Caelus Python Library (CPL)

Release v1.0.1

Applied CCM

Apr 13, 2019

Contents

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:

1. 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.
2. 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.
3. 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
```

(continues on next page)

(continued from previous page)

```yaml
  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:

```yaml
# 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
```

(continues on next page)

(continued from previous page)

```
#
# 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
`machinefile`	File used in `mpirun -machinefile <FILE>`

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.

--cml-version
: Specify CML version to use for this particular invocation of CPL command.

--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
`clone`	Clone a case directory
`tasks`	Automatic execution of workflow from a YAML file
`run`	Run a CML executable in the appropriate environment
`runpy`	Run a python script in the appropriate environment
`logs`	Parse a solver log file and extract data for analysis
`clean`	Clean a case directory after execution
`build`	Compile CML sources

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 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.

4.2.1.3 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

4.2.1.4 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"`. This ensures that the arguments meant for the CML executable are not parsed as arguments to the `caelus` executable during the run.

```
$ caelus run -h
usage: caelus run [-h] [-p] [-l LOG_FILE] [-d CASE_DIR] [-m MACHINEFILE]
                  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
  -m MACHINEFILE, --machinefile MACHINEFILE
                        machine file for distributed runs (local_mpi only)
```

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

-m, --machinefile
: File containing nodes used for a distributed MPI run. This option is ignored if `job_scheduler` is not `local_mpi`. This option has no effect if the *parallel option* is not used.

-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.

4.2.1.5 caelus runpy – Run a custom python script

Runs a user-provided python script in the case directory. CPL ensures that the correct version of CML and python environment are setup prior to the invocation of the python script. Like **caelus run**, it is recommended that the arguments meant for the user script be separated from **caelus runpy** arguments with --.

```
$ caelus runpy -h
usage: caelus runpy [-h] [-l LOG_FILE] [-d CASE_DIR]
                    script [script_args [script_args ...]]

Run a custom python script with CML and CPL environment

positional arguments:
  script                path to the python script
  script_args           additional arguments passed to command

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

-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
```

(continues on next page)

(continued from previous page)

```
-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 an ongoing run. To exit before the completion of the run, hit `Ctrl+C`.

-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] [-t] [-P] [-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
```

(continues on next page)

(continued from previous page)

```
  -m, --clean-mesh      remove polyMesh directory (default: no)
  -z, --clean-zero      remove 0 directory (default: no)
  -t, --clean-time-dirs
                        remove time directories (default: no)
  -P, --clean-processors
                        clean processor directories (default: no)
  -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.

-t, --clean-time-dirs
Remove time directories from the case directory. Note, this only removes the reconstructed time directories and not the decomposed directores that exist within `processor*` directories.

-P, --clean-processors
Remove decomposed `processor*` directories from the case directory.

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

4.2.1.8 caelus build – Compile CML sources

`caelus build` is a wrapper to SCons shipped with CML sources that can be used to build executables in both CML project and user directories. The command can be executed from any directory when building project or user directories. It determines the actual paths to the project and user directories based on the *user configuration* files, and the SCons configuration within those projects. The user can override the default project and user directories by specifying the *`--cml-version`* flag when invoking this command.

Warning: When using CPL with Python 3.x versions, you will need a recent version of CML to invoke `caelus build`. This is because the SCons versions shipped with CML versions `v8.04` and older can only run on Python 2.x.

```
$ caelus build -h
usage: caelus build [-h] [-l LOG_FILE] [-c] [-j JOBS]
                    [-a | -p | -u | -d SOURCE_DIR]
                    [scons_args [scons_args ...]]

Compile Caelus CML

positional arguments:
  scons_args            additional arguments passed to SCons

optional arguments:
```

(continues on next page)

(continued from previous page)

```
  -h, --help            show this help message and exit
  -l LOG_FILE, --log-file LOG_FILE
                       filename to redirect build output
  -c, --clean           clean CML build
  -j JOBS, --jobs JOBS  number of parallel jobs
  -a, --all             Build both project and user directories (default: no)
  -p, --project         Build Caelus CML project (default: no)
  -u, --user            Build user project (default: no)
  -d SOURCE_DIR, --source-dir SOURCE_DIR
                        Build sources in path (default: CWD)
```

The positional arguments are passed directly to SCons providing user with full control over how the SCons build must be handled. It is recommended that the user separate the *optional arguments* to `caelus build` command from the arguments that must be passed to SCons using double dashes (`--`).

`-d, --source-dir`
Build sources in the current working directory. This is the default option. If the user is in the top-level directory containing the `SConstruct` file, then it builds the entire project. If the user is in a sub-directory containing a `SConscript` file, then it just builds the libraries and executables defined in that directory and sub-directories. An example would be to recompile just the turbulence model libraries during development phase.

`-p, --project`
Build the sources in project directory only.

`-u, --user`
Build the sources in user directory only.

`-a, --all`
Build both the project and the user directories. The command will abort compilation if the compilation of the project files fail and will not attempt to build the sources in user directory.

`-j, --jobs`
The number of concurrent compilation jobs that must be launched with SCons. The default value is determined by the number of CPU cores available on the user's system.

`-c, --clean`
Instead of recompiling the sources, execute the `clean` command through SCons.

4.2.2 `caelus_sim` – Parametric Run CLI

The `caelus_sim` is a shell executable that provides a command-line interface to setup and execute a parametric analysis. Currently, the following sub-commands are available through **`caelus_sim`** executable. Please see *Parametric runs using CPL* tutorial for a detailed description of usage.

Action	Purpose
`setup`	Setup a new parametric run
`prep`	Execute pre-processing actions
`solve`	Run the solver
`post`	Execute post-processing actions
`status`	Print out status of the analysis

Note: The script also supports *`common options`* documented previously. Care must be taken to include the common options before the subcommand, i.e.,

```
# Correct usage
caelus_sim -v setup

# Incorrect usage, will generate an error
caelus_sim setup -v
```

4.2.2.1 caelus_sim setup – Setup a parametric run

By default, this command will parse the `caelus_sim.yaml` input file and setup a new analysis under a new directory `name` provided either at the command line or in the input file. The individual cases corresponding to the run matrix appear as subdirectories to the top-level analysis directory.

```
$ caelus_sim setup -h
usage: caelus_sim setup [-h] [-n SIM_NAME] [-d BASE_DIR] [-s] [-p]
                        [-f SIM_CONFIG]

setup a parametric run

optional arguments:
  -h, --help            show this help message and exit
  -n SIM_NAME, --sim-name SIM_NAME
                        name of this simulation group
  -d BASE_DIR, --base-dir BASE_DIR
                        base directory where the simulation structure is
                        created
  -s, --submit          submit solve jobs on successful setup
  -p, --prep            run pre-processing steps after successful setup
  -f SIM_CONFIG, --sim-config SIM_CONFIG
                        YAML-formatted simulation configuration
                        (caelus_sim.yaml)
```

-f, --sim-config
The input file containing the details of the analysis to be performed. Default value is `caelus_sim.yaml`

-n, --sim-name
Name of this parametric run. This option overrides the `sim_name` entry in the input file.

-d, --base-dir
Directory where the parametric analysis is setup. This directory must exist. Default value is the current working directory.

-s, --submit
Submit the solve jobs after setup is complete.

-p, --prep
Run pre-processing tasks upon successful setup.

4.2.2.2 caelus_sim status – Print status of the parametric runs

This command prints out the status of the runs so far. The meanings of the different status types are described in the table below

Status	Description
Setup	Case setup successfully
Prepped	Pre-processing completed
Submitted	Solver initialized
Running	Solver is running
Solved	Solve has completed
DONE	Post-processing completed
FAILED	Some action failed

4.2.3 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
```

(continues on next page)

(continued from previous page)

```
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.

```
- run_command:
    cmd_name: pimpleSolver
    parallel: true
    mpi_extra_args: "--hostfile my_hostfile"
```

5.2.2 run_python – Run user python scripts

This task will run a python script within the case directory. Like `run_command` task, CPL will ensure that the apporpriate CML version as well as python environment is setup correctly prior to invoking the script. The task must provide one mandatory parameter *run_python.script* that contains the path to the python script to be executed. User can pass additional options to this task as documented below.

```
- run_python:
    script: "../mytestscript.py"
    script_args: "-v -f option1 arg1 arg2"
    log_file: "mycustom.log"
```

run_python.script
Path to the custom python script.

run_python.script_args
Arguments that must be passed to the python script during invocation. As with `run_command` quote the arguments appropriately to ensure that it is passed correctly to the script.

run_python.log_file
The filename where the outputs and error messages from the script is redirected.

