the command line options and arguments

setup_logging(*log_to_file=True*, *log_file=None*, *verbose_level=0*, *quiet=False*)
Setup logging for the script.

Parameters

- **log_to_file** (*bool*) – If True, script will log to file
- **log_file** (*path*) – Filename to log
- **verbose_level** (*int*) – Level of verbosity

args = None
Arugments provided by user at the command line

description = 'Caelus CLI Application'
Description of the CLI app used in help messages

epilog = 'Caelus Python Library (CPL) v0.1.1'
Epilog for help messages

name = None
Custom name when invoked from a python interface instead of command line

parser = None
Instance of the ArgumentParser used to parse command line arguments

class caelus.scripts.core.**CaelusSubCmdScript**(*name=None*, *args=None*)
Bases: *caelus.scripts.core.CaelusScriptBase*

A CLI app with sub-commands.

Parameters

- **name** (*str*) – Custom name used in messages
- **args** (*str*) – Pass arguments instead of using sys.argv

cli_options()
Setup sub-parsers.

Part III

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

caelus, 31
caelus.config, 31
caelus.config.cmlenv, 33
caelus.config.config, 31
caelus.post, 44
caelus.post.logs, 45
caelus.post.plots, 47
caelus.run.core, 39
caelus.run.hpc_queue, 41
caelus.run.tasks, 38
caelus.scripts.core, 49
caelus.utils, 34
caelus.utils.osutils, 36
caelus.utils.struct, 35

Index

Symbols

–cli-logs log_file
    cpl command line option, 17
–no-log
    cpl command line option, 17
–version
    cpl command line option, 17
-b, –no-backup
    caelus-cfg command line option, 18
-d basedir, –base-dir basedir
    caelus-clone command line option, 19
-d casedir, –case-dir casedir
    caelus-run command line option, 20
-d write_dir, –write_dir write_dir
    caelus-env command line option, 19
-d, case_dir, –case-dir case_dir
    caelus-clean command line option, 22
    caelus-logs command line option, 21
-e exclude_fields, –exclude-patterns exclude fields
    caelus-logs command line option, 21
-e pattern, –exclude-patterns pattern
    caelus_tutorials command line option, 23
-e pattern, –extra-patterns pattern
    caelus-clone command line option, 19
-f output_file, –config-file output_file
    caelus-cfg command line option, 18
-f plot_file, –plot-file plot_file
    caelus-logs command line option, 21
-f task_file, –file task_file
    caelus-tasks command line option, 20
-f task_file, –task-file task_file
    caelus_tutorials command line option, 23
-h, –help
    cpl command line option, 17
-i include_fields, –include-fields include_fields
    caelus-logs command line option, 21
-i pattern, –include-patterns pattern
    caelus_tutorials command line option, 23
-l log_file, –log-file log_file
    caelus-run command line option, 20
-l logs_dir, –logs-dir logs_dir
    caelus-logs command line option, 21
-m, –clean-mesh
    caelus-clean command line option, 22
-m, –skip-mesh
    caelus-clone command line option, 19
-p preserve_pattern, –preserve preserve_pattern
    caelus-clean command line option, 22
-p, –parallel
    caelus-run command line option, 20
-p, –plot-residuals
    caelus-logs command line option, 21
-s, –skip-scripts
    caelus-clone command line option, 19
-v, –verbose
    cpl command line option, 17
-w, –watch
    caelus-logs command line option, 21
-z, –clean-zero
    caelus-clean command line option, 22
-z, –skip-zero
    caelus-clone command line option, 19

A

abspath() (in module caelus.utils.osutils), 36
account (caelus.run.hpc_queue.HPCQueue attribute), 41
add_rule() (caelus.post.logs.LogProcessor method), 45
args (caelus.scripts.core.CaelusScriptBase attribute), 49

B

backup_file() (in module caelus.utils.osutils), 36
bin_dir (caelus.config.cmlenv.CMLEnv attribute), 33
bound_files (caelus.post.logs.LogProcessor attribute), 46
bounding_processor() (caelus.post.logs.LogProcessor method), 45
bounding_var() (caelus.post.logs.SolverLog method), 47
build_dir (caelus.config.cmlenv.CMLEnv attribute), 33

