MaestroWF Documentation

Release 1.1.7

Francesco Di Natale

May 29, 2022

Contents:

CHAPTER 1

Getting Started

1.1 Maestro Docker Container

In order to set up the Docker container execute the following from the root of the Maestro repository:

```
$ docker --build -t maestrowf .
```

To launch the interactive shell of the Ubuntu image simply run:

```
$ docker run -it maestrowf
```

Once inside the Docker container, the following should bring up help:

```
$ maestro -h
```

For more information on using Dockerfiles, checkout Docker's *documentation <https://docs.docker.com/engine/reference/builder/>*.

1.2 Installing MaestroWF

MaestroWF can be installed via pip outside of Docker with the following:

```
$ pip install maestrowf
```

Note: Using a virtualenv is recommended.

Once installed run:

```
$ maestro -h

usage: maestro [-h] [-l LOGPATH] [-d DEBUG_LVL] [-c] {cancel,run,status} ...

The Maestro Workflow Conductor for specifiying, launching, and managing general␣
↪workflows.

positional arguments:
  {cancel,run,status}
    cancel              Cancel all running jobs.
    run                 Launch a study based on a specification
    status              Check the status of a running study.

optional arguments:
  -h, --help            show this help message and exit
  -l LOGPATH, --logpath LOGPATH
                        Alternate path to store program logging.
  -d DEBUG_LVL, --debug_lvl DEBUG_LVL
                        Level of logging messages to be output:
                        5 - Critical
                        4 - Error
                        3 - Warning
                        2 - Info (Default)
                        1 - Debug
  -c, --logstdout       Log to stdout in addition to a file. [Default: True]
```

CHAPTER 2

Quick Start Guide

If you haven't already done so, see installation instructions in *Getting Started*. For this guide, you will need to checkout Maestro via GitHub in order to use the provided samples.

2.1 Running the LULESH Study

This section will take you through the basics of using Maestro to launch a study locally. If you're looking for a more detailed breakdown of the LULESH study, skip to the *LULESH Specification Breakdown*.

LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) is a proxy application developed and open sourced by Lawrence Livermore National Laboratory. The application is capable of being compiled for both serial and distributed execution and provides a simple stand-in for an actual simulation code.

The LULESH study comes in two flavors: one for Unix systems and one for MacOSX systems. Both specifications are just about identical save for minor differences in `sed` commands. Simply pick the version for your system. In order to execute the LULESH sample study, execute the following command using Maestro from the root of the repository (using the Unix version as an example):

```
$ maestro run ./samples/lulesh/lulesh_sample1_unix.yaml -o ./tests/lulesh
```

Note: The `-o` flag is used for specifying a custom output path. Normally, Maestro creates a timestamped folder each time it is executed. The use of the `-o` is used here for consistency with provided logging.

Maestro begins by loading the specification and checks out a copy of LULESH from GitHub:

```
2018-08-07 00:58:56,887 - maestrowf.maestro:setup_logging:360 - INFO - INFO Logging␣
↪Level -- Enabled
2018-08-07 00:58:56,887 - maestrowf.maestro:setup_logging:361 - WARNING - WARNING␣
↪Logging Level -- Enabled
2018-08-07 00:58:56,887 - maestrowf.maestro:setup_logging:362 - CRITICAL - CRITICAL␣
↪Logging Level -- Enabled
```

(continues on next page)

(continued from previous page)

```
2018-08-07 00:58:56,887 - maestrowf.datastructures.core.study:__init__:195 - INFO -␣
↪OUTPUT_PATH = /home/travis/build/LLNL/maestrowf/testing/lulesh
2018-08-07 00:58:56,887 - maestrowf.datastructures.core.study:add_step:298 - INFO -␣
↪Adding step 'make-lulesh' to study 'lulesh_sample1'...
2018-08-07 00:58:56,888 - maestrowf.datastructures.core.study:add_step:298 - INFO -␣
↪Adding step 'run-lulesh' to study 'lulesh_sample1'...
2018-08-07 00:58:56,888 - maestrowf.datastructures.core.study:add_step:307 - INFO -␣
↪run-lulesh is dependent on make-lulesh. Creating edge (make-lulesh, run-lulesh)...
2018-08-07 00:58:56,888 - maestrowf.datastructures.core.study:add_step:298 - INFO -␣
↪Adding step 'post-process-lulesh' to study 'lulesh_sample1'...
2018-08-07 00:58:56,888 - maestrowf.datastructures.core.study:add_step:307 - INFO -␣
↪post-process-lulesh is dependent on run-lulesh_*. Creating edge (run-lulesh_*, post-
↪process-lulesh)...
2018-08-07 00:58:56,889 - maestrowf.datastructures.core.study:add_step:298 - INFO -␣
↪Adding step 'post-process-lulesh-trials' to study 'lulesh_sample1'...
2018-08-07 00:58:56,889 - maestrowf.datastructures.core.study:add_step:307 - INFO -␣
↪post-process-lulesh-trials is dependent on run-lulesh_*. Creating edge (run-lulesh_
↪*, post-process-lulesh-trials)...
2018-08-07 00:58:56,889 - maestrowf.datastructures.core.study:add_step:298 - INFO -␣
↪Adding step 'post-process-lulesh-size' to study 'lulesh_sample1'...
2018-08-07 00:58:56,889 - maestrowf.datastructures.core.study:add_step:307 - INFO -␣
↪post-process-lulesh-size is dependent on run-lulesh_*. Creating edge (run-lulesh_*,␣
↪post-process-lulesh-size)...
2018-08-07 00:58:56,890 - maestrowf.datastructures.core.study:setup_workspace:337 -␣
↪INFO - Setting up study workspace in '/home/travis/build/LLNL/maestrowf/testing/
↪lulesh'
2018-08-07 00:58:56,890 - maestrowf.datastructures.core.study:setup_environment:347 -␣
↪INFO - Environment is setting up.
2018-08-07 00:58:56,890 - maestrowf.datastructures.core.studyenvironment:acquire_
↪environment:191 - INFO - Acquiring dependencies
2018-08-07 00:58:56,890 - maestrowf.datastructures.core.studyenvironment:acquire_
↪environment:193 - INFO - Acquiring -- LULESH
2018-08-07 00:58:56,890 - maestrowf.datastructures.environment.
↪gitdependency:acquire:150 - INFO - Cloning LULESH from https://github.com/LLNL/
↪LULESH.git...
Cloning into '/home/travis/build/LLNL/maestrowf/testing/lulesh/LULESH'...
remote: Counting objects: 64, done.
remote: Compressing objects: 100% (40/40), done.
remote: Total 64 (delta 10), reused 43 (delta 6), pack-reused 16
Unpacking objects: 100% (64/64), done.
2018-08-07 00:58:57,306 - maestrowf.datastructures.core.study:configure_study:382 -␣
↪INFO -
----------------------------------------
Output path =               /home/travis/build/LLNL/maestrowf/testing/lulesh
Submission attempts =       1
Submission restart limit =  1
Submission throttle limit = 0
Use temporary directory =   False
----------------------------------------
2018-08-07 00:58:57,307 - maestrowf.datastructures.core.executiongraph:__init__:337 -␣
↪INFO -
----------------------------------------
Submission attempts =       1
Submission throttle limit = 0
Use temporary directory =   False
Tmp Dir =
----------------------------------------
```

Once set up is complete, Maestro will begin expanding the *Study* graph into an `ExecutionGraph`. The `ExecutionGraph` represents the complete execution plan for a study. The snippets below show some of the expected output which will be a mix of single steps and parameterized steps.

Singular steps (such as "make-lulesh") appear in the log as follows:

```
2018-08-07 00:58:57,307 - maestrowf.datastructures.core.study:_stage_
↪parameterized:431 - INFO -
 ================================================
 Processing step 'make-lulesh'
 ================================================
 2018-08-07 00:58:57,308 - maestrowf.datastructures.core.study:_stage_
↪parameterized:503 - INFO -
 ------------------------------------------------
 Adding step 'make-lulesh' (No parameters used)
 ------------------------------------------------
 2018-08-07 00:58:57,308 - maestrowf.datastructures.core.study:_stage_
↪parameterized:518 - INFO - Searching for workspaces...
 cmd = cd /home/travis/build/LLNL/maestrowf/testing/lulesh/LULESH
 sed -i 's/^CXX = $(MPICXX)/CXX = $(SERCXX)/' ./Makefile
 sed -i 's/^CXXFLAGS = -g -O3 -fopenmp/#CXXFLAGS = -g -O3 -fopenmp/' ./Makefile
 sed -i 's/^#LDFLAGS = -g -O3/LDFLAGS = -g -O3/' ./Makefile
 sed -i 's/^LDFLAGS = -g -O3 -fopenmp/#LDFLAGS = -g -O3 -fopenmp/' ./Makefile
 sed -i 's/^#CXXFLAGS = -g -O3 -I/CXXFLAGS = -g -O3 -I/' ./Makefile
 make clean
 make
```

Parameterized steps (such as "run-lulesh") appear by printing out their expansion such as in the snippet below. Each combination is printed as it is expanded, for each combination in the `global.parameters` section of the study based on the parameters the given step uses:

```
2018-08-07 00:58:57,308 - maestrowf.datastructures.core.study:_stage_
↪parameterized:431 - INFO -
================================================
Processing step 'run-lulesh'
================================================
2018-08-07 00:58:57,308 - maestrowf.datastructures.core.study:_stage_
↪parameterized:571 - INFO -
================================================
Expanding step 'run-lulesh'
================================================
-------- Used Parameters --------
set(['SIZE', 'ITERATIONS'])
---------------------------------
2018-08-07 00:58:57,308 - maestrowf.datastructures.core.study:_stage_
↪parameterized:578 - INFO -
**********************************
Combo [SIZE.10.TRIAL.1.ITER.10]
**********************************
2018-08-07 00:58:57,309 - maestrowf.datastructures.core.study:_stage_
↪parameterized:599 - INFO - Searching for workspaces...
cmd = /home/travis/build/LLNL/maestrowf/testing/lulesh/LULESH/lulesh2.0 -s 10 -i 10 -
↪p > SIZE.10.ITER.10.log
2018-08-07 00:58:57,309 - maestrowf.datastructures.core.study:_stage_
↪parameterized:630 - INFO - New cmd = /home/travis/build/LLNL/maestrowf/testing/
↪lulesh/LULESH/lulesh2.0 -s 10 -i 10 -p > SIZE.10.ITER.10.log
2018-08-07 00:58:57,309 - maestrowf.datastructures.core.study:_stage_
↪parameterized:640 - INFO - Processing regular dependencies.
```

(continues on next page)

(continued from previous page)

```
2018-08-07 00:58:57,309 - maestrowf.datastructures.core.study:_stage_
↪parameterized:648 - INFO - Adding edge (make-lulesh, run-lulesh_ITER.10.SIZE.10)...
2018-08-07 00:58:57,309 - maestrowf.datastructures.core.study:_stage_
↪parameterized:578 - INFO -
**********************************
Combo [SIZE.10.TRIAL.2.ITER.20]
**********************************
2018-08-07 00:58:57,309 - maestrowf.datastructures.core.study:_stage_
↪parameterized:599 - INFO - Searching for workspaces...
cmd = /home/travis/build/LLNL/maestrowf/testing/lulesh/LULESH/lulesh2.0 -s 10 -i 20 -
↪p > SIZE.10.ITER.20.log
2018-08-07 00:58:57,309 - maestrowf.datastructures.core.study:_stage_
↪parameterized:630 - INFO - New cmd = /home/travis/build/LLNL/maestrowf/testing/
↪lulesh/LULESH/lulesh2.0 -s 10 -i 20 -p > SIZE.10.ITER.20.log
2018-08-07 00:58:57,309 - maestrowf.datastructures.core.study:_stage_
↪parameterized:640 - INFO - Processing regular dependencies.
2018-08-07 00:58:57,310 - maestrowf.datastructures.core.study:_stage_
↪parameterized:648 - INFO - Adding edge (make-lulesh, run-lulesh_ITER.20.SIZE.10)...
```

Once expansion is complete, Maestro will prompt you to confirm if you'd like to launch the study. Simply confirm with a *y* and hit enter.:

```
$ Would you like to launch the study? [yn] y
```

Maestro will launch a conductor in the background using `nohup` in order to monitor the executing study.