5.2.3 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.

5.2.4 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.

5.2.5 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 `no`.

clean_case.remove_mesh
Boolean flag indicating whether the `constant/polyMesh` directory should be removed. The default value is `no`.

clean_case.remove_time_dirs
Boolean flag indicating whether time directories from a previous run should be removed. The default value is `no`.

clean_case.remove_processors
Boolean flag indicating whether processor directories from a previous run should be removed. The default value is `no`.

clean_case.purge_generated
A Boolean flag when enabled will set `remove_time_dirs` and `remove_processors` to `yes`.

clean_case.purge_all
A shortcut to set `remove_zero`, `remove_mesh`, `remove_time_dirs` and `remove_processors` to `yes`.

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

5.2.6 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.

5.2.7 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.

CHAPTER 6

Tutorials

6.1 Parametric runs using CPL

CPL provides two classes *CMLSimulation* and *CMLSimCollection* that can be used to create workflows that can automate the CFD analysis process. It also provides an implementation of `CMLSimCollection` called *CMLParametricRun* that can be used to automate running a parametric study over several variables and managing the analysis as a group. These classes serve as simple examples for the user to derive sub-classes from CPL to develop their own custom workflows.

The parametric run capability is also accessible from the command-line via **`caelus_sim`**. This tutorial provides a step-by-step walkthrough of exercising CPL's parametric run capabilities through the command-line. This tutorial will demonstrate an example of generating airfoil polars for a range of angles of attack at different Reynolds numbers. In addition to varying, the angle of attack and Reynolds number, it will also show how to specify other flow parameters through CPL.

6.1.1 Preliminaries

To use CPL's parametric run interface, the user needs to provide a simulation configuration file (in YAML format), and a case template directory (similar to the one used with **`caelus clone`** command).

To follow along with this tutorial, we recommend that the user download the `parametric run setup file` and a `case template`. For the purposes of this tutorial we will assume that the user is executing the commands from within `$HOME/run` directory. Once downloaded please unzip the zip file.

```
# Files downloaded for the tutorial walkthrough
bash:/tmp/run$ ls
airfoil_demo.zip caelus_sim.yaml

# Unzip the file
bash:/tmp/run$ unzip airfoil_demo.zip
Archive:  airfoil_demo.zip
   creating: airfoil_template/
```

(continues on next page)

(continued from previous page)

```
   creating: airfoil_template/0.orig/
  inflating: airfoil_template/0.orig/k
  inflating: airfoil_template/0.orig/nut
  inflating: airfoil_template/0.orig/omega
  inflating: airfoil_template/0.orig/p
  inflating: airfoil_template/0.orig/U
  inflating: airfoil_template/cmlControls
   creating: airfoil_template/constant/
   creating: airfoil_template/constant/polyMesh/
  inflating: airfoil_template/constant/polyMesh/boundary
  inflating: airfoil_template/constant/polyMesh/faces.gz
  inflating: airfoil_template/constant/polyMesh/neighbour.gz
  inflating: airfoil_template/constant/polyMesh/owner.gz
  inflating: airfoil_template/constant/polyMesh/points.gz
  inflating: airfoil_template/constant/RASProperties
  inflating: airfoil_template/constant/transportProperties
  inflating: airfoil_template/constant/turbulenceProperties
   creating: airfoil_template/system/
  inflating: airfoil_template/system/controlDict
  inflating: airfoil_template/system/decomposeParDict
  inflating: airfoil_template/system/fvSchemes
  inflating: airfoil_template/system/fvSolution
```

6.1.1.1 Preparing a case template directory

In order to simplify the process of setting up parametric run, CPL assumes that all user-configurable entries are provided in `cmlControls` within the case directory. Other files within the case directory use the `#include` option to include this file and use variable replacement macro syntax `$variable` to interface with the parametric run utility. In this airfoil demonstration example, a `cmlControls` that will be used is shown below:

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      "cmlControls";
}

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

density         1.225;

Uinf            15.0;

chord           1.0;

Re              1000000.0;

aoa             0.0;

turbKe          3.75e-07;

velVector       (15.0 0.0 0.0);

nuValue         1.0e7;
```

(continues on next page)

(continued from previous page)

```
26
27 turbulenceModel kOmegaSST;
28
29 // ************************************************************************* //
```

An example of using this file to set the turbulence model in `constant/RASProperties` is shown below:

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      RASProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

#include "../cmlControls"

RASModel        $turbulenceModel;

turbulence      on;

printCoeffs     on;

kMin            1.e-20;

// ************************************************************************* //
```

Notice how the `cmlControls` file is included in line 11, and the property `RASModel` is set with `$turbulenceModel` (see line 27 in `cmlControls` snippet above). The user is referred to `constant/transportProperties`, `0.orig/U`, `0.orig/k`, and `functions` section of `system/controlDict` for further examples of how the inputs from `cmlControls` is used to customize the case.

Tip: Currently, `cmlControls` and CPL's dictionary manipulation capabilities are restricted to text files only. If you want to customize binary files within `0/` directory, then we recommend using `cmlControls` to modify `system/changeDictionaryDict` and execute CML's **`changeDictionary`** executable in the pre-processing phase to modify binary files.

6.1.1.2 Inputs for setting up parametric run

The first step to creating a parametric analysis directory structure is to execute the **`caelus_sim setup`** command. By default, this command will attempt to load the analysis configuration from the `caelus_sim.yaml`. The user can, however, change this by providing an alternate file with the `-f` flag. The contents of the `caelus_sim.yaml` used for this demo is shown below.

```
1 # -*- mode: yaml -*-
2
3 # caelus_sim requires a simulation section in YAML file
4 simulation:
5
6   # Name of the parametric run top-level directory. Can also set using -n flag
7   # at command line which takes precedence
```

(continues on next page)

(continued from previous page)

```yaml
sim_name: airfoil_demo

# The template directory for creating case directories. Requires 'path'
# argument. Other optional variables can be provided to control clone behavior
template:
  path: "./airfoil_template"
  # copy_polymesh: no
  # copy_zero: no
  # copy_scripts: no
  # extra_patterns: ["*.py"]

# This section details the parametric run options
simulation_setup:
  # User defined format for case directory names
  case_format: "Re_{Re:.1e}/aoa_{aoa:+06.2f}"

  # The matrix of runs
  run_matrix:
    - Re: [1.0e6, 2.0e6]
      aoa:
        start: 0.0
        stop: 2.0
        step: 2.0

    # Only run one Re for the negative AOA
    - Re: [1.0e6]
      aoa: [-4.0, -2.0]

  # Other parameters that are passed to cmlControls file
  constant_parameters:
    density: 1.225
    Uinf: 15.0
    chord: 1.0
    turbKe: 3.75e-07
    turbulenceModel: kOmegaSST

  # User-defined transformations on the variables
  apply_transforms:
    transform_type: code
    # Only pass these variables to cmlControls
    extract_vars: [velVector, Re, nuValue, liftVector, dragVector]
    # Python code that is executed before generating case parameters
    code: |
      Re = float(Re)
      aoaRad = np.radians(aoa)
      Ux = Uinf * np.cos(aoaRad)
      Uy = Uinf * np.sin(aoaRad)
      velVector = np.array([Ux, Uy, 0.0])
      nuValue = Uinf / Re
      liftVector = np.array([-np.sin(aoaRad), np.cos(aoaRad), 0.0])
      dragVector = np.array([np.cos(aoaRad), np.sin(aoaRad), 0.0])

# Configuration for running each case within this analysis group
run_configuration:
  # Number of MPI ranks for parallel runs
  num_ranks: 2
  # Extra MPI arguments passed during parallel runs
```

(continues on next page)

(continued from previous page)

```
    # mpi_extra_args: -machinefile mymachines

    # Should the case be reconstructed on successful run
    reconstruct: no

    # Modify the default template input files
    change_inputs:
      controlDict:
        endTime: 1
        writeFormat: binary

    # Pre-processing actions to be performed before solve. The syntax is similar
    # to the Tasks YAML interface. Note, the case manager will automatically
    # perform decomposition and similar tasks. Users only need to provide
    # non-standard pre-processing actions (e.g., copy actions, changeDictionary,
    # or mesh generation) during the prep phase.
    prep:
      - copy_tree:
          src: "0.orig"
          dest: "0"

    # Perform the solve
    solve: simpleSolver

    # solve:
    #   - solver: potentialSolver
    #     solver_args: "-initializeUBCs -noFunctionObjects"

    #   - solver: pimpleSolver

    # Similar to prep, perform post-processing actions
    # post:
    #   - run_command:
    #       cmd_name: "python ./my_post_process.py"
```

The input file must contain one section `simulation` that provides all the information necessary for setting up and executing the parametric run. The `simulation` dictionary contains the following major sections:

`sim_name`

> The name of the parametric analysis group. The program creates a unique run directory with this name. This parameter can also be overridden from the command line.