C

caelus
    CPL configuration value, 13
caelus (module), 31
caelus-cfg command line option
    -b, –no-backup, 18
    -f output_file, –config-file output_file, 18
caelus-clean command line option
    -d, case_dir, –case-dir case_dir, 22
    -m, –clean-mesh, 22
    -p preserve_pattern, –preserve preserve_pattern, 22
    -z, –clean-zero, 22
caelus-clone command line option
    -d basedir, –base-dir basedir, 19
    -e pattern, –extra-patterns pattern, 19
    -m, –skip-mesh, 19
    -s, –skip-scripts, 19
    -z, –skip-zero, 19
caelus-env command line option
    -d write_dir, –write_dir write_dir, 19
caelus-logs command line option
    -d, case_dir, –case-dir case_dir, 21
    -e exclude_fields, –exclude-patterns exclude fields, 21
    -f plot_file, –plot-file plot_file, 21
    -i include_fields, –include-fields include_fields, 21
    -l logs_dir, –logs-dir logs_dir, 21
    -p, –plot-residuals, 21
    -w, –watch, 21
caelus-run command line option
    -d casedir, –case-dir casedir, 20
    -l log_file, –log-file log_file, 20
    -p, –parallel, 20
caelus-tasks command line option
    -f task_file, –file task_file, 20
caelus.caelus_cml
    CPL configuration value, 15
caelus.caelus_cml.default
    CPL configuration value, 15
caelus.caelus_cml.versions
    CPL configuration value, 15
caelus.caelus_cml.versions.build_option
    CPL configuration value, 15
caelus.caelus_cml.versions.mpi_bin_path
    CPL configuration value, 16
caelus.caelus_cml.versions.mpi_lib_path
    CPL configuration value, 16
caelus.caelus_cml.versions.mpi_root
    CPL configuration value, 16
caelus.caelus_cml.versions.path
    CPL configuration value, 15
caelus.caelus_cml.versions.version
    CPL configuration value, 15
caelus.config (module), 31
caelus.config.cmlenv (module), 33
caelus.config.config (module), 31
caelus.cpl
    CPL configuration value, 13
caelus.cpl.conda_settings
    CPL configuration value, 14
caelus.cpl.python_env_name
    CPL configuration value, 14
caelus.cpl.python_env_type
    CPL configuration value, 14
caelus.logging
    CPL configuration value, 14
caelus.logging.log_file
    CPL configuration value, 15
caelus.logging.log_to_file
    CPL configuration value, 15
caelus.post (module), 44
caelus.post.logs (module), 45
caelus.post.plots (module), 47
caelus.run.core (module), 39
caelus.run.hpc_queue (module), 41
caelus.run.tasks (module), 38
caelus.scripts.core (module), 49
caelus.system
    CPL configuration value, 14
caelus.system.always_use_scheduler
    CPL configuration value, 14
caelus.system.job_scheduler
    CPL configuration value, 14
caelus.system.scheduler_defaults
    CPL configuration value, 14
caelus.utils (module), 34
caelus.utils.osutils (module), 36
caelus.utils.struct (module), 35
caelus_execute() (in module caelus.run.hpc_queue), 44
CAELUS_PROJECT_DIR, 15
caelus_scripts
    CPL configuration value, 13
caelus_tutorials command line option
    -e pattern, –exclude-patterns pattern, 23
    -f task_file, –task-file task_file, 23
    -i pattern, –include-patterns pattern, 23
caelus_user
    CPL configuration value, 13
CaelusCfg (class in caelus.config.config), 31
CaelusPlot (class in caelus.post.plots), 47
CAELUSRC, 11, 33
CAELUSRC_SYSTEM, 11, 33
CaelusScriptBase (class in caelus.scripts.core), 49
CaelusSubCmdScript (class in caelus.scripts.core), 50
case_dir (caelus.post.logs.LogProcessor attribute), 46
case_dir (caelus.run.tasks.Tasks attribute), 39
casedir (caelus.post.plots.CaelusPlot attribute), 48
clean_case.preserve