2.2 Monitoring a Running Study

Once the conductor is spun up, you will be returned to the command line prompt. There should now be a `.tests/lulesh` directory within the root of the repository. This directory represents the executing study's workspace, or where Maestro will place this study's data, logs, and state. For a more in-depth description of the contents of a workspace see the documentation about *Study Workspaces*.

In order to check the status of a running study, use the `maestro status` subcommand. The only required parameter to the status command is the path to the running study's workspace. In this case, to find the status of the running study (from the root of the repository) is:

```
$ maestro status ./tests/lulesh
```

The resulting output will look something like below:

```
Step Name                              Workspace             State        Run Time
↪Elapsed Time    Start Time                      Submit Time                  End Time
↪                  Number Restarts
-------------------------------------  --------------------  -----------  --------------
↪--------------  ------------------------------  --------------------------  ------------
↪--------------  ----------------
run-lulesh_ITER.20.SIZE.20             ITER.20.SIZE.20       FINISHED     0:00:00.226297
↪0:00:00.226320  2018-08-07 12:54:23.233567  2018-08-07 12:54:23.233544  2018-08-07
↪12:54:23.459864                  0
post-process-lulesh                    post-process-lulesh   INITIALIZED  --:--:--
↪--:--:--        --                              --                          --
↪                                 0
post-process-lulesh-trials_TRIAL.9     TRIAL.9               INITIALIZED  --:--:--
↪--:--:--        --                              --                          --
↪                                 0
```

(continues on next page)

(continued from previous page)

```
post-process-lulesh-trials_TRIAL.8  TRIAL.8              INITIALIZED  --:--:--
↪--:--:--        --                        --                        --
↪                                  0
post-process-lulesh-size_SIZE.10    SIZE.10              INITIALIZED  --:--:--
↪--:--:--        --                        --                        --
↪                                  0
post-process-lulesh-trials_TRIAL.1  TRIAL.1              INITIALIZED  --:--:--
↪--:--:--        --                        --                        --
↪                                  0
post-process-lulesh-trials_TRIAL.3  TRIAL.3              INITIALIZED  --:--:--
↪--:--:--        --                        --                        --
↪                                  0
post-process-lulesh-trials_TRIAL.2  TRIAL.2              INITIALIZED  --:--:--
↪--:--:--        --                        --                        --
↪                                  0
post-process-lulesh-trials_TRIAL.5  TRIAL.5              INITIALIZED  --:--:--
↪--:--:--        --                        --                        --
↪                                  0
post-process-lulesh-trials_TRIAL.4  TRIAL.4              INITIALIZED  --:--:--
↪--:--:--        --                        --                        --
↪                                  0
post-process-lulesh-trials_TRIAL.7  TRIAL.7              INITIALIZED  --:--:--
↪--:--:--        --                        --                        --
↪                                  0
post-process-lulesh-trials_TRIAL.6  TRIAL.6              INITIALIZED  --:--:--
↪--:--:--        --                        --                        --
↪                                  0
run-lulesh_ITER.30.SIZE.20          ITER.30.SIZE.20      FINISHED     0:00:00.543726
↪0:00:00.543743  2018-08-07 12:54:23.469009  2018-08-07 12:54:23.468992  2018-08-07
↪12:54:24.012735                   0
run-lulesh_ITER.10.SIZE.20          ITER.10.SIZE.20      FINISHED     0:00:00.148773
↪0:00:00.148794  2018-08-07 12:54:23.068119  2018-08-07 12:54:23.068098  2018-08-07
↪12:54:23.216892                   0
post-process-lulesh-size_SIZE.30    SIZE.30              INITIALIZED  --:--:--
↪--:--:--        --                        --                        --
↪                                  0
run-lulesh_ITER.20.SIZE.30          ITER.20.SIZE.30      FINISHED     0:00:01.066736
↪0:00:01.066757  2018-08-07 12:54:24.892856  2018-08-07 12:54:24.892835  2018-08-07
↪12:54:25.959592                   0
run-lulesh_ITER.30.SIZE.10          ITER.30.SIZE.10      FINISHED     0:00:00.054475
↪0:00:00.054488  2018-08-07 12:54:23.005877  2018-08-07 12:54:23.005864  2018-08-07
↪12:54:23.060352                   0
make-lulesh                         make-lulesh          FINISHED     0:00:05.416096
↪0:00:05.416109  2018-08-07 12:53:17.395362  2018-08-07 12:53:17.395349  2018-08-07
↪12:53:22.811458                   0
run-lulesh_ITER.10.SIZE.10          ITER.10.SIZE.10      FINISHED     0:00:00.043584
↪0:00:00.043610  2018-08-07 12:54:22.905328  2018-08-07 12:54:22.905302  2018-08-07
↪12:54:22.948912                   0
run-lulesh_ITER.20.SIZE.10          ITER.20.SIZE.10      FINISHED     0:00:00.035449
↪0:00:00.035463  2018-08-07 12:54:22.958755  2018-08-07 12:54:22.958741  2018-08-07
↪12:54:22.994204                   0
run-lulesh_ITER.10.SIZE.30          ITER.10.SIZE.30      FINISHED     0:00:00.812721
↪0:00:00.812764  2018-08-07 12:54:24.069466  2018-08-07 12:54:24.069423  2018-08-07
↪12:54:24.882187                   0
post-process-lulesh-size_SIZE.20    SIZE.20              INITIALIZED  --:--:--
↪--:--:--        --                        --                        --
↪                                  0
```

(continues on next page)

(continued from previous page)

```
run-lulesh_ITER.30.SIZE.30          ITER.30.SIZE.30      FINISHED     0:00:01.376227
↪0:00:01.376240  2018-08-07 12:54:25.968730  2018-08-07 12:54:25.968717  2018-08-07
↪12:54:27.344957                    0
```

The general statuses that are usually encountered are:

- `INITIALIZED`: A step that has been generated and is awaiting execution.
- `RUNNING`: A step that is currently in progress.
- `FINISHED`: A step that has completed successfully.
- `FAILED`: A step that during execution encountered a non-zero error code.

2.3 Cancelling a Running Study

Similar to checking the status of a running study, cancelling a study uses the `maestro cancel` subcommand with the only required parameter being the path to the study workspace. In the case of the LULESH study, cancel the study using the following command from the root of the repository:

```
$ maestro cancel ./tests/lulesh
```

Note: Cancelling a study is not instantaneous. The background conductor is a daemon which spins up periodically, so cancellation occurs the next time the conductor returns from sleeping and sees that a cancel has been triggered.

When a study is cancelled, the cancellation is reflected in the status when calling the `maestro status` command:

```
Step Name                           Workspace            State      Run Time
↪Elapsed Time    Start Time                  Submit Time                  End Time
↪                  Number Restarts
----------------------------------  -------------------  ---------  --------------  --
↪------------  -------------------------  --------------------------  --------------
↪------------  -----------------
run-lulesh_ITER.20.SIZE.20          ITER.20.SIZE.20      FINISHED   0:00:00.238367
↪0:00:00.238549  2018-08-07 17:24:04.178433  2018-08-07 17:24:04.178251  2018-08-07
↪17:24:04.416800                    0
post-process-lulesh                 post-process-lulesh  CANCELLED  --:--:--        --
↪:--:--        --                          --                          2018-08-07
↪17:25:06.813454                    0
post-process-lulesh-trials_TRIAL.9  TRIAL.9              CANCELLED  --:--:--        --
↪:--:--        --                          --                          2018-08-07
↪17:25:06.813207                    0
post-process-lulesh-trials_TRIAL.8  TRIAL.8              CANCELLED  --:--:--        --
↪:--:--        --                          --                          2018-08-07
↪17:25:06.812957                    0
post-process-lulesh-size_SIZE.10    SIZE.10              CANCELLED  --:--:--        --
↪:--:--        --                          --                          2018-08-07
↪17:25:06.809833                    0
post-process-lulesh-trials_TRIAL.1  TRIAL.1              CANCELLED  --:--:--        --
↪:--:--        --                          --                          2018-08-07
↪17:25:06.810962                    0
post-process-lulesh-trials_TRIAL.3  TRIAL.3              CANCELLED  --:--:--        --
↪:--:--        --                          --                          2018-08-07
↪17:25:06.811659                    0
```

(continues on next page)

(continued from previous page)

```
post-process-lulesh-trials_TRIAL.2  TRIAL.2              CANCELLED  --:--:--        --
↪:--:--        --                          --                          2018-08-07␣
↪17:25:06.811368                  0
post-process-lulesh-trials_TRIAL.5  TRIAL.5              CANCELLED  --:--:--        --
↪:--:--        --                          --                          2018-08-07␣
↪17:25:06.812205                  0
post-process-lulesh-trials_TRIAL.4  TRIAL.4              CANCELLED  --:--:--        --
↪:--:--        --                          --                          2018-08-07␣
↪17:25:06.811927                  0
post-process-lulesh-trials_TRIAL.7  TRIAL.7              CANCELLED  --:--:--        --
↪:--:--        --                          --                          2018-08-07␣
↪17:25:06.812708                  0
post-process-lulesh-trials_TRIAL.6  TRIAL.6              CANCELLED  --:--:--        --
↪:--:--        --                          --                          2018-08-07␣
↪17:25:06.812458                  0
run-lulesh_ITER.30.SIZE.20          ITER.30.SIZE.20      FINISHED   0:00:00.324670 ␣
↪0:00:00.324849  2018-08-07 17:24:04.425894  2018-08-07 17:24:04.425715  2018-08-07␣
↪17:24:04.750564                  0
run-lulesh_ITER.10.SIZE.20          ITER.10.SIZE.20      FINISHED   0:00:00.134795 ␣
↪0:00:00.135016  2018-08-07 17:24:04.032750  2018-08-07 17:24:04.032529  2018-08-07␣
↪17:24:04.167545                  0
post-process-lulesh-size_SIZE.30    SIZE.30              CANCELLED  --:--:--        --
↪:--:--        --                          --                          2018-08-07␣
↪17:25:06.810583                  0
run-lulesh_ITER.20.SIZE.30          ITER.20.SIZE.30      FINISHED   0:00:00.678922 ␣
↪0:00:00.679114  2018-08-07 17:24:05.129377  2018-08-07 17:24:05.129185  2018-08-07␣
↪17:24:05.808299                  0
run-lulesh_ITER.30.SIZE.10          ITER.30.SIZE.10      FINISHED   0:00:00.048609 ␣
↪0:00:00.048803  2018-08-07 17:24:03.974073  2018-08-07 17:24:03.973879  2018-08-07␣
↪17:24:04.022682                  0
make-lulesh                         make-lulesh          FINISHED   0:00:04.979883 ␣
↪0:00:04.980055  2018-08-07 17:22:58.735953  2018-08-07 17:22:58.735781  2018-08-07␣
↪17:23:03.715836                  0
run-lulesh_ITER.10.SIZE.10          ITER.10.SIZE.10      FINISHED   0:00:00.045598 ␣
↪0:00:00.045783  2018-08-07 17:24:03.853461  2018-08-07 17:24:03.853276  2018-08-07␣
↪17:24:03.899059                  0
run-lulesh_ITER.20.SIZE.10          ITER.20.SIZE.10      FINISHED   0:00:00.044422 ␣
↪0:00:00.044655  2018-08-07 17:24:03.912904  2018-08-07 17:24:03.912671  2018-08-07␣
↪17:24:03.957326                  0
run-lulesh_ITER.10.SIZE.30          ITER.10.SIZE.30      FINISHED   0:00:00.359750 ␣
↪0:00:00.359921  2018-08-07 17:24:04.760954  2018-08-07 17:24:04.760783  2018-08-07␣
↪17:24:05.120704                  0
post-process-lulesh-size_SIZE.20    SIZE.20              CANCELLED  --:--:--        --
↪:--:--        --                          --                          2018-08-07␣
↪17:25:06.810216                  0
run-lulesh_ITER.30.SIZE.30          ITER.30.SIZE.30      FINISHED   0:00:00.915474 ␣
↪0:00:00.915682  2018-08-07 17:24:05.818191  2018-08-07 17:24:05.817983  2018-08-07␣
↪17:24:06.733665                  0
```

CHAPTER 3

Basics of Study Construction

Now that you're acquainted with Maestro's interface running a pre-made example, we'll walk you through the basics of making a simple "Hello, World" specification of your own. This page will walk you through the following:

- A single step "Hello World" without parameterization.
- An introduction to a single parameter "Hello World" study.
- An introduction to a multi-parameter "Hello World" study.
- Adding a "farewell" step to "Hello World".