`template`

> Details of the case template that will be used to create the individual case directories. It must contain one mandatory entry: `path`, that is the path to the template case directory. In this demo we will use the `airfoil_template` that we unzipped from the `airfoil_demo.zip`. Additional parameters are passed to control the cloning of the template directory and are similar to **`caelus clone`** command.

`simulation_setup`

> This section contains the details of the parametric run. `case_template` is a template suitable to be processed by python str.format() method.
>
> `run_matrix` contains the list of parametric combinations that will be run. In this example, we will run two angles of attack for each of the two Reynolds numbers specified. Each entry in the list generates all possible combinations of runs possible, and these are sub-groups of parametric runs. User can provide multiple entries in the list to generate additional custom combinations.

`constant_parameters`, if present, are variables that that will be populated in addition to the variable parameters (in `run_matrix`) when setting up the case.

Finally, `apply_transforms` is an optional section, that uses valid python code snippets to perform user-defined transformations to the variables in `run_matrix` and `constant_parameters` to generate dependent variables or perform additional processing. After transformation, by default, all variables introduced by the python code is extracted and passed along with `constant_parameters` and variables in `run_matrix` to `cmlControls`. However, if the user has imported modules or defined functions, this could lead to error. So it is recommended that the user manually specify the variables to extracted through the `extract_vars` option in `apply_transforms` section.

`run_configuration`

This section contains the details on how each case is executed after setup. It has a general section that has information regarding parallel run setup etc.

`change_inputs` lists changes to be performed to input files cloned from the template directory. This step is performed after setting up case directory, but before execution of any pre-processing or solve tasks.

`prep` contains tasks (see *Caelus Tasks*) that must be executed before decomposing and executing the case. Note that the user need not specify the `decomposePar` task, as this is handled automatically by the parametric run interface.

`solve` indicates the name of the solver that is used to run these cases. It can also accept a list entry with multiple solvers (e.g., running `potentialSolver` before `simpleSolver`, etc.)

`post` is similar to `prep` but are tasks that are executed after a successful solver completion.

6.1.2 Setting up a parametric run

Now that we have the requisite inputs for setting up a parametric run, we will use **`caelus_sim setup`** command to setup a case (see *caelus_sim – Parametric Run CLI* for more details).

```
bash:/tmp/run$ caelus_sim setup
INFO: Caelus Python Library (CPL) v0.1.1
#
# Output deleted
#
INFO: Successfully setup simulation: airfoil_demo (6)
```

On successful execution, you should see a new directory `airfoil_demo` that contains six Caelus case directories for the various combinations of Reynolds number and angles of attack. User can query the status of the analysis by executing the `status` sub-command.

```
bash:/tmp/run/airfoil_demo$ ls
Re_1.0e+06      Re_2.0e+06      caelus_sim.yaml
bash:/tmp/run/airfoil_demo$ caelus_sim status
INFO: Caelus Python Library (CPL) v0.1.1

Run status for: airfoil_demo
Directory: /private/tmp/run/airfoil_demo
================================================
#. NAME                       STATUS
================================================
1. Re_1.0e+06/aoa_+00.00     Setup
2. Re_1.0e+06/aoa_+02.00     Setup
3. Re_2.0e+06/aoa_+00.00     Setup
4. Re_2.0e+06/aoa_+02.00     Setup
```

(continues on next page)

(continued from previous page)

```
5. Re_1.0e+06/aoa_-04.00    Setup
6. Re_1.0e+06/aoa_-02.00    Setup
================================================
TOTAL = 6; SUCCESS = 0; FAILED = 0
================================================
```

For a description of the various status tags, please consult *caelus_sim status* documentation.

Note:

1. If you are running from a directory outside of `airfoil_demo` then provide the case path using the `-d` option to **`caelus_sim status -d`**. You don't need to provide this if you were, say, within `airfoil_demo/Re_1.0e+06` directory or any of the subdirectories.
2. Setup sub-command allows the user to immediately perform `prep` tasks or submit the `solve` jobs immediately upon setup using the `-p` or `-s` flags during the invocation of `setup`.

6.1.3 Prep, solve, and post

Now that the cases as setup, the user can examine the auto-generated case directories to ensure everything is setup properly and can *run* the simulation by just invoking the `solve` sub-command. CPL will detect if pre-processing and case decomposition haven't been performed and will perform these tasks. User also has the option to explicitly invoke the `prep` task separately from command line. Without any arguments, these sub-commands will choose all the cases within a parametric run for execution. User can, however, pass shell-style wildcard arguments to select a subset of cases where the command is executed. In this tutorial, we will demonstrate this behavior by executing `prep` only on the cases where $Re = 2 \times 10^6$.

```
#
# Submit prep only for Re=2e6
#
bash:/tmp/run/airfoil_demo$ caelus_sim prep 'Re_2.0*/*'
INFO: Caelus Python Library (CPL) v0.1.144-g41f57bc-dirty
INFO: Executing pre-processing tasks for case: Re_2.0e+06/aoa_+00.00
INFO: Writing Caelus input file: system/decomposeParDict
INFO: Decomposing case: Re_2.0e+06/aoa_+00.00
INFO: Executing pre-processing tasks for case: Re_2.0e+06/aoa_+02.00
INFO: Writing Caelus input file: system/decomposeParDict
INFO: Decomposing case: Re_2.0e+06/aoa_+02.00

#
# Check status of simulation
#
bash:/tmp/run/airfoil_demo$ caelus_sim status
INFO: Caelus Python Library (CPL) v0.1.1-44-g41f57bc-dirty

Run status for: airfoil_demo
Directory: /private/tmp/run/airfoil_demo
================================================
#. NAME                        STATUS
================================================
1. Re_1.0e+06/aoa_+00.00    Setup
2. Re_1.0e+06/aoa_+02.00    Setup
3. Re_2.0e+06/aoa_+00.00    Prepped
```

(continues on next page)

(continued from previous page)

```
4. Re_2.0e+06/aoa_+02.00    Prepped
5. Re_1.0e+06/aoa_-04.00    Setup
6. Re_1.0e+06/aoa_-02.00    Setup
================================================
TOTAL = 6; SUCCESS = 0; FAILED = 0
================================================
```

Note the use of single quotes around the wildcard arguments to prevent expansion by the shell when parsing the command line.

In the next step, we will directly invoke `solve` on the positive angle of attack cases for $Re = 1 \times 10^6$ to demonstrate the automatic invocation of `prep` if not already performed.

```
#
# Submit solve only for positive aoa and Re=1e6
#
bash:/tmp/run/airfoil_demo$ caelus_sim solve 'Re_1.0*/aoa_+*'
INFO: Caelus Python Library (CPL) v0.1.1
INFO: Executing pre-processing tasks for case: Re_1.0e+06/aoa_+00.00
INFO: Submitting solver (simpleSolver) for case: Re_1.0e+06/aoa_+00.00
INFO: Executing pre-processing tasks for case: Re_1.0e+06/aoa_+02.00
INFO: Submitting solver (simpleSolver) for case: Re_1.0e+06/aoa_+02.00

#
# Check status of simulation
#
bash:/tmp/run/airfoil_demo$ caelus_sim status
INFO: Caelus Python Library (CPL) v0.1.1

Run status for: airfoil_demo
Directory: /private/tmp/run/airfoil_demo
================================================
#. NAME                       STATUS
================================================
1. Re_1.0e+06/aoa_+00.00     Solved
2. Re_1.0e+06/aoa_+02.00     FAILED
3. Re_2.0e+06/aoa_+00.00     Prepped
4. Re_2.0e+06/aoa_+02.00     Prepped
5. Re_1.0e+06/aoa_-04.00     Setup
6. Re_1.0e+06/aoa_-02.00     Setup
================================================
TOTAL = 6; SUCCESS = 0; FAILED = 1
================================================
```

Note: For the purposes of demonstration, the `endTime` is `controlDict` is set to one timestep. Also a deliberate error was introduced in solve step to demonstrate the `FAILED` status flag.