CPL task option, 28
clean_case.remove_mesh
CPL task option, 28
clean_case.remove_zero
CPL task option, 27
clean_casedir() (in module caelus.run.core), 39
clean_directory() (in module caelus.utils.osutils), 37
clean_polymesh() (in module caelus.run.core), 39
cli_options() (caelus.scripts.core.CaelusScriptBase method), 49
cli_options() (caelus.scripts.core.CaelusSubCmdScript method), 50
clone_case() (in module caelus.run.core), 40
cmd_clean_case() (caelus.run.tasks.Tasks method), 38
cmd_copy_files() (caelus.run.tasks.Tasks method), 38
cmd_copy_tree() (caelus.run.tasks.Tasks method), 38
cmd_exec_tasks() (caelus.run.tasks.Tasks method), 38
cmd_process_logs() (caelus.run.tasks.Tasks method), 38
cmd_run_command() (caelus.run.tasks.Tasks method), 38
cmd_task_set() (caelus.run.tasks.Tasks method), 38
cml_get_latest_version() (in module caelus.config.cmlenv), 34
cml_get_version() (in module caelus.config.cmlenv), 34
CMLEnv (class in caelus.config.cmlenv), 33
completion_processor() (caelus.post.logs.LogProcessor method), 45
configure_logging() (in module caelus.config.config), 32
continuity_errors() (caelus.post.logs.SolverLog method), 47
continuity_processor() (caelus.post.logs.LogProcessor method), 45
continuity_processor() (caelus.post.plots.LogWatcher method), 48
converged (caelus.post.logs.LogProcessor attribute), 46
converged_time (caelus.post.logs.LogProcessor attribute), 46
convergence_processor() (caelus.post.logs.LogProcessor method), 45
copy_files.dest
CPL task option, 27
copy_files.src
CPL task option, 27
copy_tree() (in module caelus.utils.osutils), 37
copy_tree.dest
CPL task option, 27
copy_tree.ignore_patterns
CPL task option, 27
copy_tree.preserve_symlinks
CPL task option, 27
copy_tree.src
CPL task option, 27
courant_processor() (caelus.post.logs.LogProcessor method), 45
cpl command line option
–cli-logs log_file, 17
–no-log, 17
–version, 17
-h, –help, 17
-v, –verbose, 17
CPL configuration value
caelus, 13
caelus.caelus_cml, 15
caelus.caelus_cml.default, 15
caelus.caelus_cml.versions, 15
caelus.caelus_cml.versions.build_option, 15
caelus.caelus_cml.versions.mpi_bin_path, 16
caelus.caelus_cml.versions.mpi_lib_path, 16
caelus.caelus_cml.versions.mpi_root, 16
caelus.caelus_cml.versions.path, 15
caelus.caelus_cml.versions.version, 15
caelus.cpl, 13
caelus.cpl.conda_settings, 14
caelus.cpl.python_env_name, 14
caelus.cpl.python_env_type, 14
caelus.logging, 14
caelus.logging.log_file, 15
caelus.logging.log_to_file, 15
caelus.system, 14
caelus.system.always_use_scheduler, 14
caelus.system.job_scheduler, 14
caelus.system.scheduler_defaults, 14
caelus_scripts, 13
caelus_user, 13
CPL task option
clean_case.preserve, 28
clean_case.remove_mesh, 28
clean_case.remove_zero, 27
copy_files.dest, 27
copy_files.src, 27
copy_tree.dest, 27
copy_tree.ignore_patterns, 27
copy_tree.preserve_symlinks, 27
copy_tree.src, 27
process_logs.log_file, 28
process_logs.logs_directory, 28
process_logs.plot_continuity_errors, 28
process_logs.plot_residuals, 28
process_logs.residual_fields, 28
process_logs.residuals_plot_file, 28
run_command.cmd_args, 26
run_command.cmd_name, 26
run_command.log_file, 26
run_command.mpi_extra_args, 27
run_command.num_ranks, 26
run_command.parallel, 26
task_set.case_dir, 28
task_set.name, 28

task_set.tasks, 28
current_state (caelus.post.logs.LogProcessor attribute), 46

D

delete() (caelus.run.hpc_queue.HPCQueue static method), 41
delete() (caelus.run.hpc_queue.PBSQueue static method), 42
delete() (caelus.run.hpc_queue.SerialJob static method), 43
delete() (caelus.run.hpc_queue.SlurmQueue static method), 43
description (caelus.scripts.core.CaelusScriptBase attribute), 49
discover_versions() (in module caelus.config.cmlenv), 34

E

email_address (caelus.run.hpc_queue.HPCQueue attribute), 41
ensure_directory() (in module caelus.utils.osutils), 37
env (caelus.run.tasks.Tasks attribute), 39
environ (caelus.config.cmlenv.CMLEnv attribute), 34
environment variable
CAELUS_PROJECT_DIR, 15
CAELUSRC, 11, 33
CAELUSRC_SYSTEM, 11, 33
epilog (caelus.scripts.core.CaelusScriptBase attribute), 49
exec_time_processor() (caelus.post.logs.LogProcessor method), 45
extend_rule() (caelus.post.logs.LogProcessor method), 46