Maestro's default study description uses general YAML notation, which stands for "Yet Another Markup Language" and is a standard data serialization language. For more information on the YAML language, head here to learn more.

3.1 Creating a Single Step Study

To start, we will walk through constructing a single step "Hello World" study that simply echoes "Hello, World!" to a file. The first step is to name your study – in this case we'll settle for something simple and just call our study "Hello World". In your editor of choice, begin by adding the following:

```
description:
    name: hello_world
    description: A simple 'Hello World' study.
```

Note: The *description* block is a required section in every study and has two required keys: name, and description. You may add other keys to the description section, but Maestro will not check for them.

Next we will add the *env* section. This section isn't required, but in this case, we want to stash all study workspaces in a common directory. The *env* section can contain a section named *variables*, which can contain a variable named *OUTPUT_PATH*. Maestro recognizes *OUTPUT_PATH* as a keyword and we can use it to have Maestro create new workspaces for this study in a single place. In this case, we want to create the path *./sample_output/hello_world* to collect all "Hello World" studies. To do that, add the *env* section as follows to the specification:

```
env:
    variables:
        OUTPUT_PATH: ./sample_output/hello_world
```

Note: The *OUTPUT_PATH* variable is a Maestro recognized keyword that specifies the base path where study output is written.

The final section to add will be the *study* section which will only contain a single step. Below the *description* section in the study file you've created add the following block:

```
study:
    - name: hello_world
      description: Say hello to the world!
      run:
          cmd: |
            echo "Hello, World!" > hello_world.txt
```

Note: The - denotes a list item in YAML. To add elements, simply add new elements prefixed with a hyphen. For now, we will keep it simple with one step and will cover adding extra steps later in this guide.

The only required keys for a study step are the name, description, and a run section containing a command (*cmd*). You might notice the similarity in requirement to the study itself of a *name* and *description* entry. This requirement is intentional in order to encourage documentation as you develop a study. The following are descriptions of the required keys:

name A unique name to identify this step by (tip: make this something relevant).

description A human-readable sentence or paragraph describing what this step is meant to achieve.

run A dictionary containing keys that describe what runs in this step.

cmd A string of commands to be executed by this step.

The completed "Hello World" specification should now look like the following:

```
description:
    name: hello_world
    description: A simple 'Hello World' study.

env:
    variables:
        OUTPUT_PATH: ./sample_output/hello_world

study:
    - name: hello_world
      description: Say hello to the world!
      run:
          cmd: |
            echo "Hello, World!" > hello_world.txt
```

Now that the single step "Hello World" study is complete, go ahead and save it to the file *hello_world.yaml*. In order to run the study, simply run the following:

```
$ maestro run hello_world.yaml
```

The command above will produce a timestamped folder that contains the output of the above study. If you'd like to know more about Maestro's command line interface and study output, take a look at our *Quick Start* guide. The "hello_world" study above produces a directory that looks similar to the following:

```
drwxr-xr-x  6 frank  staff   192B Jun 18 11:32 hello_world
-rw-r--r--  1 frank  staff   1.8K Jun 18 11:32 hello_world.pkl
-rw-r--r--  1 frank  staff     0B Jun 18 11:32 hello_world.txt
-rw-r--r--  1 frank  staff   306B Jun 18 11:32 hello_world.yaml
drwxr-xr-x  3 frank  staff    96B Jun 18 11:32 logs
drwxr-xr-x  5 frank  staff   160B Jun 18 11:32 meta
-rw-r--r--  1 frank  staff   241B Jun 18 11:32 status.csv
```

From here, change into the "hello_world" subdirectory. Here you'll see that there are four files: the generated "hello_world.sh" shell script, the resulting output "hello_world.txt", a .out log file, and a .err error log. Your directory should look similar to:

```
-rw-r--r--  1 frank  staff     0B Jun 18 11:32 hello_world.err
-rw-r--r--  1 frank  staff     0B Jun 18 11:32 hello_world.out
-rwxr--r--  1 frank  staff    53B Jun 18 11:32 hello_world.sh
-rw-r--r--  1 frank  staff    14B Jun 18 11:32 hello_world.txt
```

You'll notice that the study directory only contains "hello_world" and the contents for a single run (which corresponds to the singular step above). Maestro detects that the step is not parameterized and uses the workspace that corresponds with the "hello_world" step. If we execute the command *cat hello_world.txt* we see that the output is exactly as specified in the *cmd* portion of the step:

```
$ cat hello_world.txt
$ Hello, World!
```

In the next section we cover the basics of how to add a single parameter to the "Hello World" study.

3.2 Adding a Single Parameter to "Hello World"

Now that you have a functioning single step study, let's expand "Hello World" to greet multiple people. To add this new functionality, that means you need to add a new section called *global.parameters* to our *hello_world.yaml* study specification. So, let's say we want to say hello to Pam, Jim, Michael, and Dwight. The *global.paramters* section would look as follows:

```
global.parameters:
    NAME:
        values: [Pam, Jim, Michael, Dwight]
        label: NAME.%%
```

Note: *%%* is a special token that defines where the value in the label is placed. In this case the parameter labels will be *NAME.Pam*, *NAME.Jim*, and etc. The label can take a custom text format, so long as the *%%* token is included to be able to substitute the parameter's value in the appropriate place.

In order to use the *NAME* parameter, we simply modify the "hello_world" step as follows:

```
study:
    - name: hello_world
      description: Say hello to the world!
      run:
```

(continues on next page)

(continued from previous page)

```
          cmd: |
            echo "Hello, $(NAME)!" > hello_world.txt
```

Note: The *$(NAME)* format is an example of the general format used for variables, parameters, dependency references, and labels. For more examples of referencing values, see the LULESH study in the samples folder in the Maestro GitHub repository.

The full single parameter version of the study specification that says hello to different people is as follows:

```
description:
    name: hello_world
    description: A simple 'Hello World' study.

env:
    variables:
        OUTPUT_PATH: ./sample_output/hello_world

study:
    - name: hello_world
      description: Say hello to someone!
      run:
          cmd: |
            echo "Hello, $(NAME)!" > hello_world.txt

global.parameters:
    NAME:
        values: [Pam, Jim, Michael, Dwight]
        label: NAME.%%
```

If we execute the study and print the contents of the study's workspace, we'll see that the contents are the same as described above. Just as before, if we change into the *hello_world* directory we'll see that the format of the directory has changed. There will now be a set of four directories, one for each parameter value, each containing the *hello_world.txt* output.

```
drwxr-xr-x 6 root root 4096 Mar 25 01:30 ./
drwxr-xr-x 5 root root 4096 Mar 25 01:30 ../
drwxr-xr-x 2 root root 4096 Mar 25 01:30 NAME.Dwight/
drwxr-xr-x 2 root root 4096 Mar 25 01:30 NAME.Jim/
drwxr-xr-x 2 root root 4096 Mar 25 01:30 NAME.Michael/
drwxr-xr-x 2 root root 4096 Mar 25 01:30 NAME.Pam/
```

However, if we *cat* each of the outputs from each directory, we'll see that the value for *$(NAME)* has been substituted:

```
$ cat */hello_world.txt
$ Hello, Dwight!
$ Hello, Jim!
$ Hello, Michael!
$ Hello, Pam!
```

3.3 Expanding "Hello World" to Multiple Steps

Now that we've got our specification set up to say hello to multiple people, let's take a step back and look at our base "Hello World" specification and add "bye_world" as specified below:

```
description:
    name: hello_world
    description: A simple 'Hello World' study.

env:
    variables:
        OUTPUT_PATH: ./sample_output/hello_world

study:
    - name: hello_world
      description: Say hello to the world!
      run:
          cmd: |
            echo "Hello, World!" > hello_world.txt

    - name: bye_world
      description: Say bye to someone!
      run:
          cmd: |
            echo "Bye, World!" > bye_world.txt
          depends: [hello_world]
```

After adding this step to your specification, go ahead and run it using *maestro run* as before. Now, if you look at the generated study directory, we see that the study generates an extra directory for the "bye_world" step.

```
drwxr-xr-x  6 frank  staff   192B Jun 25 20:54 bye_world
-rw-r--r--  1 frank  staff   2.3K Jun 25 20:54 hello_bye.pkl
-rw-r--r--  1 frank  staff     0B Jun 25 20:53 hello_bye.txt
-rw-r--r--  1 frank  staff   551B Jun 25 20:53 hello_bye_world.yaml
drwxr-xr-x  6 frank  staff   192B Jun 25 20:53 hello_world
drwxr-xr-x  3 frank  staff    96B Jun 25 20:54 logs
drwxr-xr-x  5 frank  staff   160B Jun 25 20:53 meta
-rw-r--r--  1 frank  staff   383B Jun 25 20:54 status.csv
```

If you change into this directory, you'll see that a similar set of files to the previous "hello_world" step have been created. You'll see that executing *cat bye_world.txt* prints out "Bye, World!". Now, to take this a step further – what if we wanted to say bye to each particular person in our parameterized "hello world" example?

Now, if we start with our parameterized hello world specification, we add the "bye_world" step and make it dependent on the "hello_world" step. You should also update the description and study name to something meaningful for the new study.

The study workspace looks the same as the "hello_bye_world" study specified above at the top level; however, like the multi-parameterized "hello_world" study you'll see that each step's workspaces have parameterized folders. The "hello_world" step has the same workspace set up as the previous parameterized study as expected.

If you look into the "bye_world" workspace, you'll also notice it has the same exact set of folders as "hello_world". While this set up might seem weird at first, it is a feature of how Maestro expands the study using parameters. In a later section, we'll describe how Maestro expands the study in a predictable manner – but for now, it is enough to know that the "bye_world" step was expanded in a 1:1 fashion because the step is dependent on "hello_world" and the parameters it used. Maestro, in this case, can not make any assumptions and simply expands the "bye_world" one to one with each parameterized "hello_world".

Note: You can view the sample specifications constructed here in their entirety in Maestro's GitHub repository here.

CHAPTER 4

LULESH Specification Breakdown

Stub

CHAPTER 5

Specifying Study Parameters

Maestro supports parameterization as a means of iterating over steps in a study with varying information. Maestro uses token replacement to define variables in the study specification to be replaced when executing the study. Token replacement can be used in various contexts in Maestro; however Maestro implements specific features for managing parameters.

Maestro makes no assumptions about how parameters are defined or used in a study, enabling flexibility in regards to how a study may be configured with parameters.