User can execute the `post` step and it will only execute post-processing actions on cases that have completed the solve.

```
#
# Execute post-processing actions
#
bash:/tmp/run/airfoil_demo$ caelus_sim post
INFO: Caelus Python Library (CPL) v0.1.1
```

(continues on next page)

(continued from previous page)

```
INFO: Executing post-processing tasks for case: Re_1.0e+06/aoa_+00.00
WARNING: Re_1.0e+06/aoa_+02.00: No previous solve detected, skipping post
WARNING: Re_2.0e+06/aoa_+00.00: No previous solve detected, skipping post
WARNING: Re_2.0e+06/aoa_+02.00: No previous solve detected, skipping post
WARNING: Re_1.0e+06/aoa_-04.00: No previous solve detected, skipping post
WARNING: Re_1.0e+06/aoa_-02.00: No previous solve detected, skipping post

#
# Check status of simulation
#
bash:/tmp/run/airfoil_demo$ caelus_sim status
INFO: Caelus Python Library (CPL) v0.1.1

Run status for: airfoil_demo
Directory: /private/tmp/run/airfoil_demo
================================================
#. NAME                       STATUS
================================================
1. Re_1.0e+06/aoa_+00.00     DONE
2. Re_1.0e+06/aoa_+02.00     FAILED
3. Re_2.0e+06/aoa_+00.00     Prepped
4. Re_2.0e+06/aoa_+02.00     Prepped
5. Re_1.0e+06/aoa_-04.00     Setup
6. Re_1.0e+06/aoa_-02.00     Setup
================================================
TOTAL = 6; SUCCESS = 1; FAILED = 1
================================================
```

6.2 Mainpulating CML input files with CPL

CPL provides a pythonic interface to read, create, modify, and write out input files necessary for running simulations using CML executables within a case directory. Users can interact with input files as python objects and use python data structures and functions to manipulate them. The modified objects can then be written out to files and CPL will pretty-print the files in the apporpriate locations in the case directory. Most CML/OpenFOAM objects have a one-to-one correspondence with python data structures within CPL. For example, OpenFOAM dictionaries are accessed as Python dictionaries, specifically an instance of *CaelusDict* which provides both attribute and dictionary-style access to entries. FOAM `List<Scalar>` data types are accessible as NumPy arrays, whereas generic lists containing mixed datatype entries (e.g., the *blocks* entry in `blockMeshDict`) are represented as lists.

This tutorial provides a walkthrough of using `caelus.io` module to read, manipulate, and write out input files in a CML case directory. The code snippets shown in this tutorial will use the `ACCM_airFoil2D` tutorial in the *$PROJECT_DIR*`/tutorials/incompressible/simpleSolver/ras/ACCM_airFoil2D` directory. To execute the example code snippets shown in this tutorial, it is recommended that execute them from this case directory and have the following modules loaded in your script or interactive shell

```
import numpy as np
import caelus.io as cio
```

The most general interface to a CML input file is through the *DictFile* class. It provides three ways of creating an input file object that can be manipulated using CPL in python scripts. Using the constructor creates a default CML file object that can be populated by the user. In most situations, however, the user would load a template file using *read_if_present()* or *load()* functions. The former as the name indicates will load the file if present in the case directory, whereas the latter will generate an error if the file doesn't exist. The first example will use

`system/controlDict` file to show how the user can load, examine, and manipulate the contents of a simple input file.

```
# Load the controlDict file from the case directory
cdict = cio.DictFile.load("system/controlDict")

# Show the keywords present in the controlDict file
print(cdict.keys())

# Change the variable 'startFrom' to 'latestTime'
cdict['startFrom'] = 'latestTime'

# Change 'writeFormat' to 'binary
cdict['writeFormat'] = 'binary'

# Show the current state of the controlDict contents
print(cdict)

# Save the updated controlDict file to the case directory
cdict.write()
```

The next example uses the *DictFile* to modify the `0/U`. For the purposes of this demonstration, we will change the inflow conditions from $\alpha = 0^\circ$ angle of attack to a flow at $\alpha = 6^\circ$ angle of attack.

```
# Load the U field file
ufile = cio.DictFile.load("0/U")

# Access the internalField variable
internal = ufile['internalField']
```

If you print out *ufile['internalField']* you will notice that it is an instance of *Field* that contains two attributes: `ftype` representing the field type (`uniform` or `nonuniform`), and `value` that contains the value of the field. In this the present example, we will access the uniform velocity field value and update it with the u and v velocities corresponding to $\alpha = 6^\circ$.

```
# Access the wind speed
wspd = internal.value[0]

# Compute u and v components
aoa_rad = np.radians(6.0)
uvel = wspd * np.cos(aoa_rad)
vvel = wspd * np.sin(aoa_rad)

# Update the velocity field
internal.value[0] = uvel
internal.value[1] = vvel

# Update the inlet value also (note attribute-style access)
inlet = ufile['boundaryField'].inlet
inlet.value = internal.value

# Check the current state of the 0/U file
print(ufile)

# Write the updated 0/U file
ufile.write()
```

6.2.1 Specialized CPL classes for CML input files

While *DictFile* provides a generic interface to all CML input files, CPL also defines specialized classes that provide additional functionality for those specific input files. The available classes that provide customized functionality are listed below

ControlDict	system/controlDict interface
FvSchemes	system/fvSchemes interface
FvSolution	system/fvSolution interface
DecomposeParDict	system/decomposeParDict interface
TransportProperties	constant/transportProperties interface
TurbulenceProperties	constant/turbulenceProperties interface
RASProperties	constant/RASProperties interface
LESProperties	constant/LESProperties interface
BlockMeshDict	constant/polyMesh/blockMeshDict interface

The specialized classes provide the ability to create default entries as well as provide a limited amount of syntax checking to ensure that the keywords contain acceptable values. It also allows attribute style access (in addition to dictionary style access) for the keywords present in the input file. The *read_if_present()* method is really useful with the specialized classes, as the user does not have to provide the file name, as shown below

```
# Load files from system directory
cdict = cio.ControlDict.read_if_present()
fvsol = cio.FvSolution.read_if_present()
fvsch = cio.FvSchemes.read_if_present()
```

For example, when using the *DictFile* interface with `system/controlDict` file, the user could assign any arbitrary value to `startFrom`. However, when using the *ControlDict*, CPL will raise an error if the user provides invalid value

```
# Attempt to pass invalid value will raise a ValueError as shown below
cdict.startFrom = "bananas"
# ValueError: ControlDict: Invalid option for 'startFrom'. Valid options are:
#    ('firstTime', 'startTime', 'latestTime')

# The keywords in file can be accessed either as attributes or keys
print ( cdict.stopAt, cdict['stopAt'])
```

6.2.2 Accessing keywords with special characters

While most keywords can be accessed as attributes, certain CML/OpenFOAM keywords contain invalid characters and therefore must be accessed as dictionary keys only. The `fvSolution` and `fvSchemes` provide good examples of such keywords.

```
# Accessing the divSchemes for specific equation must use dictionary style
# access
divU = fvsch.divSchemes["div(phi,U)"]

# Accessing the "(k|omega|nuTilda)" solver in fvSolution
turbSolver = fvsol.solvers['"(k|omega|nuTilda)"']
```

Note the nested quotation marks for the `"(k|omega|nuTilda)"` keyword. CML/OpenFOAM requires the double quotes because keyword starts with a non-alphabetical character. Wrapping the entire thing in single quotes protects the double quotes within Python.

6.2.3 Input files for turbulence models

```
# Load the TurbulenceProperties file
tprops = cio.TurbulenceProperties.read_if_present()

# Examine the type of turbulence model being used
print(tprops.simulationType)

# Get an instance of the model input file (returns None if laminar)
rans = tprops.get_turb_file()

# Show the model and coeffs
print(rans.model)
print(rans.coeffs)

# Options common to both RASProperties and LESProperties
rans.turbulence   # Flag indicating if turbulence is active
rans.printCoeffs  # Flag indicating whether coeffs are printed

# Switch to k-omega SST model
rans.model = "kOmegaSST"
# Note that coeffs has switched to 'kOmegaSSTCoeffs' present in input file
print(rans.coeffs)

# Turn curvature correction on (this updates kOmegaSSTCoeffs now)
rans.coeffs.curvatureCorrection = "on"

# Note, we can still access SpalartAllmarasCoeffs manually
# However, need to use dictionary style access
sacoeffs = rans['SpalartAllmarasCoeffs']

# Can still change S-A coeffs if necessary
sacoeffs.curvatureCorrection = 'off'

# Save updated RASProperties file
rans.write()
```

In the next example, we will change the turbulence model from RANS to LES and let CPL generate a default LESProperties file for us.

```
# Load the TurbulenceProperties file
tprops = cio.TurbulenceProperties.read_if_present()

# Switch to LES model
tprops.simulationType = "LESModel"

# Get the default LESProperties file generated by CPL
les = tprops.get_turb_file()

# Show default values created by CPL
print(les)

# Set up the appropriate coefficients for Smagorinsky
coeffs = les.coeffs
coeffs.ce = 1.05
coeffs.ck = 0.07
```

(continues on next page)

(continued from previous page)

```
# Write out the updated files
les.write()
tprops.write()
```

6.3 Custom scripts using CPL

CPL can be used to create custome CFD workflows that may fall outside the capabilities of the command-line applications. The classes used to build the command-line applications can likewise be use to create custom Python scripts, as shown with following example.