F

find_caelus_recipe_dirs() (in module caelus.run.core), 40
find_case_dirs() (in module caelus.run.core), 40
find_recipe_dirs() (in module caelus.run.core), 40
from_yaml() (caelus.utils.struct.Struct class method), 35

G

gen_yaml_decoder() (in module caelus.utils.struct), 36
gen_yaml_encoder() (in module caelus.utils.struct), 36
get_appdata_dir() (in module caelus.config.config), 32
get_caelus_root() (in module caelus.config.config), 32
get_config() (in module caelus.config.config), 32
get_cpl_root() (in module caelus.config.config), 32
get_default_config() (in module caelus.config.config), 32
get_job_scheduler() (in module caelus.run.hpc_queue), 44
get_mpi_size() (in module caelus.run.core), 40
get_queue_settings() (caelus.run.hpc_queue.HPCQueue method), 42
get_queue_settings() (caelus.run.hpc_queue.PBSQueue method), 42
get_queue_settings() (caelus.run.hpc_queue.SerialJob method), 43
get_queue_settings() (caelus.run.hpc_queue.SlurmQueue method), 43

H

HPCQueue (class in caelus.run.hpc_queue), 41

I

is_caelus_casedir() (in module caelus.run.core), 40
is_job_scheduler() (caelus.run.hpc_queue.HPCQueue static method), 42
is_job_scheduler() (caelus.run.hpc_queue.SerialJob static method), 43
is_parallel() (caelus.run.hpc_queue.HPCQueue static method), 42
is_parallel() (caelus.run.hpc_queue.ParallelJob static method), 43
is_parallel() (caelus.run.hpc_queue.SerialJob static method), 43

J

join_outputs (caelus.run.hpc_queue.HPCQueue attribute), 41

L

lib_dir (caelus.config.cmlenv.CMLEnv attribute), 34
load() (caelus.run.tasks.Tasks class method), 39
load_yaml() (caelus.utils.struct.Struct class method), 35
logfile (caelus.post.logs.LogProcessor attribute), 46
LogProcessor (class in caelus.post.logs), 45
logs_dir (caelus.post.logs.LogProcessor attribute), 46
LogWatcher (class in caelus.post.plots), 48

M

mail_opts (caelus.run.hpc_queue.HPCQueue attribute), 41
make_plot_method() (in module caelus.post.plots), 49
merge() (caelus.utils.struct.Struct method), 35
merge() (in module caelus.utils.struct), 36
mpi_bindir (caelus.config.cmlenv.CMLEnv attribute), 34
mpi_dir (caelus.config.cmlenv.CMLEnv attribute), 34
mpi_extra_args (caelus.run.hpc_queue.HPCQueue attribute), 41
mpi_libdir (caelus.config.cmlenv.CMLEnv attribute), 34
mpl_settings() (in module caelus.post.plots), 49

N

name (caelus.run.hpc_queue.HPCQueue attribute), 41
name (caelus.scripts.core.CaelusScriptBase attribute), 49
num_nodes (caelus.run.hpc_queue.HPCQueue attribute), 41

num_ranks (caelus.run.hpc_queue.HPCQueue attribute), 41

O

ostype() (in module caelus.utils.osutils), 37

P

ParallelJob (class in caelus.run.hpc_queue), 43
parser (caelus.scripts.core.CaelusScriptBase attribute), 50
PBSQueue (class in caelus.run.hpc_queue), 42
plot_continuity_errors (caelus.post.plots.CaelusPlot attribute), 48
plot_fields (caelus.post.plots.LogWatcher attribute), 48
plot_force_coeffs_hist() (caelus.post.plots.CaelusPlot method), 47
plot_forces_hist() (caelus.post.plots.CaelusPlot method), 48
plot_residuals() (caelus.post.plots.LogWatcher method), 48
plot_residuals_hist() (caelus.post.plots.CaelusPlot method), 48
plotdir (caelus.post.plots.CaelusPlot attribute), 48
PlotsMeta (class in caelus.post.plots), 49
prepare_mpi_cmd() (caelus.run.hpc_queue.HPCQueue method), 42
prepare_mpi_cmd() (caelus.run.hpc_queue.ParallelJob method), 43
prepare_mpi_cmd() (caelus.run.hpc_queue.SerialJob method), 43
prepare_srun_cmd() (caelus.run.hpc_queue.SlurmQueue method), 43
process_logs.log_file
    CPL task option, 28
process_logs.logs_directory
    CPL task option, 28
process_logs.plot_continuity_errors
    CPL task option, 28
process_logs.plot_residuals
    CPL task option, 28
process_logs.residual_fields
    CPL task option, 28
process_logs.residuals_plot_file
    CPL task option, 28
process_run_env() (caelus.run.hpc_queue.HPCQueue method), 42
project_dir (caelus.config.cmlenv.CMLEnv attribute), 34