There are two ways Maestro supports parameters:

- Directly in the study specification as the `global.parameters` block
- Through the use of a user created Python module called *Parameter Generator (pgen)*

5.1 Maestro Parameter Block

The quickest and easiest way to setup parameters in a Maestro study is by defining a `global.parameters` block directly in the specification

```
global.parameters:
 TRIAL:
     values  : [1, 2, 3, 4, 5, 6, 7, 8, 9]
     label   : TRIAL.%%
 SIZE:
     values  : [10, 10, 10, 20, 20, 20, 30, 30, 30]
     label   : SIZE.%%
 ITERATIONS:
     values  : [10, 20, 30, 10, 20, 30, 10, 20, 30]
     label   : ITER.%%
```

The above example defines the parameters `TRIAL`, `SIZE`, and `ITERATIONS`. Parameters can be used in study steps to vary information. When a parameter is defined in a study, Maestro will automatically detect the usage of a parameter moniker and handle the substitution automatically in the study expansion. This ensures that each set of parameters are run as part of the study.

The `label` key in the block specifies the pattern to use for the directory name when the workspace is created. By default, Maestro constructs a unique workspace for each parameter combination.

Defining multiple parameters in the parameter block will share a 1:1 mapping. Maestro requires all combinations be resolved when using the parameter block. The combinations in the above example will be expanded as follows:

- TRIAL.1.SIZE.10.ITERATIONS.10
- TRIAL.2.SIZE.10.ITERATIONS.20
- TRIAL.3.SIZE.10.ITERATIONS.30
- ...

Maestro does not do any additional operations on parameters such as cross products. If more complex methodologies are required to define parameters then the use of Maestro's *Parameter Generator (pgen)* is recommended.

Defined parameters can be used in steps directly:

```
- name: run-lulesh
  description: Run LULESH.
  run:
    cmd: |
        $(LULESH)/lulesh2.0 -s $(SIZE) -i $(ITERATIONS) -p > $(outfile)
    depends: [make-lulesh]
```

Even though this is defined in Maestro as a single step, Maestro will automatically run this step with each parameter combinations. This makes it very easy to setup studies and apply parameters to be run.

Note: Maestro will only use parameters if they've been defined in at least one step

---

In addition to direct access to parameter values, a parameter label can be used in steps by appending the `.label` moniker to the name (as seen below with `$(ITERATIONS.label)`):

```
- name: run-lulesh
  description: Run LULESH.
  run:
    cmd: |
        echo "Running case: $(SIZE.label), $(ITERATIONS.label)"
        $(LULESH)/lulesh2.0 -s $(SIZE) -i $(ITERATIONS) -p > $(outfile)
    depends: [make-lulesh]
```

5.2 What can be Parameterized in Maestro?

A common use case for Maestro is to use the parameter block to specify values to iterate over for a simulation parameter study; however, Maestro does not make any assumptions about what these values are. This makes the use of Maestro's parameter block very flexible. For example, Maestro does not require the parameter variations to be numeric.

Listing 1: study.yaml

```
study:
 - name: run-simulation
   description: Run a simulation.
   run:
     cmd: |
```

(continues on next page)

(continued from previous page)

```
          $(CODE) -in $(SPECROOT)/$(INPUT)
   depends: []

global.parameters:
    INPUT:
        values  : [input1.in, input2.in, input3.in]
        label   : INPUT.%%
```

The above example highlights a partial study spec that defines a parameter block of simulation inputs that will be varied when the study runs. The `run-simulation` step will run three times, once for each defined input file.

Listing 2: study.yaml

```
study:
    - name: run-simulation
      description: Run a simulation.
      run:
        cmd: |
            $(CODE_PATH)/$(VERSION)/code.exe -in $(SPECROOT)/$(INPUT)
      depends: []

global.parameters:
    INPUT:
        values  : [input1.in, input2.in, input3.in, input1.in, input2.in, input3.in]
        label   : INPUT.%%
    VERSION:
        values  : [4.0.0, 4.0.0, 4.0.0, 5.0.0, 5.0.0, 5.0.0]
        label   : VERSION.%%
```

This example parameterizes the inputs and the version of the code being run. Maestro will run each input with the different code version. The above example assumes that all the code versions share a base path, `$(CODE_PATH)` which is inserted via the token replacment mechanism to yeild the full paths (e.g. /usr/gapps/code/4.0.0/code.exe).

5.3 Where can Parameters be used in Study Steps?

Maestro uses monikers to reference parameters in study steps, and will automatically perform token replacement on used parameters when the study is run. The page Maestro Token Replacement goes into detail about how token replacement works in Maestro.

Maestro is very flexible in the way it manages token replacement for parameters and as such tokens can be used in a variety of ways in a study.

5.3.1 Cmd block

Parameters can be defined in the Maestro `cmd` block in the study step. Everything in Maestro's `cmd` block will be written to a bash shell or batch script (if batch is configured). Any shell commands should be valid in the `cmd` block. A common way to use parameters is to pass them in via arguments to a code, script, or tool.

Listing 3: study.yaml

```
...

- name: run-simulation
```

(continues on next page)

(continued from previous page)

```
  description: Run a simulation.
    run:
        cmd: |
            /usr/gapps/code/bin/code -in input.in -def param $(PARAM)
        depends: []

...
```

The specific syntax for using a parameter with a specific code, script, or tool will depend on how the application supports command line arguments.

5.3.2 Batch Configuration Keys

Step based batch configurations can also be parameterized in Maestro. This provides an easy way to configure scaling studies or to manage studies where batch settings are dependent on the parameter values.

Listing 4: study.yaml

```
study:
    - name: run-simulation
      description: Run a simulation.
      run:
        cmd: |
            $(CODE_PATH)/$(VERSION)/code.exe -in input.in -def RES $(RES)
        procs: $(PROC)
        nodes: $(NODE)
        walltime: $(WALLTIME)
        depends: []

global.parameters:
    RES:
        values  : [2, 4, 6, 8]
        label   : RES.%%
    PROC:
        values  : [8, 8, 16, 32]
        label   : PROC.%%
    NODE:
        values  : [1, 1, 2, 4]
        label   : NODE.%%
    WALLTIME:
        values  : ["00:10:00", "00:15:00", "00:30:00", "01:00:00"]
        label   : PROC.%%
```

5.4 Parameter Generator (pgen)

Maestro's Parameter Generator (**pgen**) supports setting up more flexible and complex parameter generation. Maestro's pgen is a user supplied python file that contains the parameter generation logic, overriding the `global.parameters` block in the yaml specification file. To run a Maestro study using a parameter generator just pass in the path to the **pgen** file to Maestro on the command line when launching the study, such as this example where the study specification file and **pgen** file live in the same directory:

```
$ maestro run study.yaml --pgen pgen.py
```

The minimum requirements for making a valid pgen file is to make a function called `get_custom_generator()` which returns a Maestro *`ParameterGenerator`* object as demonstrated in the simple example below:

```
from maestrowf.datastructures.core import ParameterGenerator

def get_custom_generator(env, **kwargs):
    p_gen = ParameterGenerator()
    params = {
        "COUNT": {
            "values": [i for i in range(1, 10)],
            "label": "COUNT.%%"
        },
    }

    for key, value in params.items():
        p_gen.add_parameter(key, value["values"], value["label"])

    return p_gen
```

The object simply builds the same nested key:value pairs seen in the `global.parameters` block available in the yaml specification.

For this simple example above, this may not offer compelling advantages over writing out the flattened list in the yaml specification directly. This programmatic approach becomes preferable when expanding studies to use hundreds of parameters and parameter values or requiring non-trivial parameter value distributions. The following examples will demonstrate these scenarios using both standard python library tools and additional 3rd party packages from the larger python ecosystem.

EXAMPLE: Using Python's standard itertools package to perform a Cartesian Product of parameters in the lulesh example specification.

Listing 5: lulesh_itertools_pgen.py

```
from maestrowf.datastructures.core import ParameterGenerator
import itertools as iter


def get_custom_generator(env, **kwargs):
    """
    Create a custom populated ParameterGenerator.
    This function recreates the exact same parameter set as the sample LULESH
    specifications. The difference here is that itertools is employed to
    programatically generate the samples instead of manually writing out
    all of the combinations.
    :params env: A StudyEnvironment object containing custom information.
    :params kwargs: A dictionary of keyword arguments this function uses.
    :returns: A ParameterGenerator populated with parameters.
    """
    p_gen = ParameterGenerator()

    sizes = (10, 20, 30)
    iterations = (10, 20, 30)

    size_values = []
    iteration_values = []
    trial_values = []

    for trial, param_combo in enumerate(iter.product(sizes, iterations)):
```

(continues on next page)

(continued from previous page)

```
        size_values.append(param_combo[0])
        iteration_values.append(param_combo[1])
        trial_values.append(trial)

    params = {
        "TRIAL": {
            "values": trial_values,
            "label": "TRIAL.%%"
        },
        "SIZE": {
            "values": size_values,
            "label": "SIZE.%%"
        },
        "ITER": {
            "values": iteration_values,
            "label": "ITER.%%"
        },
    }

    for key, value in params.items():
        p_gen.add_parameter(key, value["values"], value["label"])

    return p_gen
```

This results in the following set of parameters, matching the lulesh sample workflow:

Table 1: Sample parameters from itertools_pgen.py

Parameter	Values								
TRIAL	0	1	2	3	4	5	6	7	8
SIZE	10	10	10	20	20	20	30	30	30
ITER	10	20	30	10	20	30	10	20	30

5.4.1 Pgen Arguments (pargs)

There is an additional *pgen* feature that can be used to make them more dynamic. The above example generates a fixed set of parameters, requiring editing the *lulesh_itertools_pgen.py* file to change that. Maestro supports passing arguments to these generator functions on the command line:

```
$ maestro run study.yaml --pgen itertools_pgen_pargs.py --parg "SIZE_MIN:10" --parg
↪"SIZE_STEP:10" --parg "NUM_SIZES:4"
```

Each argument is a string in `key:value` form, which can be accessed in the parameter generator function as shown below:

Listing 6: itertools_pgen_pargs.py

```
from maestrowf.datastructures.core import ParameterGenerator
import itertools as iter

def get_custom_generator(env, **kwargs):
    p_gen = ParameterGenerator()

    # Unpack any pargs passed in
    size_min = int(kwargs.get('SIZE_MIN', '10'))
```

(continues on next page)

(continued from previous page)

```
    size_step = int(kwargs.get('SIZE_STEP', '10'))
    num_sizes = int(kwargs.get('NUM_SIZES', '3'))

    sizes = range(size_min, size_min+num_sizes*size_step, size_step)
    iterations = (10, 20, 30)

    size_values = []
    iteration_values = []
    trial_values = []

    for trial, param_combo in enumerate(iter.product(sizes, iterations)):
        size_values.append(param_combo[0])
        iteration_values.append(param_combo[1])
        trial_values.append(trial)

    params = {
        "TRIAL": {
            "values": trial_values,
            "label": "TRIAL.%%"
        },
        "SIZE": {
            "values": size_values,
            "label": "SIZE.%%"
        },
        "ITER": {
            "values": iteration_values,
            "label": "ITER.%%"
        },
    }

    for key, value in params.items():
        p_gen.add_parameter(key, value["values"], value["label"])

    return p_gen
```

Passing the **pargs** `SIZE_MIN:10', 'SIZE_STEP:10', and 'NUM_SIZES:4' then yields the expanded parameter set:

Table 2: Sample parameters from itertools_pgen_pargs.py

Parameter	Values											
TRIAL	0	1	2	3	4	5	6	7	8	9	10	11
SIZE	10	10	10	20	20	20	30	30	30	40	40	40
ITER	10	20	30	10	20	30	10	20	30	10	20	30

Notice that using the pgen input method makes it trivially easy to add 1000's of parameters, something which would be cumbersome via manual editing of the `global.parameters` block in the study specification file.