This tutorial mimics the workflow of the task file used for the VOF mutliphase solver *damBreak* tutorial provided with Caelus. To follow along it is recommended that the user download the `custom CPL script`. It is assumed the user is executing the script from within the `$CAELUS_PROJECT_DIR/tutorials/multiphase/vof/vofSolver/ras/damBreak` directory. To use CPL's Python interface directly, the user needs to ensure is CPL installed, preferably in a conda or virtualenv environment (see: *Installing Caelus Python Library (CPL)*). As a Python script, other non-CPL functionality can be used in coordination with CPL (e.g. *matplotlib*).

```
import os
import sys
import shutil

import matplotlib.pyplot as plt
```

Several CPL classes and methods are required. Refer to the CPL Python API docs (*caelus*) for a complete listing of modules and associated functionality.

```
from caelus.config.cmlenv import cml_get_version
from caelus.io import DictFile, DecomposeParDict
from caelus.run.cmd import CaelusCmd
from caelus.run.core import get_mpi_size
from caelus.post.logs import LogProcessor, SolverLog
from caelus.post.plots import CaelusPlot
```

An environment specifies the particular CML version and installation location. This examples loads the default (no argument to *cml_get_version* returns the default).

```
print("Searching for default caelus version...")
cenv_default = cml_get_version()

cenv = cenv_default
print("Using Caelus version: " + cenv.version)
print("Caelus path: " + cenv.project_dir)
```

Commands are run using *CaelusCmd*. The environment to the job manager object. The command is executed by calling the object and a boolean is returned to enable status checking. Here, the meshing application, *blockMesh*, is run.

```
status = 0
print("Executing blockMesh... ")
caelus_cmd = CaelusCmd("blockMesh", cml_env=cenv)
status = caelus_cmd()
if status != 0:
    print("ERROR generating blockMesh. Exiting!")
    sys.exit(1)
```

Use built-in Python modules for filesystem related tasks.

```
shutil.copy2("0/alpha1.org", "0/alpha1")
```

The solution is initialized solution using *funkySetFields* with the *CaelusCmd* as shown previously.

```
status = 0
print("Executing funkySetFields... ")
caelus_cmd = CaelusCmd("funkySetFields", cml_env=cenv)
caelus_cmd.cml_exe_args = "-latestTime"
status = caelus_cmd()
if status != 0:
    print("ERROR running funkySetFields. Exiting!")
    sys.exit(1)
```

An automated way to detect and set up a parallel run is to check for a `system/decomposeParDict` file, use the *CaelusDict* class to retrieve the *numberOfSubdomains* parameter, and set the number of MPI ranks to run applications with.

```
decomp_dict = DecomposeParDict.read_if_present()

parallel = True if decomp_dict['numberOfSubDomains'] > 1 else False

status = 0
solver_cmd = CaelusCmd("vofSolver", cml_env=cenv)

if parallel:
    print("Executing decomposePar... ")
    decomp_cmd = CaelusCmd("decomposePar", cml_env=cenv)
    decomp_cmd.cml_exe_args = ("-force")
    status = decomp_cmd()
    if status != 0:
        print("ERROR running decomposePar. Exiting!")
        sys.exit(1)
    solver_cmd.num_mpi_ranks = decomp_dict['numberOfSubdomains']
    solver_cmd.parallel = True
    print("Executing vofSolver in parallel on %d cores..."%solver_cmd.num_mpi_ranks)

else:
    print("Executing vofSolver...")

status = solver_cmd()
if status != 0:
    print("ERROR running vofSolver. Exiting!")
    sys.exit(1)
```

Finally, the *SolverLog* class is invoked to parse the log file and generate a plot of the residuals.

```
print("Processing logs... ")
clog = SolverLog(logfile="vofSolver.log")
cplot = CaelusPlot(clog.casedir)
cplot.plot_continuity_errors = True
cplot.plot_residuals_hist(plotfile="residuals.png")
```

Part II

Developer Manual

CHAPTER 7

Caelus Python API

7.1 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.

7.1.1 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**(***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 caelus.utils.struct.StructYAMLLoader

yaml_encoder
alias of caelus.utils.struct.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

7.1.2 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.

user_bindir
Return path to user bin directory

user_dir
Return the user directory

user_libdir
Return path to user lib 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

7.2 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.
osutils	Miscellaneous utilities

7.2.1 Struct Module

Implements *Struct*.

class caelus.utils.struct.**Struct**(***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

7.2.2 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.**path_exists**(*pname*)
Check path of the directory exists.

This function expands the user home directory as well as any shell variables found in the path provided and checks if that path exists.

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

Returns True if path exists

Return type bool

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.**split_path**(*fname*)
Split a path into directory, basename, extension

Returns (directory, basename, extension)

Return type tuple

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

7.3 caelus.run – CML Execution Utilities

7.3.1 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_change_inputs(*options*)
Change input files in case directory

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.

This method is an interface to `CaelusCmd`

cmd_run_python(*options*)
Execute a python script

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.

7.3.2 CML Simulation

This module defines *CMLSimulation* that provides a pythonic interface to detail with a CML case directory. In addition to implementing methods to perform actions, it also tracks the state of the analysis at any given time.

The module also provides an abstract interface *CMLSimCollection* that provides basic infrastructure to manage and manipulate a collection of simulations as a group.

class caelus.run.case.**CMLSimCollection**(*name*, *env=None*, *basedir=None*)
Bases: caelus.utils.tojson.JSONSerializer

Interface representing a collection of cases

Implementations must implement *setup()* that provides a concrete implementation of how the case is setup (either from a template or otherwise).

Provides *prep()*, *solve()*, *post()*, and *status()* to interact with the collection as a whole. Prep, solve, and post can accept a list of shell-style wildcard patterns that will restrict the actions to matching cases only.

Parameters

- **name** (*str*) – Unique name for this parametric run
- **env** (CMLEnv) – CML excution environment
- **basedir** (*path*) – Path where analysis directory is created

filter_cases(*patterns*)
Filter the cases based on a list of patterns

The patterns are shell-style wildcard strings to match case directory names.

Parameters **patterns** (*list*) – A list of one or more patterns

classmethod load(*env=None*, *casedir=None*, *json_file=None*)
Reload a persisted analysis group

Parameters

- **env** (CMLEnv) – Environment for the analysis
- **casedir** (*path*) – Path to the case directory
- **json_file** (*filename*) – Persistence information

post(*cnames=None*, *force=False*)
Run post-processing tasks on the cases

Parameters

- **cnames** (*list*) – Shell-style wildcard patterns
- **force** (*bool*) – Force rerun

prep(*cnames=None*, *force=False*)
Run prep actions on the cases

Parameters

- **cnames** (*list*) – Shell-style wildcard patterns
- **force** (*bool*) – Force rerun

save_state(***kwargs*)
Dump persistence file in JSON format

setup()
Logic to set up the analysis

classmethod simulation_class()
Concrete instance of a Simulation

Default is *CMLSimulation*

solve(*cnames=None*, *force=False*)
Run solve actions on the cases

Parameters

- **cnames** (*list*) – Shell-style wildcard patterns
- **force** (*bool*) – Force rerun

status()
Return the status of the runs

Yields *tuple* – (name, status) for each case

basedir = None
Location where parametric run setup is located

case_names = None
Names of cases

casedir = None
Location of the parametric run

cases = None
List of CMLSimulation instances

env = None
CML execution environment

name = None
Unique name for this parametric collection of cases

class caelus.run.case.**CMLSimMeta**(*name*, *bases*, *cdict*)
Bases: type

Decorator to add dictfile accessors to CMLSimulation

add_dictfile_attrs(*attrmap*)
Create getters for dictionary file objects

process_attr(*key*, *value*)
Create the attribute

class caelus.run.case.**CMLSimulation**(*case_name*, *cml_env=None*, *basedir=None*, *parent=None*)
Bases: caelus.utils.tojson.JSONSerializer

Pythonic interface to CML/OpenFOAM simulation

This class defines the notion of an analysis. It provides methods to interact with an analysis directory from within python, and provides basic infrastructure to track the status of the simulation.