Q

qos (caelus.run.hpc_queue.HPCQueue attribute), 41
queue (caelus.run.hpc_queue.HPCQueue attribute), 41
queue_name (caelus.run.hpc_queue.HPCQueue attribute), 42

R

rcfiles_loaded() (in module caelus.config.config), 32
reload_config() (in module caelus.config.config), 33
remove_files_dirs() (in module caelus.utils.osutils), 37
res_files (caelus.post.logs.LogProcessor attribute), 46
reset_default_config() (in module caelus.config.config), 33
residual() (caelus.post.logs.SolverLog method), 47
residual_processor() (caelus.post.logs.LogProcessor method), 46
residual_processor() (caelus.post.plots.LogWatcher method), 48
root (caelus.config.cmlenv.CMLEnv attribute), 34
run_command.cmd_args
    CPL task option, 26
run_command.cmd_name
    CPL task option, 26
run_command.log_file
    CPL task option, 26
run_command.mpi_extra_args
    CPL task option, 27
run_command.num_ranks
    CPL task option, 26
run_command.parallel
    CPL task option, 26

S

script_body (caelus.run.hpc_queue.HPCQueue attribute), 42
search_cfg_files() (in module caelus.config.config), 33
SerialJob (class in caelus.run.hpc_queue), 43
set_work_dir() (in module caelus.utils.osutils), 37
setup_logging() (caelus.scripts.core.CaelusScriptBase method), 49
shell (caelus.run.hpc_queue.HPCQueue attribute), 41
skip_field() (caelus.post.plots.LogWatcher method), 48
skip_fields (caelus.post.plots.LogWatcher attribute), 48
SlurmQueue (class in caelus.run.hpc_queue), 43
solve_completed (caelus.post.logs.LogProcessor attribute), 46
solver_log (caelus.post.plots.CaelusPlot attribute), 48
SolverLog (class in caelus.post.logs), 47
stderr (caelus.run.hpc_queue.HPCQueue attribute), 41
stdout (caelus.run.hpc_queue.HPCQueue attribute), 41
Struct (class in caelus.utils.struct), 35
StructMeta (class in caelus.utils.struct), 35
subiter_map (caelus.post.logs.LogProcessor attribute), 46
submit() (caelus.run.hpc_queue.HPCQueue class method), 42
submit() (caelus.run.hpc_queue.PBSQueue class method), 42
submit() (caelus.run.hpc_queue.SerialJob class method), 43

submit() (caelus.run.hpc_queue.SlurmQueue class method), 43

T

task_file (caelus.run.tasks.Tasks attribute), 39
task_set.case_dir
    CPL task option, 28
task_set.name
    CPL task option, 28
task_set.tasks
    CPL task option, 28
tasks (caelus.run.tasks.Tasks attribute), 39
Tasks (class in caelus.run.tasks), 38
TasksMeta (class in caelus.run.tasks), 39
time (caelus.post.logs.LogProcessor attribute), 46
time_array (caelus.post.plots.LogWatcher attribute), 48
time_limit (caelus.run.hpc_queue.HPCQueue attribute), 41
time_processor() (caelus.post.logs.LogProcessor method), 46
time_processor() (caelus.post.plots.LogWatcher method), 48
time_str (caelus.post.logs.LogProcessor attribute), 46
timestamp() (in module caelus.utils.osutils), 38
to_yaml() (caelus.utils.struct.Struct method), 35

U

update() (caelus.run.hpc_queue.HPCQueue method), 42
user_home_dir() (in module caelus.utils.osutils), 38
username() (in module caelus.utils.osutils), 38

V

version (caelus.config.cmlenv.CMLEnv attribute), 34

W

watch_file() (caelus.post.logs.LogProcessor method), 46
write_config() (caelus.config.config.CaelusCfg method), 32
write_script() (caelus.run.hpc_queue.HPCQueue method), 42

Y

yaml_decoder (caelus.config.config.CaelusCfg attribute), 32
yaml_decoder (caelus.utils.struct.Struct attribute), 35
yaml_encoder (caelus.config.config.CaelusCfg attribute), 32
yaml_encoder (caelus.utils.struct.Struct attribute), 35