There are no requirements to cram all of the logic into the `get_custom_generator()` function. The next example demonstrates using 3rd party libraries and breaking out the actual parameter generation algorithm into separate helper functions that the `get_custom_generator()` function uses to get some more complicated distributions. The only concerns with this approach will be to ensure the library is installed in the same virtual environment as the Maestro executable you are using. The simple parameter distribution demoed in here is often encountered in polynomial interpolation applications and is designed to suppress the Runge phenomena by sampling the function to be interpolated at the Chebyshev nodes.

EXAMPLE: Using numpy to calculate a sampling of a function at the Chebyshev nodes.

Listing 7: np_cheb_pgen_pargs.py

```
from maestrowf.datastructures.core import ParameterGenerator
import numpy as np


def chebyshev_dist(var_range, num_pts):
    """
    Helper function for generating Chebyshev points in a specified range.

    :params var_range: Length 2 list or tuple defining the value range
    :params num_pts: Integer number of points to generate
    :returns: ndarrays of the Chebyshev x points, and the corresponding y
              values of the circular mapping
    """
    r = 0.5*(var_range[1] - var_range[0])

    angles = np.linspace(np.pi, 0.0, num_pts)
    xpts = r*np.cos(angles) + r
    ypts = r*np.sin(angles)

    return xpts, ypts


def get_custom_generator(env, **kwargs):
    """
    Create a custom populated ParameterGenerator.
    This function generates a 1D distribution of points for a single variable,
    using the Chebyshev points scaled to the requested range.
    The point of this file is to present an example of using external libraries
    and helper functions to generate parameter value distributions.  This
    technique can be used to build reusable/modular sampling libraries that
    pgen can hook into.

    :params env: A StudyEnvironment object containing custom information.
    :params kwargs: A dictionary of keyword arguments this function uses.
    :returns: A ParameterGenerator populated with parameters.
    """
    p_gen = ParameterGenerator()

    # Unpack any pargs passed in
    x_min = int(kwargs.get('X_MIN', '0'))
    x_max = int(kwargs.get('X_MAX', '1'))
    num_pts = int(kwargs.get('NUM_PTS', '10'))

    x_pts, ypts = chebyshev_dist([x_min, x_max], num_pts)

    params = {
        "X": {
            "values": list(x_pts),
            "label": "X.%%"
        },
    }

    for key, value in params.items():
        p_gen.add_parameter(key, value["values"], value["label"])

```

(continues on next page)

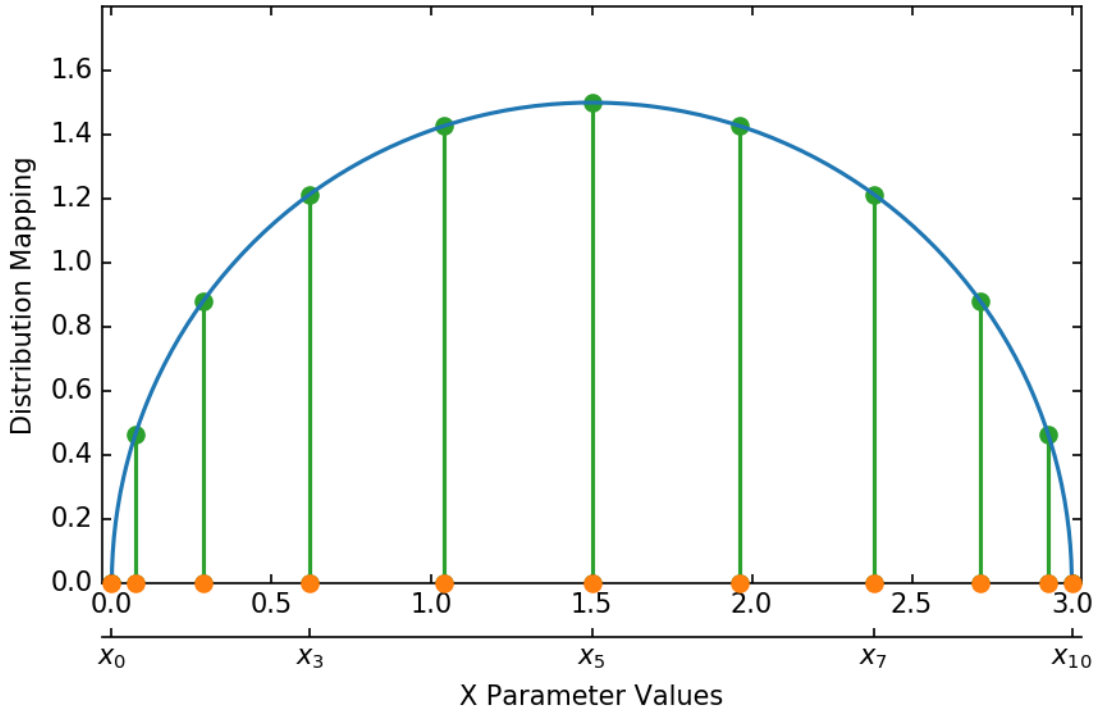
(continued from previous page)

```
56      return p_gen
```

Running this parameter generator with the following pargs

```
$ maestro run study.yaml --pgen np_cheb_pgen.py --parg "X_MIN:0" --parg "X_MAX:3" --
↪parg "NUM_PTS:11"
```

results in the 1D distribution of points for the `X` parameter shown by the orange circles:



5.4.2 Referencing Values from a Specification's Env Block

In addition to command line arguments via *pargs*, the variables defined in the `env` block in the workflow specification file can be accessed inside the *`ParameterGenerator`* objects, which is passed in to `get_custom_generator()` as the first argument. The lulesh sample specification can be extended to store the default values for the *pgen*, enhancing the reproducability of the generator. The following example makes use of an optional `seed` parameter which can be added to the `env` block or set via *pargs* to make a repeatable study specification, while omitting it can enable fully randomized workflows upon every instantiation. The variables are accessed via the *`StudyEnvironment`*'s *`find()`* function, which will return `None` if the variable is not defined in the study specification.

Listing 8: lulesh_montecarlo_args.py

```
1  """An example file that produces a custom parameters for the LULESH example."""
2
3  from random import randint, seed
4
5  from maestrowf.datastructures.core import ParameterGenerator
```

(continues on next page)

(continued from previous page)

```


def get_custom_generator(env, **kwargs):
    """
    Create a custom populated ParameterGenerator.
    This function adapts the LULESH custom generator to randomly generate
    values for the SIZE parameter within a prescribed range.  An optional
    seed is included, which if present on the command line or in the spec's
    env block will allow reproducible random values.

    :params env: A StudyEnvironment object containing custom information.
    :params kwargs: A dictionary of keyword arguments this function uses.
    :returns: A ParameterGenerator populated with parameters.
    """
    p_gen = ParameterGenerator()
    trials = int(kwargs.get("trials", env.find("TRIALS").value))
    size_min = int(kwargs.get("smin", env.find("SMIN").value))
    size_max = int(kwargs.get("smax", env.find("SMAX").value))
    iterations = int(kwargs.get("iter", env.find("ITER").value))
    r_seed = kwargs.get("seed", env.find("SEED").value)

    seed(a=r_seed)

    params = {
        "TRIAL": {
            "values": [i for i in range(1, trials)],
            "label": "TRIAL.%%"
        },
        "SIZE": {
            "values": [randint(size_min, size_max) for i in range(1, trials)],
            "label": "SIZE.%%"
        },
        "ITERATIONS": {
            "values": [iterations for i in range(1, trials)],
            "label": "ITERATIONS.%%"
        }
    }

    for key, value in params.items():
        p_gen.add_parameter(key, value["values"], value["label"])

    return p_gen
```

CHAPTER 6

Maestro Core Concepts

Stub

CHAPTER 7

Maestro Workflow Conductor

7.1 maestrowf package

The core module contains all core abstracts and classes.

All core abstracts and implementations for core concept classes (Study, Environment, Parameter generation, etc.). This module also includes interface abstracts, base class abstracts, and general utilities.

7.1.1 Subpackages

7.1.1.1 maestrowf.abstracts package

The core abstract APIs that define various class behaviors.

This module contains all of the abstract classes and APIs for defining objects. Abstracts include abstract data stuctures (like a graph), APIs for concepts such as queueing adapters and environment APIs, as well as fundamental data structures.

class maestrowf.abstracts.**abstractclassmethod**(*callable*)

Bases: classmethod

Python 2.7 does not include built in @abstractclassmethod so we create our own class that extends @classmethod and then attachs a __isabstractmethod variable to the callable function. See ref: https://stackoverflow.com/a/11218474

class maestrowf.abstracts.**Dependency**

Bases: *maestrowf.abstracts.envobject.Substitution*

Abstract object representing a dependency.

The Dependency base class is intended to be used to capture external items the workflow is dependent on. These items include (but are not limited to):

- Remotely stored repositories (such as bitbucket)
- Paths located on the filesystem that hold required files or binaries

- Binaries that are required to be installed using a specific package manager
- External APIs that a workflow needs to pull data from

The goal of this base class is to make it so that this package is able to pull exernal dependencies in a consistent manner.

acquire(*substitutions=None*)
Acquire the dependency as specfied by the class instance.

Subclasses that implement this interface should raise exceptions during acquisition should they be unable to retrieve their specified dependency. It is assumed that if acquiring throws an exception that the study cannot proceed forward.

Parameters **substitutions** – List of Substitution objects that can be applied.

class maestrowf.abstracts.**Graph**
Bases: object

An abstract graph data structure.

add_edge(*src*, *dest*)
Add the edge (src, dest) to the graph.

Parameters

- **src** – Source vertex name.
- **dest** – Destination vertex name.

add_node(*name*, *obj*)
Method to add a node to the graph.

Parameters

- **name** – String identifier of the node.
- **obj** – An object representing the value of the node.

remove_edge(*src*, *dest*)
Remove edge (src, dest) from the graph.

Parameters

- **src** – Source vertex name.
- **dest** – Destination vertex name.

class maestrowf.abstracts.**PickleInterface**
Bases: object

A mixin class that implements a general pickle interface using dill.

pickle(*path*)
Generate a pickle file of of a class instance.

Parameters **path** – The path to write the pickle to.

classmethod unpickle(*path*)
Load a pickled instance from a pickle file.

Parameters **path** – Path to a pickle file containing a class instance.

class maestrowf.abstracts.**Singleton**
Bases: maestrowf.abstracts.SingletonMeta

Single type to allow for classes to be typed as a singleton.

class maestrowf.abstracts.**Source**
Bases: *maestrowf.abstracts.envobject.EnvObject*

Abstract class representing classes that alter environment sourcing.

WARNING: The API for this class is still in development. The Source environment class is meant to provide a way to programmatically set environment settings that binaries or other scripts may require in the workflow. Such settings that are intended to be captured are:

- Exporting of shell/environment variables (using 'export')
- Setting of an environment package with the 'use' command

apply(*data*)
Apply the Source to some string data.

Subclasses of Source should use this method in order to apply an environment alterating change. The 'data' parameter should be a string representing a command to apply Source to or a list of other comands that Source should be included with.

Parameters **data** – A string representing a command or set of other sources.

Returns A string with the Source applied.

class maestrowf.abstracts.**Specification**
Bases: object

Abstract class for loading and verifying a Study Specification

desc
Getter for the description of a study specification.

Returns A string containing the description of the study specification.

get_parameters()
Generate a ParameterGenerator object from the global parameters.

Returns A ParameterGenerator with data from the specification.

get_study_environment()
Generate a StudyEnvironment object from the environment in the spec.

Returns A StudyEnvironment object with the data in the specification.

get_study_steps()
Generate a list of StudySteps from the study in the specification.

Returns A list of StudyStep objects.

classmethod load_specification(*path*)
Method for loading a study specification from a file.

Parameters **path** – Path to a study specification.

Returns A specification object containing the information loaded from path.

classmethod load_specification_from_stream(*stream*)
Method for loading a study specification from a stream.

Parameters **stream** – Raw text stream containing specification data.

Returns A specification object containing the information in string.

name
Getter for the name of a study specification.