After successful setup(), the simulation moves through a series of stages, that can be queried via *status()* method:

Status	Description
Setup	Case setup successfully
Prepped	Pre-processing completed
Submitted	Solver initialized
Running	Solver is running
Solved	Solve has completed
DONE	Post-processing completed
FAILED	Some action failed

Parameters

- **case_name** (*str*) – Unique identifier for the case
- **env** (CMLEnv) – CML environment used to setup/run the case
- **basedir** (*path*) – Location where the case is located/created
- **parent** (CMLSimCollection) – Instance of the group manager

case_log(*force_reload=False*)
Return a SolverLog instance for this case

clean(*preserve_extra=None*, *preserve_polymesh=True*, *preserve_zero=True*, *preserve_times=False*, *preserve_processors=False*)
Clean an existing case directory.

Parameters

- **preserve_extra** (*list*) – List of shell wildcard patterns to preserve
- **preserve_polymesh** (*bool*) – If False, purges polyMesh directory
- **preserve_zero** (*bool*) – If False, removes the 0 directory
- **preserve_times** (*bool*) – If False, removes the time directories
- **preserve_processors** (*bool*) – If False, removes processor directories

clone(*template_dir*, *copy_polymesh=True*, *copy_zero=True*, *copy_scripts=True*, *extra_patterns=None*, *clean_if_present=False*)
Create the case directory from a given template

Parameters

- **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
- **clean_if_present** (*bool*) – Overwrite existing case

Raises IOError – If `casedir` exists and `clean_if_present` is False

decompose_case(*dep_job_id=None*, *force=False*)
Decompose case if necessary

Parameters

- **dep_job_id** (*int*) – Job ID to wait for

- **force** (*bool*) – Force rerun of decomposition tasks

get_input_dict(*dictname*)
Return a CPL instance of the input file

For standard input files, prefer to use the accessors directly instead of this method. For example, case.controlDict, case.turbulenceProperties, etc.

Parameters **dictname** (*str*) – File name relative to case directory

classmethod load(*env=None*, *casedir=None*, *parent=None*, *json_file=None*)
Loads a previously setup case from persistence file

post_case(*post_tasks=None*, *force=False*)
Execute post-processing tasks for this case

prep_case(*prep_tasks=None*, *force=False*)
Execute pre-processing tasks for this case

If not tasks are provided, then uses the section `prep` from `run_configuration` that was passed during the setup phase.

Parameters

- **prep_tasks** (*list*) – List of tasks for Tasks
- **force** (*bool*) – Force prep again if already run

reconstruct_case()
Reconstruct a parallel case

run_tasks(*task_file=None*)
Run tasks within case directory using the tasks file

save_state(***kwargs*)
Dump persistence file in JSON format

solve(*force=False*)
Execute solve for this case

Parameters **force** (*bool*) – Force resubmit even if previously submitted

status()
Determine status of the run

Returns Status of the run as a string

Return type str

update(*input_mods=None*)
Update the input files within a case directory

Parameters **input_mods** (CaelusDict) – Dictionary with changes

LESProperties
Return LESProperties instance for this case

RASProperties
Return RASProperties instance for this case

basedir = None
Root directory containing the case

blockMeshDict
Return blockMeshDict instance for this case

casedir = None
Absolute path to the case directory

changeDictionaryDict
Return changeDictionaryDict instance for this case

cmlControls
Return cmlControls instance for this case

controlDict
Return controlDict instance for this case

decomposeParDict
Return decomposeParDict instance for this case

env = None
CML environment used to run this case

fvSchemes
Return fvSchemes instance for this case

fvSolution
Return fvSolution instance for this case

job_ids = None
Job IDs for SLURM/PBS jobs (internal use only)

logfile
The log file for the solver

name = None
Unique name for this case

parent = None
Instance of CMLSimCollection if part of a larger set

run_config = None
Dictionary containing run configuration (internal use only)

run_flags = None
Dictionary tracking status (internal use only)

solver
Return the solver used for this case

task_file = 'caelus_tasks.yaml'
Name of the task file for this case

transportProperties
Return transportProperties instance for this case

turbulenceProperties
Return turbulenceProperties instance for this case

7.3.3 CML Parametric Run Manager

class caelus.run.parametric.**CMLParametricRun**(*name*, *sim_dict*, *env=None*, *basedir=None*)
Bases: *caelus.run.case.CMLSimCollection*

A class to handle parametric runs

Parameters

- **name** (*str*) – Unique name for this parametric run
- **sim_dict** (CaelusDict) – Dictionary with simulation settings
- **env** (CMLEnv) – CML execution environment
- **basedir** (*path*) – Path where the parametric run directories are created

setup()
Setup the parametric case directories

setup_case(*cname*, *tmpl_dir*, *cparams*, *runconf*, *clone_opts*)
Helper function to setup the cases

sim_dict = None
Dictionary containing the run settings

caelus.run.parametric.**iter_case_params**(*sim_options*)
Normalize the keys and yield all possible run setups

caelus.run.parametric.**normalize_variable_param**(*varspec*)
Helper function to normalize the different run matrix options

7.3.4 Caelus Job Manager Interface

class caelus.run.cmd.**CaelusCmd**(*cml_exe*, *casedir=None*, *cml_env=None*, *output_file=None*)
Bases: object

CML execution interface.

CaelusCmd is a high-level interface to execute CML binaries within an appropriate enviroment across different operating systems.

Parameters

- **cml_exe** (*str*) – The binary to be executed (e.g., blockMesh)
- **casedir** (*path*) – Absolute path to case directory
- **cml_env** (CMLEnv) – Environment used to run the executable
- **output_file** (*file*) – Filename to redirect all output

prepare_exe_cmd()
Prepare the shell command and return as a string

Returns The CML command invocation with all its options

prepare_shell_cmd()
Prepare the complete command line string as executed

casedir = None
Case directory

cfg = None
CPL configuration object

cml_env = None
CML version used for this run

cml_exe = None
CML program to be executed

cml_exe_args = None
Arguments passed to the CML executable

mpi_extra_args
Extra arguments passed to MPI

num_mpi_ranks
Number of MPI ranks for a parallel run

output_file = None
Log file where all output and error are captured

parallel = None
Is this a parallel run

runner = None
Handle to the subprocess instance running the command

7.3.5 CML Execution Utilities

caelus.run.core.**clean_casedir**(*casedir*, *preserve_extra=None*, *preserve_zero=True*, *preserve_times=False*, *preserve_processors=False*, *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
- **preserve_times** (*bool*) – If False, removes the time directories
- **preserve_processors** (*bool*) – If False, removes processor directories

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)

7.3.6 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*)

Bases: object

Abstract base class for job submission interface

name
Job name

Type str

queue
Queue/partition where job is submitted

Type str

account
Account the job is charged to

Type str

num_nodes
Number of nodes requested

Type int

num_ranks
Number of MPI ranks

Type int

stdout
Filename where standard out is redirected

Type path

stderr
Filename where standard error is redirected

Type path

join_outputs
Merge stdout/stderr to same file

Type bool

mail_opts
Mail options (see specific queue implementation)

Type str

email_address
Email address for notifications

Type str

qos
Quality of service

Type str

time_limit
Wall clock time limit

Type str

shell
shell to use for scripts

Type str

mpi_extra_args
additional arguments for MPI

Type str

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*)
Bases: *caelus.run.hpc_queue.HPCQueue*

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*)
Bases: *caelus.run.hpc_queue.SerialJob*

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*)
Bases: *caelus.run.hpc_queue.HPCQueue*

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*)
Bases: *caelus.run.hpc_queue.HPCQueue*

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

caelus.run.hpc_queue.**python_execute**(*pyscript*, *script_args=''*, *env=None*, *log_file=None*, *log_to_file=True*)
Execute a python script with the right environment

This function will setup the correct CPL and CML environment and execute the python script within this environment. The user should only provide the name of the script and not `python script` as this it is this functions job to detect the correct python executable and execute within that environment.

If `log_file` isn't provided it automatically creates a "py_*.log" file to redirect output messages from the script where `*` is replaced with the basename of the python script.

Parameters

- **pyscript** (*path*) – Filename of the python script
- **script_args** (*str*) – Extra arguments to be passed to the python script
- **env** (CMLEnv) – CML environment used for execution
- **log_file** (*filename*) – Filename to redirect output to
- **log_to_file** (*bool*) – Should outputs be redirected to log file

Returns The status of the execution

Return type status (int)

7.4 caelus.post – Post-processing utilities

Provides log analysis and plotting utilities

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

7.4.1 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(***kwargs*)

Process the bounding lines

completion_processor(***kwargs*)

Process End line indicating solver completion

continuity_processor(***kwargs*)

Process continuity error lines from log file

convergence_processor(***kwargs*)

Process convergence information (steady solvers only)

courant_processor(***kwargs*)

Process Courant Number lines

exec_time_processor(***kwargs*)

Process execution/clock time lines

exiting_processor(***kwargs*)

Process exiting option

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

fatal_error_processor(***kwargs*)
Process CAELUS FATAL ERROR line

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

time_processor(***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

failed = None
Flag indicating whether the solver failed

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

7.4.2 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(***kwargs*)
Capture continuity errors for plot updates

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

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

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

time_processor(***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

7.5 caelus.io – CML Input File Manipulation

7.5.1 Caelus/OpenFOAM Input File Interface

class `caelus.io.dictfile.`**`BlockMeshDict`**(*filename=None*, *populate_defaults=True*)
Bases: *caelus.io.dictfile.DictFile*

constant/polyMesh/blockMeshDict interface

Parameters **`filename`** (*path*) – Path to the input file

`create_default_entries`()
Create defaults from property list

`blocks`

`boundary`

`convertToMeters`

`edges`

`mergePatchPairs`

`vertices`

class `caelus.io.dictfile.`**`ChangeDictionaryDict`**(*filename=None*, *populate_defaults=True*)
Bases: *caelus.io.dictfile.DictFile*

system/changeDictionaryDict interface

Parameters **`filename`** (*path*) – Path to the input file

`create_default_entries`()
Create defaults from property list

`dictionaryReplacement`

class `caelus.io.dictfile.`**`CmlControls`**(*filename=None*, *populate_defaults=True*)
Bases: *caelus.io.dictfile.DictFile*

cmlControls interface

Parameters **`filename`** (*path*) – Path to the input file

class `caelus.io.dictfile.`**`ControlDict`**(*filename=None*, *populate_defaults=True*)
Bases: *caelus.io.dictfile.DictFile*

system/controlDict interface

Parameters **`filename`** (*path*) – Path to the input file

create_default_entries()
Create defaults from property list

adjustTimeStep

application

deltaT

endTime

functions
function object definitions in controlDict

graphFormat

maxCo

purgeWrite

runTimeModifiable

startFrom

startTime

stopAt

timeFormat

timePrecision

writeCompression

writeControl

writeFormat

writeInterval

writePrecision

class caelus.io.dictfile.**DecomposeParDict**(*filename=None*, *populate_defaults=True*)
Bases: *caelus.io.dictfile.DictFile*

system/decomposeParDict interface

Parameters **filename** (*path*) – Path to the input file

create_default_entries()
Create defaults from property list

method

numberOfSubdomains

class caelus.io.dictfile.**DictFile**(*filename=None*, *populate_defaults=True*)
Bases: object

Caelus/OpenFOAM input file reader/writer

The default constructor does not read a file, but instead creates a new input file object. If a property list is provided, this is used to initialize the default entries. To read an existing file, the use of *DictFile.read_if_present()* method is recommended.

Parameters **filename** (*path*) – Path to the input file

create_default_entries()
Create default entries for this file

create_header()
Create a default header

keys()
Return list of variable names in the dictionary

classmethod load(*filename=None*, *debug=False*)
Load a Caelus input file from disk

Parameters

- **filename** (*path*) – Path to the input files
- **debug** (*bool*) – Turn on detailed errors

merge(**args*)
Merge entries from one dictionary to another

classmethod read_if_present(*casedir=None*, *filename=None*, *debug=False*, *populate_defaults=True*)
Read the file if present, else create object with default values

Parameters

- **casedir** (*path*) – Path to the case directory
- **filename** (*path*) – Filename to read
- **debug** (*bool*) – Turn on detailed errors
- **populate_defaults** (*bool*) – Populate the defaults

write(*casedir=None*, *filename=None*, *update_object=True*, *write_header=True*)
Write a formatted Caelus input file

Parameters

- **casedir** (*path*) – Path to the case directory
- **filename** (*path*) – Filename to write
- **update_object** (*bool*) – Ensure object type is consistent
- **write_header** (*bool*) – Write header for the file

contents
Access entries within the Caelus CML dictionary

data = None
Contents of the file as a dictionary suitable for manipulation

filename = None
File to read/write data

header = None
Contents of the FoamFile sub-dictionary in the file

class caelus.io.dictfile.**DictMeta**(*name*, *bases*, *cdict*)
Bases: type

Create property methods and add validation for properties.

This metaclass implements the boilerplate code necessary to add getter/setters for various entries found in a Caelus input file. It expects a class variable `_dict_properties` that contains tuples for the various entries in the input file. The tuple can be of two forms:

- (name, default_value)

- (name, default_value, valid_values)

If the default_value is not None, then this value will be used to automatically initialize the particular entry by *create_default_entries()* method. If `valid_values` are provided, any attempt to set/modify this value will be checked to ensure that only the allowed values are used.

process_defaults(*proplist*)
Process default entries

process_properties(*proplist*)
Create getters/setters for properties

process_property(*plist*)
Process a property

class caelus.io.dictfile.**FvSchemes**(*filename=None*, *populate_defaults=True*)
Bases: *caelus.io.dictfile.DictFile*

system/fvSchemes interface

Parameters **filename** (*path*) – Path to the input file

create_default_entries()
Create defaults from property list

ddtSchemes

divSchemes

fluxRequired

gradSchemes

interpolationSchemes

laplacianSchemes

snGradSchemes

class caelus.io.dictfile.**FvSolution**(*filename=None*, *populate_defaults=True*)
Bases: *caelus.io.dictfile.DictFile*

system/fvSolution interface

Parameters **filename** (*path*) – Path to the input file

create_default_entries()
Create defaults from property list

PIMPLE

PISO

SIMPLE

potentialFlow

relaxationFactors

solvers

class caelus.io.dictfile.**LESProperties**(*filename=None*, *populate_defaults=True*)
Bases: *caelus.io.dictfile.TurbModelProps*

constant/LESProperties interface

Parameters **filename** (*path*) – Path to the input file

create_default_entries()
Create the default turbulence model entries

In addition to the default options specified in turbulence properties class, this also triggers the default entries for delta.

delta
LES delta

class caelus.io.dictfile.**PolyMeshBoundary**(*filename=None*, *populate_defaults=True*)
Bases: *caelus.io.dictfile.DictFile*

constant/polyMesh/boundary interface

Parameters **filename** (*path*) – Path to the input file

class caelus.io.dictfile.**RASProperties**(*filename=None*, *populate_defaults=True*)
Bases: *caelus.io.dictfile.TurbModelProps*

constant/RASProperties interface

Parameters **filename** (*path*) – Path to the input file

class caelus.io.dictfile.**TransportProperties**(*filename=None*, *populate_defaults=True*)
Bases: *caelus.io.dictfile.DictFile*

constant/transportProperties interface

Parameters **filename** (*path*) – Path to the input file

create_default_entries()
Create defaults from property list

transportModel

class caelus.io.dictfile.**TurbModelProps**(*filename=None*, *populate_defaults=True*)
Bases: *caelus.io.dictfile.DictFile*

Common interface for LES/RAS models

Parameters **filename** (*path*) – Path to the input file

create_default_entries()
Create defaults from property list

coeffs
Turbulence model coefficients

This represents the sub-dictionary (e.g., kOmegaSSTCoeffs, SmagorinksyCoeffs) containing the additional parameters necessary for the turbulence model. The accessor automatically populates the right name when generating the dictionary depending on the turbulence model selected.

model
Turbulence model

Depending on the type (RANS or LES), this is the entry RASModel or LESModel respectively in the RASProperties and LESProperties file. To simplify access, it is simply named model here.

printCoeffs

turbulence

class caelus.io.dictfile.**TurbulenceProperties**(*filename=None*, *populate_defaults=True*)
Bases: *caelus.io.dictfile.DictFile*

constant/turbulenceProperties interface

Parameters **filename** (*path*) – Path to the input file

create_default_entries()
Create defaults from property list

get_turb_file()
Return the appropriate RASProperties or LESProperties file

simulationType

caelus.io.dictfile.**cml_std_files = {'LESProperties': <class 'caelus.io.dictfile.LESPropert**
Mapping of standard files known to exist in a case directory

7.5.2 Caelus/OpenFOAM Dictionary Implementation

class caelus.io.caelusdict.**CaelusDict**(***kwds*)
Bases: *caelus.utils.struct.Struct*

Caelus Input File Dictionary

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 caelus.utils.struct.StructYAMLLoader

yaml_encoder
alias of caelus.utils.struct.StructYAMLDumper

Caelus/OpenFOAM Input File Datatypes

class caelus.io.dtypes.**BoundaryList**(*value*)
Bases: *caelus.io.dtypes.FoamType*

polyMesh/boundary file

write_value(*fh=<open file '<stdout>', mode 'w'>*, *indent_str=''*)
Write as a Caelus/OpenFOAM entry

This method is called by *DictPrinter* to format the data suitable for writing to a Caelus input file that can be read by the solvers.