Returns The name of the study described by the specification.

output_path
Return the OUTPUT_PATH variable (if it exists).

Returns Returns OUTPUT_PATH if it exists, empty string otherwise.

verify()
Verify the whole specification.

class maestrowf.abstracts.**Substitution**
Bases: *maestrowf.abstracts.envobject.EnvObject*

Abstract class representing classes that perform value replacements.

substitute(*data*)
Perform a replacement of some substring into data.

The method takes the input string data and performs a replacement. This API is used to represent concepts such as variables or parameters that would want to be replaced within the string data.

Parameters **data** – A string to perform a replacement on.

Returns A string equal to the original string data with substitutions made (if any were performed).

Subpackages

maestrowf.abstracts.enums package

Package for providing enumerations for interfaces

class maestrowf.abstracts.enums.**JobStatusCode**
Bases: enum.Enum

An enumeration.

ERROR = 2

NOJOBS = 1

OK = 0

class maestrowf.abstracts.enums.**State**
Bases: enum.Enum

Workflow step state enumeration.

CANCELLED = 12

DRYRUN = 13

FAILED = 7

FINISHED = 5

FINISHING = 4

HWFAILURE = 9

INCOMPLETE = 8

INITIALIZED = 0

PENDING = 1

QUEUED = 6

RUNNING = 3

TIMEDOUT = 10

UNKNOWN = 11

WAITING = 2

class maestrowf.abstracts.enums.**SubmissionCode**
Bases: enum.Enum

An enumeration.

ERROR = 1

OK = 0

class maestrowf.abstracts.enums.**StudyStatus**
Bases: enum.Enum

Workflow status enumeration

CANCELLED = 3

FAILURE = 2

FINISHED = 0

RUNNING = 1

maestrowf.abstracts.interfaces package

Abstract classes for handling interfacing with various services.

class maestrowf.abstracts.interfaces.**SchedulerScriptAdapter**(***kwargs*)
Bases: *maestrowf.abstracts.interfaces.scriptadapter.ScriptAdapter*

Abstract class representing the interface for scheduling scripts.

This class handles both the construction of scripts (as required by the ScriptAdapter base class) but also includes the necessary methods for constructing parallel commands. The adapter will substitute parallelized commands but also defines how to schedule and check job status.

add_batch_parameter(*name*, *value*)
Add a parameter to the ScriptAdapter instance.

Parameters

- **name** – String name of the parameter that's being added.
- **value** – Value associated with the parameter name (should have a str method).

get_header(*step*)
Generate the header present at the top of execution scripts.

Parameters **step** – A StudyStep instance.

Returns A string of the header based on internal batch parameters and the parameter step.

get_parallelize_command(*procs*, *nodes*, ***kwargs*)
Generate the parallelization segement of the command line.

Parameters

- **procs** – Number of processors to allocate to the parallel call.

- **nodes** – Number of nodes to allocate to the parallel call (default = 1).

Returns A string of the parallelize command configured using nodes and procs.

get_scheduler_command(*step*)
Generate the full parallelized command for use in a batch script.

Parameters **step** – A StudyStep instance.

Returns 1. A Boolean value - True if command is to be scheduled, False otherwise. 2. A string representing the parallelized batch command for the specified step command. 3. A string representing the parallelized batch command for the specified step restart command.

```
launcher_regex = re.compile('\\$\\(LAUNCHER\\)\\[(?P<alloc>.*)\\]')

launcher_var = '$(LAUNCHER)'

legacy_alloc = '(?P<nodes>[0-9]+),\\s*(?P<procs>[0-9]+)'

node_alloc = '(?P<nodes>[0-9]+)n'

task_alloc = '(?P<procs>[0-9]+)p'
```

class maestrowf.abstracts.interfaces.**ScriptAdapter**(***kwargs*)
Bases: object

Abstract class representing the interface for constructing scripts.

The ScriptAdapter abstract class is meant to provide a consistent high level interface to generate scripts automatically based on an ExecutionDAG. Adapters as a whole should only interface with the ExecutionDAG because it is ultimately the DAG that manages the state of tasks. Adapters attempt to bridge the 'how' in an abstract way such that the interface is refined to methods such as: - Generating a script with the proper syntax to submit. - Submitting a script using the proper command. - Checking job status.

cancel_jobs(*joblist*)
For the given job list, cancel each job.

Parameters **joblist** – A list of job identifiers to be cancelled.

Returns The return code to indicate if jobs were cancelled.

check_jobs(*joblist*)
For the given job list, query execution status.

Parameters **joblist** – A list of job identifiers to be queried.

Returns The return code of the status query, and a dictionary of job identifiers to their status.

key
Return the key name for a ScriptAdapter..

This is used to register the adapter in the ScriptAdapterFactory and when writing the workflow specification.

submit(*step*, *path*, *cwd*, *job_map=None*, *env=None*)
Submit a script to the scheduler.

If cwd is specified, the submit method will operate outside of the path specified by the 'cwd' parameter. If env is specified, the submit method will set the environment variables for submission to the specified values. The 'env' parameter should be a dictionary of environment variables.

Parameters

- **step** – An instance of a StudyStep.
- **path** – Path to the script to be executed.

- **cwd** – Path to the current working directory.
- **job_map** – A map of workflow step names to their job identifiers.
- **env** – A dict containing a modified environment for execution.

Returns The return code of the submission command and job identiifer.

write_script(*ws_path*, *step*)
Generate the script for the specified StudyStep.

Parameters

- **ws_path** – Workspace path for the step.
- **step** – An instance of a StudyStep class.

Returns A tuple containing a boolean set to True if step should be scheduled (False otherwise), path to the generate script, and path to the generated restart script (None if step cannot be restarted).

Submodules

maestrowf.abstracts.interfaces.schedulerscriptadapter module

Abstract Cluster Interfaces defining the API for interacting with queues.

class maestrowf.abstracts.interfaces.schedulerscriptadapter.**SchedulerScriptAdapter**(***kwargs*)
Bases: *maestrowf.abstracts.interfaces.scriptadapter.ScriptAdapter*

Abstract class representing the interface for scheduling scripts.

This class handles both the construction of scripts (as required by the ScriptAdapter base class) but also includes the necessary methods for constructing parallel commands. The adapter will substitute parallelized commands but also defines how to schedule and check job status.

add_batch_parameter(*name*, *value*)
Add a parameter to the ScriptAdapter instance.

Parameters

- **name** – String name of the parameter that's being added.
- **value** – Value associated with the parameter name (should have a str method).

get_header(*step*)
Generate the header present at the top of execution scripts.

Parameters **step** – A StudyStep instance.

Returns A string of the header based on internal batch parameters and the parameter step.

get_parallelize_command(*procs*, *nodes*, ***kwargs*)
Generate the parallelization segement of the command line.

Parameters

- **procs** – Number of processors to allocate to the parallel call.
- **nodes** – Number of nodes to allocate to the parallel call (default = 1).

Returns A string of the parallelize command configured using nodes and procs.

get_scheduler_command(*step*)
Generate the full parallelized command for use in a batch script.

Parameters **step** – A StudyStep instance.

Returns 1. A Boolean value - True if command is to be scheduled, False otherwise. 2. A string representing the parallelized batch command for the specified step command. 3. A string representing the parallelized batch command for the specified step restart command.

```
launcher_regex = re.compile('\\$\\(LAUNCHER\\)\\[(?P<alloc>.*)\\]')
launcher_var = '$(LAUNCHER)'
legacy_alloc = '(?P<nodes>[0-9]+),\\s*(?P<procs>[0-9]+)'
node_alloc = '(?P<nodes>[0-9]+)n'
task_alloc = '(?P<procs>[0-9]+)p'
```

maestrowf.abstracts.interfaces.scriptadapter module

Abstract Script Interfaces for generating scripts.

class maestrowf.abstracts.interfaces.scriptadapter.**ScriptAdapter**(***kwargs*)
Bases: object

Abstract class representing the interface for constructing scripts.

The ScriptAdapter abstract class is meant to provide a consistent high level interface to generate scripts automatically based on an ExecutionDAG. Adapters as a whole should only interface with the ExecutionDAG because it is ultimately the DAG that manages the state of tasks. Adapters attempt to bridge the 'how' in an abstract way such that the interface is refined to methods such as: - Generating a script with the proper syntax to submit. - Submitting a script using the proper command. - Checking job status.

cancel_jobs(*joblist*)
For the given job list, cancel each job.

Parameters **joblist** – A list of job identifiers to be cancelled.

Returns The return code to indicate if jobs were cancelled.

check_jobs(*joblist*)
For the given job list, query execution status.

Parameters **joblist** – A list of job identifiers to be queried.

Returns The return code of the status query, and a dictionary of job identifiers to their status.

key
Return the key name for a ScriptAdapter..

This is used to register the adapter in the ScriptAdapterFactory and when writing the workflow specification.

submit(*step*, *path*, *cwd*, *job_map=None*, *env=None*)
Submit a script to the scheduler.

If cwd is specified, the submit method will operate outside of the path specified by the 'cwd' parameter. If env is specified, the submit method will set the environment variables for submission to the specified values. The 'env' parameter should be a dictionary of environment variables.

Parameters

- **step** – An instance of a StudyStep.
- **path** – Path to the script to be executed.

- **cwd** – Path to the current working directory.
- **job_map** – A map of workflow step names to their job identifiers.
- **env** – A dict containing a modified environment for execution.

Returns The return code of the submission command and job identiifer.

write_script(*ws_path*, *step*)
Generate the script for the specified StudyStep.

Parameters

- **ws_path** – Workspace path for the step.
- **step** – An instance of a StudyStep class.

Returns A tuple containing a boolean set to True if step should be scheduled (False otherwise), path to the generate script, and path to the generated restart script (None if step cannot be restarted).

Submodules

maestrowf.abstracts.abstractclassmethod module

Implementation of a new abstractclassmethod property.

class maestrowf.abstracts.abstractclassmethod.**abstractclassmethod**(*callable*)
Bases: classmethod

Python 2.7 does not include built in @abstractclassmethod so we create our own class that extends @classmethod and then attachs a __isabstractmethod variable to the callable function. See ref: https://stackoverflow.com/a/11218474

maestrowf.abstracts.envobject module

The collection of basic classes that can be used for an environment.

class maestrowf.abstracts.envobject.**Dependency**
Bases: *maestrowf.abstracts.envobject.Substitution*

Abstract object representing a dependency.

The Dependency base class is intended to be used to capture external items the workflow is dependent on. These items include (but are not limited to):

- Remotely stored repositories (such as bitbucket)
- Paths located on the filesystem that hold required files or binaries
- Binaries that are required to be installed using a specific package manager
- External APIs that a workflow needs to pull data from

The goal of this base class is to make it so that this package is able to pull exernal dependencies in a consistent manner.

acquire(*substitutions=None*)
Acquire the dependency as specfied by the class instance.

Subclasses that implement this interface should raise exceptions during acquisition should they be unable to retrieve their specified dependency. It is assumed that if acquiring throws an exception that the study cannot proceed forward.

Parameters **substitutions** – List of Substitution objects that can be applied.

class maestrowf.abstracts.envobject.**EnvObject**
Bases: object

An abstract class representing objects that exist in a study's environment.

The EnvObject is meant to be used to represent entities in the larger environment that affect the execution of a study (and therefore jobs). This abstact base class should be used to represent things such as data dependencies, code dependencies, variables, aliases, etc. The only method we require is _verify, to allow users to verify that they've provided the minimal information for the object to be valid.

class maestrowf.abstracts.envobject.**Source**
Bases: *maestrowf.abstracts.envobject.EnvObject*

Abstract class representing classes that alter environment sourcing.

WARNING: The API for this class is still in development. The Source environment class is meant to provide a way to programmatically set environment settings that binaries or other scripts may require in the workflow. Such settings that are intended to be captured are:

- Exporting of shell/environment variables (using 'export')
- Setting of an environment package with the 'use' command

apply(*data*)
Apply the Source to some string data.