Parameters

- **fh** (*file*) – A valid file handle
- **indent_str** (*str*) – Padding for indentation

class caelus.io.dtypes.**CalcDirective**(*directive*, *value*)
Bases: *caelus.io.dtypes.FoamType*

A #calc directive entry

Example:: radHalfAngle #calc "degToRad($halfAngle)";

write_value(*fh=<open file '<stdout>', mode 'w'>*, *indent_str=''*)
Write out the dimensional value

class caelus.io.dtypes.**CodeStream**(*value*)
Bases: *caelus.io.dtypes.FoamType*

A codestream entry

Contains C++ code that can be compiled and executed to determine dictionary parameters.

write_value (*fh=<open file '<stdout>', mode 'w'>*, *indent_str=''*)
Write out the dimensional value

class caelus.io.dtypes.**DimValue** (*name*, *dims*, *value*)
Bases: *caelus.io.dtypes.FoamType*

A dimensioned value

A dimensioned value contains three parts: the name, units, and the value. Units are of type *Dimension* and value can be a scalar, vector, tensor or a symmetric tensor.

write_value (*fh=<open file '<stdout>', mode 'w'>*, *indent_str=''*)
Write out the dimensional value

class caelus.io.dtypes.**Dimension** (*units=None*, ***kwargs*)
Bases: *caelus.io.dtypes.FoamType*

Caelus dimensional units

Represents the units of a dimensional quantity as an array of seven integers that represent the powers of the fundamental units: mass, length, time, temperature, quantity, current, and luminous intensity.

Provide an array of individual units as keyword arguments

Parameters **units** (*list*) – A list of 5 or 7 entries

write_value (*fh=<open file '<stdout>', mode 'w'>*, *indent_str=''*)
Write out the dimensions

Parameters

- **fh** (*file*) – A valid file handle
- **indent_str** (*str*) – Padding for indentation

class caelus.io.dtypes.**Directive** (*directive*, *value*)
Bases: *caelus.io.dtypes.FoamType*

A Caelus directive type

Directives are keyword-less entries that indicate certain processing actions and begin with a hash (#) symbol. For example, the `#includeEtc` directive that can be used to include files from `foamEtc` directory.

write_value (*fh=<open file '<stdout>', mode 'w'>*, *indent_str=''*)
Write out the dimensional value

directive = None
Type of directive (str)

value = None
Value of the directive (e.g., file to be included)

class caelus.io.dtypes.**Field** (*ftype*, *value*)
Bases: *caelus.io.dtypes.FoamType*

A field declaration

This class represents both uniform and non-uniform fields. The attribute `ftype` indicates the type of field and the `value` contains the value for the given field. Uniform fields can be scalar, vector, tensor, or symmetric tensors. Non-uniform fields are typically a *ListTemplate* entity.

write_nonuniform (*fh=<open file '<stdout>', mode 'w'>*)
Write a non-uniform field

write_uniform (*fh=<open file '<stdout>', mode 'w'>*)
Write a uniform field

write_value(*fh=<open file '<stdout>', mode 'w'>*, *indent_str=''*)
Write value in OpenFOAM format

class caelus.io.dtypes.**FoamType**
Bases: object

Base class for a FOAM type

write_value(*fh=<open file '<stdout>', mode 'w'>*, *indent_str=''*)
Write as a Caelus/OpenFOAM entry

This method is called by *DictPrinter* to format the data suitable for writing to a Caelus input file that can be read by the solvers.

Parameters

- **fh** (*file*) – A valid file handle
- **indent_str** (*str*) – Padding for indentation

class caelus.io.dtypes.**ListTemplate**(*ltype*, *value*)
Bases: *caelus.io.dtypes.FoamType*

List<T> type entries

write_value(*fh=<open file '<stdout>', mode 'w'>*, *indent_str=''*)
Write out a List<T> value

class caelus.io.dtypes.**MacroSubstitution**(*value*)
Bases: *caelus.io.dtypes.FoamType*

Macro substition without keyword

write_value(*fh=<open file '<stdout>', mode 'w'>*, *indent_str=''*)
Write standalone macro substitution

class caelus.io.dtypes.**MultipleValues**(*value*)
Bases: *caelus.io.dtypes.FoamType*

Multiple values for single keyword

Example:: laplacian(nuEff,U) Gauss linear corrected;

Here "Gauss linear corrected" is stored as an instance of this class to disambiguate between multi-valued entries and plain lists.

write_value(*fh=<open file '<stdout>', mode 'w'>*, *indent_str=''*)
Write as a Caelus/OpenFOAM entry

This method is called by *DictPrinter* to format the data suitable for writing to a Caelus input file that can be read by the solvers.

Parameters

- **fh** (*file*) – A valid file handle
- **indent_str** (*str*) – Padding for indentation

7.5.3 Caelus Input File Pretty-printer

class caelus.io.printer.**DictPrinter**(*buf=<open file '<stdout>', mode 'w'>*, *tab_width=4*)
Bases: object

Caelus Input File Pretty-printer

Given a CaelusDict instance, this class will emit formatted data suitable for use with Caelus solvers

Parameters

- **buf** (*file handle*) – A valid buffer to output to
- **tab_width** (*int*) – Indentation width

write_dict(*value*)
Pretty-print a Caelus dictionary type

Parameters **value** (*Mapping*) – A valid python dict-like instance

write_dict_item(*key*, *value*, *nested=False*)
Pretty-print a dictionary entry

Parameters

- **key** (*str*) – Keyword for the parameter
- **value** (*object*) – Value for the keyword
- **nested** (*bool*) – Flag indicating whether the entries are nested

write_list(*value*, *recursive=False*)
Pretty-print a list entry

Lists are mixed-type data entries. Empty lists and short string lists are printed flat in the same line. All other lists have their entries printed on new lines.

Parameters

- **value** (*list*) – A list entry
- **recursive** (*bool*) – Flag indicating whether this list is part of another list or dict

write_ndarray(*value*, *recursive=False*)
Pretty-print a numeric list

Parameters

- **value** (*np.ndarray*) – Array object
- **recursive** (*bool*) – Flag indicating whether it is part of a list or dict

write_value(*value*, *recursive=False*, *indented=False*)
Pretty-print an RHS entry based on its type

Parameters

- **value** (*object*) – Value to be printed
- **recursive** (*bool*) – Flag indicating whether the value is part of a dictionary or a list
- **indented** (*bool*) – Flag indicating whether value must be indented

keyword_fmt = '%-20s'
Default width for keywords

class caelus.io.printer.**Indenter**(*tab_width=4*)
Bases: object

An indentation utility for use with DictPrinter

Parameters **tab_width** (*int*) – Default indentation width

dedent()
Dedent the tab

emit (*fh*)
Emit the leading identation

indent ()
Indent the tab

curr_indent = None
Current indentation column

indent_str
Return an indentation string

tab_width = None
Identation width

caelus.io.printer.**foam_writer** (**args*, ***kwds*)
Caelus/OpenFOAM file writer

Parameters **header** (CaelusDict) – The FoamFile entries

Yields *printer (DictPrinter)* – A dictionary printer for printing data

7.6 caelus.scripts – CLI App Utilities

7.6.1 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) v1.0.1'
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

Index

Symbols

A

B

C

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y