Subclasses of Source should use this method in order to apply an environment alterating change. The 'data' parameter should be a string representing a command to apply Source to or a list of other comands that Source should be included with.

Parameters **data** – A string representing a command or set of other sources.

Returns A string with the Source applied.

class maestrowf.abstracts.envobject.**Substitution**
Bases: *maestrowf.abstracts.envobject.EnvObject*

Abstract class representing classes that perform value replacements.

substitute(*data*)
Perform a replacement of some substring into data.

The method takes the input string data and performs a replacement. This API is used to represent concepts such as variables or parameters that would want to be replaced within the string data.

Parameters **data** – A string to perform a replacement on.

Returns A string equal to the original string data with substitutions made (if any were performed).

maestrowf.abstracts.graph module

A module representing an abstract Graph base class.

class maestrowf.abstracts.graph.**Graph**
Bases: object

An abstract graph data structure.

add_edge(*src*, *dest*)
Add the edge (src, dest) to the graph.

Parameters

- **src** – Source vertex name.
- **dest** – Destination vertex name.

add_node(*name*, *obj*)
Method to add a node to the graph.

Parameters

- **name** – String identifier of the node.
- **obj** – An object representing the value of the node.

remove_edge(*src*, *dest*)
Remove edge (src, dest) from the graph.

Parameters

- **src** – Source vertex name.
- **dest** – Destination vertex name.

maestrowf.abstracts.specification module

class maestrowf.abstracts.specification.**Specification**
Bases: object

Abstract class for loading and verifying a Study Specification

desc
Getter for the description of a study specification.

Returns A string containing the description of the study specification.

get_parameters()
Generate a ParameterGenerator object from the global parameters.

Returns A ParameterGenerator with data from the specification.

get_study_environment()
Generate a StudyEnvironment object from the environment in the spec.

Returns A StudyEnvironment object with the data in the specification.

get_study_steps()
Generate a list of StudySteps from the study in the specification.

Returns A list of StudyStep objects.

classmethod load_specification(*path*)
Method for loading a study specification from a file.

Parameters **path** – Path to a study specification.

Returns A specification object containing the information loaded from path.

classmethod load_specification_from_stream(*stream*)
Method for loading a study specification from a stream.

Parameters **stream** – Raw text stream containing specification data.

Returns A specification object containing the information in string.

name
Getter for the name of a study specification.

Returns The name of the study described by the specification.

output_path
Return the OUTPUT_PATH variable (if it exists).

Returns Returns OUTPUT_PATH if it exists, empty string otherwise.

verify()
Verify the whole specification.

7.1.1.2 maestrowf.datastructures package

The datastructures module contains all classes that other modules use.

The datastructures module contains all high level abstractions representing the aspects of a study. These structures include all abstracts that define interfaces objects should adhere to and other classes representing the package wide concepts of a Study, Environment, and Parameter generation.

class maestrowf.datastructures.**DAG**
Bases: *maestrowf.abstracts.graph.Graph*

A directed acyclic graph (DAG) data structure.

The implementation of this DAG uses an adjacency map with a map to index the values (or objects) at each node.

add_edge(*src*, *dest*)
Add an edge to the DAG if edge (src, dest) is a valid edge.

Parameters

- **src** – Source vertex name.
- **dest** – Destination vertex name.

add_node(*name*, *obj*)
Add node 'name' to the DAG.

Parameters

- **name** – String identifier of the node.
- **obj** – An object representing the value of the node.

bfs_subtree(*src*)
Create a subtree of the DAG starting at src in BFS order.

Parameters **src** – Source node name to begin search.

Returns A list representing the path taken by BFS.

Returns A dictionary containing a mapping from node to parent node.

detect_cycle()
Detect if the DAG contains a cycle.

dfs_subtree(*src*, *par=None*)
Create a subtree of the DAG starting at src in DFS order.

Parameters

- **src** – Source node name to begin search.
- **par** – Name of parent node to the specified source node.

Returns A list representing the path taken by DFS.

Returns A dictionary containing a mapping from node to parent node.

remove_edge(*src*, *dest*)
Remove edge (src, dest) from the DAG.

Parameters

- **src** – Source vertex name.
- **dest** – Destination vertex name.

topological_sort()
Perform a topological ordering of the vertices in the DAG.

Returns A list of the vertices sorted in topological order.

Subpackages

maestrowf.datastructures.core package

The core data structures for starting up studies.

This module contains all of the core data structures that are needed for constructing and representing studies and the moving parts that they require. These moving parts include but are not limited to:

- Classes for representing the abstract flow of a study. These objects at their core are the Study and StudyStep classes that are used to construct a DAG for the flow.
- Classes that represent the items in a study's environment such as variables, scripts, and dependencies (paths, git repos, etc.)
- Classes for managing the environment and that know how to apply the environment to an abstract flow.
- A set of classes for managing parameters and generating combinations of parameters in a clean Pythonic way.

class maestrowf.datastructures.core.**Combination**(*token='$'*)
Bases: object

Class representing a combination of parameters.

This class represents a combination of parameters generated by a class of type ParameterGenerator. The only time a user should ever get an instance of a Combination from the ParameterGenerator is when a combination of parameters is VALID.

add(*key*, *name*, *value*, *label*)
Add a parameter to the Combination object.

Parameters

- **key** – Parameter key that identifies a replacement.
- **name** – Custom name that identifies a parameter.
- **value** – Value of the parameter in this combination.
- **label** – Value of the parameter label for this combination.

apply(*item*)
Apply the combination to an item.

Parameters **item** – String that may contain parameters to be substituted.

Returns String equal to item, except with parameters replaced.

get_param_string(*params*)
Get the combination string for the specified parameters.

Parameters **params** – A set of parameters to be used in the string.

Returns A string containing the labels for the parameters in params.

class maestrowf.datastructures.core.**ExecutionGraph**(*submission_attempts=1*, *submission_throttle=0*, *use_tmp=False*, *dry_run=False*)
Bases: *maestrowf.datastructures.dag.DAG*, *maestrowf.abstracts.PickleInterface*

Datastructure that tracks, executes, and reports on study execution.

The ExecutionGraph is used to manage, monitor, and interact with tasks and the scheduler. This class searches its graph for tasks that are ready to run, marks tasks as complete, and schedules ready tasks.

The Execution class is where functionality for checking task status, logic for managing and automatically directing and manipulating the workflow should go. Essentially, if logic is needed to automatically manipulate the workflow in some fashion or additional monitoring is needed, this class is where that would go.

add_connection(*parent*, *step*)
Add a connection between two steps in the ExecutionGraph.

Parameters

- **parent** – The parent step that is required to execute 'step'
- **step** – The dependent step that relies on parent.

add_description(*name*, *description*, ***kwargs*)
Add a study description to the ExecutionGraph instance.

Parameters

- **name** – Name of the study.
- **description** – Description of the study.

add_step(*name*, *step*, *workspace*, *restart_limit*)
Add a StepRecord to the ExecutionGraph.

Parameters

- **name** – Name of the step to be added.
- **step** – StudyStep instance to be recorded.
- **workspace** – Directory path for the step's working directory.
- **restart_limit** – Upper limit on the number of restart attempts.

cancel_study()
Cancel the study.

check_study_status()
Check the status of currently executing steps in the graph.

This method is used to check the status of all currently in progress steps in the ExecutionGraph. Each ExecutionGraph stores the adapter used to generate and execute its scripts.

cleanup()
Clean up output produced by the ExecutionGraph.

description
Return the description for the study in the ExecutionGraph instance.

Returns A string of the description for the study.

execute_ready_steps()
Execute any steps whose dependencies are satisfied.

The 'execute_ready_steps' method is the core of how the ExecutionGraph manages execution. This method does the following:

- **Checks the status of existing jobs that are executing.**
  - Updates the state if changed.
- **Finds steps that are initialized and determines what can be run:**
  - Scans a steps dependencies and stages if all are me.
  - Executes any steps whose dependencies are met.

Returns True if the study has completed, False otherwise.

generate_scripts()
Generate the scripts for all steps in the ExecutionGraph.

The generate_scripts method scans the ExecutionGraph instance and uses the stored adapter to write executable scripts for either local or scheduled execution. If a restart command is specified, a restart script will be generated for that record.

log_description()
Log the description of the ExecutionGraph.

name
Return the name for the study in the ExecutionGraph instance.

Returns A string of the name of the study.

set_adapter(*adapter*)
Set the adapter used to interface for scheduling tasks.

Parameters **adapter** – Adapter name to be used when launching the graph.

write_status(*path*)
Write the status of the DAG to a CSV file.

class maestrowf.datastructures.core.**ParameterGenerator**(*token='$', ltoken='%%'*)
Bases: object

Class for containing parameters and generating combinations.

The goal of this class is to provide one centralized location for managing and storing parameters. This implementation of the ParameterGenerator, currently, is very basic. It takes lists of parameters and uses those to construct combinations, meaning that if you were to view this as an Excel table, you would have a row for each valid combination you wanted to study.

The other goal is to make it so that by having the ParameterGenerator manage parameters, functionality can be added without affecting how the end user interacts with this class. The ParameterGenerator has an Iterator defined and will generate each combination one by one. The end user should NEVER SEE AN INVALID

COMBINATION. Because this class generates the combinations as specified by the parameters added (eventually with types or enforced inheritence), and eventually constraints, it opens up being able to quietly change how this class generates its combinations.

Easily convert studies to other types of studies. Because the API doesn't change from its nice Pythonic style, you can in theory swap out a ParameterGenerator that performs completely differently. All of a sudden, you can get the following for simply deriving from this class:

- Uncertainty Quantification (UQ): Add the ability to statistically sample parameters behind the scenes. Let the ParameterGenerator constraint solve behind the scenes and return the Combination objects it was going to return in the first place. If you can't find a valid sampling, just return nothing and the study won't run.
- Boundary and constraint testing: Like UQ above, hide the solving from the user. Simply add parameters to be constraint solved on behind the API and all the user sees is combinations on the frontend.

Ideally, all parameter generation schemes should boil down as follows:

1. Derive from this class, add constraint solving.
2. Construct a study how you would otherwise do so, just use the new ParameterGenerator and add parameters.
3. Setup, stage, and execute your study.
4. Profit.

add_parameter(*key*, *values*, *label=None*, *name=None*)
Add a parameter to the ParameterGenerator.

Currently, all parameters added to a ParameterGenerator instance must have a list of values that are the same length. Future improvements will add the ability to specify either types of parameters or provide different ParameterGenerators derivations that have unique behavior.

Parameters

- **key** – Parameter key to find for replacement.
- **values** – List of values the parameter can take.
- **label** – Label string for labeling the parameter.
- **name** – Custom name for identifying parameter.

get_combinations()
Generate all combinations of parameters.

Returns A generator with all combinations of parameters.

get_metadata()
Produce metadata for the parameters in a generator instance.

Returns A dictionary containing metadata about the instance.

get_used_parameters(*step*)
Return the parameters used by a StudyStep.

Parameters **step** – A StudyStep instance to be checked.

Returns A set of the parameter names used within the step parameter.

class maestrowf.datastructures.core.**Study**(*name*, *description*, *studyenv=None*, *parameters=None*, *steps=None*, *out_path='./'*)
Bases: *maestrowf.datastructures.dag.DAG*, *maestrowf.abstracts.PickleInterface*

Collection of high level objects to perform study construction.

The Study class is part of the meat and potatoes of this whole package. A Study object is where the intersection of the major moving parts are collected. These moving parts include:

- ParameterGenerator for getting combinations of user parameters
- StudyEnvironment for managing and applying the environment to studies
- Study flow, which is a DAG of the abstract workflow

The class is responsible for a number of the major key steps in study setup as well. Those responsibilities include (but are not limited to):

- Setting up the workspace where a simulation campaign will be run.
- **Applying the StudyEnvionment to the abstract flow DAG:**
  - Creating the global workspace for a study.
  - Setting up the parameterized workspaces for each combination.
  - Acquiring dependencies as specified in the StudyEnvironment.
- **Intelligently constructing the expanded DAG to be able to:**
  - Recognize when a step executes in a parameterized workspace
  - Recognize when a step executes in the global workspace
- Expanding the abstract flow to the full set of specified parameters.

Future functionality that makes sense to add here:

- Metadata collection. If we're setting things up here, collect the

general information. We might even want to venture to say that a set of directives may be useful so that they could be placed into Dependency classes as hooks for dumping that data automatically. - A way of packaging an instance of the class up into something that is easy to store in the ExecutionDAG class so that an API can be designed in whatever class ends up managing all of this to have machine learning applications pipe messages to spin up new studies using the same environment.

- The current solution to this is VERY basic. Currently the plan is

to write a parameterized specification (not unlike the method of using parameterized .dat files for simulators) and just have the ML engine string replace those. It's crude because currently we'd have to just construct a new environment, with no way to manage injecting the new set into an existing workspace.

add_step(*step*)

Add a step to a study.

For this helper to be most effective, it recommended to apply steps in the order that they will be encountered. The method attempts to be intelligent and make the intended edge based on the 'depends' entry in a step. When adding steps out of order it's recommended to just use the base class DAG functionality and manually make connections.

param step A StudyStep instance to be added to the Study instance.

configure_study(*submission_attempts=1*, *restart_limit=1*, *throttle=0*, *use_tmp=False*, *hash_ws=False*, *dry_run=False*)

Perform initial configuration of a study. The method is used for going through and actually acquiring each dependency, substituting variables, sources and labels. :param submission_attempts: Number of attempted submissions before marking a step as failed. :param restart_limit: Upper limit on the number of times a step with a restart command can be resubmitted before it is considered failed. :param throttle: The

maximum number of in-progress jobs allowed. [0 denotes no cap]. :param use_tmp: Boolean value specifying if the generated ExecutionGraph dumps its information into a temporary directory. :param dry_run: Boolean value that toggles dry run to just generate study workspaces and scripts without execution or status checking. :returns: True if the Study is successfully setup, False otherwise.

load_metadata()
Load metadata for the study.

output_path
Property method for the OUTPUT_PATH specified for the study.

Returns The string path stored in the OUTPUT_PATH variable.

setup_environment()
Set up the environment by acquiring outside dependencies.

setup_workspace()
Set up the study's main workspace directory.

stage()
Generate the execution graph for a Study.

Staging creates an ExecutionGraph based on the combinations generated by the ParameterGeneration object stored in an instance of a Study. The stage method also sets up individual working directories (or workspaces) for each node in the workflow that requires it.

Returns An ExecutionGraph object with the expanded workflow.

store_metadata()
Store metadata related to the study.

walk_study(*src='_source'*)
Walk the study and create a spanning tree.

Parameters **src** – Source node to start the walk.

Returns A generator of (parent, node name, node value) tuples.

class maestrowf.datastructures.core.**StudyEnvironment**
Bases: object

StudyEnvironment for managing a study environment.

The StudyEnvironment provides the context where all study steps can find variables, sources, dependencies, etc.

acquire_environment()
Acquire any environment items that may be stored remotely.

add(*item*)
Add the item parameter to the StudyEnvironment.

Parameters **item** – EnvObject to be added to the environment.

apply_environment(*item*)
Apply the environment to the specified item.

Parameters **item** – String to apply environment to.

Returns String with the environment applied.

find(*key*)
Find the environment object labeled by the specified key.

Parameters **key** – Name of the environment object to find.

Returns The environment object labeled by key, None if key is not found.

is_set_up
Check that the StudyEnvironment is set up.

Returns True is the instance is set up, False otherwise.

remove(*key*)
Remove the environment object labeled by the specified key.

Parameters **key** – Name of the environment object to remove.

Returns The environment object labeled by key.

class maestrowf.datastructures.core.**StudyStep**
Bases: object

Class that represents the data and API for a single study step.

This class is primarily a 1:1 mapping of a study step in the YAML spec in terms of data. The StudyStep's class API should capture all functions that a step can be expected to perform, including:

- Applying a combination of parameters to itself.
- Tests for equality and non-equality to check for changes.
- Other – WIP

apply_parameters(*combo*)
Apply a parameter combination to the StudyStep.

Parameters **combo** – A Combination instance to be applied to a StudyStep.

Returns A new StudyStep instance with combo applied to its members.

Submodules

maestrowf.datastructures.core.executiongraph module

Module for the execution of DAG workflows.

class maestrowf.datastructures.core.executiongraph.**ExecutionGraph**(*submission_attempts=1*, *submission_throttle=0*, *use_tmp=False*, *dry_run=False*)
Bases: *maestrowf.datastructures.dag.DAG*, *maestrowf.abstracts.PickleInterface*

Datastructure that tracks, executes, and reports on study execution.

The ExecutionGraph is used to manage, monitor, and interact with tasks and the scheduler. This class searches its graph for tasks that are ready to run, marks tasks as complete, and schedules ready tasks.

The Execution class is where functionality for checking task status, logic for managing and automatically directing and manipulating the workflow should go. Essentially, if logic is needed to automatically manipulate the workflow in some fashion or additional monitoring is needed, this class is where that would go.

add_connection(*parent*, *step*)
Add a connection between two steps in the ExecutionGraph.

Parameters

- **parent** – The parent step that is required to execute 'step'

- **step** – The dependent step that relies on parent.

add_description(*name*, *description*, ***kwargs*)
Add a study description to the ExecutionGraph instance.

Parameters

- **name** – Name of the study.
- **description** – Description of the study.

add_step(*name*, *step*, *workspace*, *restart_limit*)
Add a StepRecord to the ExecutionGraph.

Parameters

- **name** – Name of the step to be added.
- **step** – StudyStep instance to be recorded.
- **workspace** – Directory path for the step's working directory.
- **restart_limit** – Upper limit on the number of restart attempts.

cancel_study()
Cancel the study.

check_study_status()
Check the status of currently executing steps in the graph.

This method is used to check the status of all currently in progress steps in the ExecutionGraph. Each ExecutionGraph stores the adapter used to generate and execute its scripts.

cleanup()
Clean up output produced by the ExecutionGraph.

description
Return the description for the study in the ExecutionGraph instance.

Returns A string of the description for the study.

execute_ready_steps()
Execute any steps whose dependencies are satisfied.

The 'execute_ready_steps' method is the core of how the ExecutionGraph manages execution. This method does the following:

- **Checks the status of existing jobs that are executing.**
  - Updates the state if changed.
- **Finds steps that are initialized and determines what can be run:**
  - Scans a steps dependencies and stages if all are me.
  - Executes any steps whose dependencies are met.

Returns True if the study has completed, False otherwise.

generate_scripts()
Generate the scripts for all steps in the ExecutionGraph.

The generate_scripts method scans the ExecutionGraph instance and uses the stored adapter to write executable scripts for either local or scheduled execution. If a restart command is specified, a restart script will be generated for that record.

log_description()
Log the description of the ExecutionGraph.

name
Return the name for the study in the ExecutionGraph instance.

Returns A string of the name of the study.

set_adapter(*adapter*)
Set the adapter used to interface for scheduling tasks.

Parameters **adapter** – Adapter name to be used when launching the graph.

write_status(*path*)
Write the status of the DAG to a CSV file.

maestrowf.datastructures.core.parameters module

This module contains classes related to parameters and parameter generation.

The goal of this module is to abstract away how parameters are generated from the user. In terms of parameters the only things a user should see is the ParameterGenerator that offers an API for managing parameters and generates individual Combinations (the second object a user should ever see).

class maestrowf.datastructures.core.parameters.**Combination**(*token='$'*)
Bases: object

Class representing a combination of parameters.

This class represents a combination of parameters generated by a class of type ParameterGenerator. The only time a user should ever get an instance of a Combination from the ParameterGenerator is when a combination of parameters is VALID.

add(*key*, *name*, *value*, *label*)
Add a parameter to the Combination object.

Parameters

- **key** – Parameter key that identifies a replacement.
- **name** – Custom name that identifies a parameter.
- **value** – Value of the parameter in this combination.
- **label** – Value of the parameter label for this combination.

apply(*item*)
Apply the combination to an item.

Parameters **item** – String that may contain parameters to be substituted.

Returns String equal to item, except with parameters replaced.

get_param_string(*params*)
Get the combination string for the specified parameters.

Parameters **params** – A set of parameters to be used in the string.

Returns A string containing the labels for the parameters in params.

class maestrowf.datastructures.core.parameters.**ParameterGenerator**(*token='$'*, *ltoken='%%'*)
Bases: object

Class for containing parameters and generating combinations.

The goal of this class is to provide one centralized location for managing and storing parameters. This implementation of the ParameterGenerator, currently, is very basic. It takes lists of parameters and uses those to construct combinations, meaning that if you were to view this as an Excel table, you would have a row for each valid combination you wanted to study.

The other goal is to make it so that by having the ParameterGenerator manage parameters, functionality can be added without affecting how the end user interacts with this class. The ParameterGenerator has an Iterator defined and will generate each combination one by one. The end user should NEVER SEE AN INVALID COMBINATION. Because this class generates the combinations as specified by the parameters added (eventually with types or enforced inheritence), and eventually constraints, it opens up being able to quietly change how this class generates its combinations.

Easily convert studies to other types of studies. Because the API doesn't change from its nice Pythonic style, you can in theory swap out a ParameterGenerator that performs completely differently. All of a sudden, you can get the following for simply deriving from this class:

- Uncertainty Quantification (UQ): Add the ability to statistically sample parameters behind the scenes. Let the ParameterGenerator constraint solve behind the scenes and return the Combination objects it was going to return in the first place. If you can't find a valid sampling, just return nothing and the study won't run.
- Boundary and constraint testing: Like UQ above, hide the solving from the user. Simply add parameters to be constraint solved on behind the API and all the user sees is combinations on the frontend.

Ideally, all parameter generation schemes should boil down as follows:

1. Derive from this class, add constraint solving.
2. Construct a study how you would otherwise do so, just use the new ParameterGenerator and add parameters.
3. Setup, stage, and execute your study.
4. Profit.

add_parameter(*key*, *values*, *label=None*, *name=None*)
Add a parameter to the ParameterGenerator.

Currently, all parameters added to a ParameterGenerator instance must have a list of values that are the same length. Future improvements will add the ability to specify either types of parameters or provide different ParameterGenerators derivations that have unique behavior.

Parameters

- **key** – Parameter key to find for replacement.
- **values** – List of values the parameter can take.
- **label** – Label string for labeling the parameter.
- **name** – Custom name for identifying parameter.

get_combinations()
Generate all combinations of parameters.

Returns A generator with all combinations of parameters.

get_metadata()
Produce metadata for the parameters in a generator instance.

Returns A dictionary containing metadata about the instance.

get_used_parameters(*step*)
Return the parameters used by a StudyStep.

Parameters **step** – A StudyStep instance to be checked.

Returns A set of the parameter names used within the step parameter.

maestrowf.datastructures.core.study module

Class related to the construction of study campaigns.

class maestrowf.datastructures.core.study.**Study**(*name*, *description*, *studyenv=None*, *parameters=None*, *steps=None*, *out_path='./'*)
Bases: *maestrowf.datastructures.dag.DAG*, *maestrowf.abstracts.PickleInterface*

Collection of high level objects to perform study construction.

The Study class is part of the meat and potatoes of this whole package. A Study object is where the intersection of the major moving parts are collected. These moving parts include:

- ParameterGenerator for getting combinations of user parameters
- StudyEnvironment for managing and applying the environment to studies
- Study flow, which is a DAG of the abstract workflow

The class is responsible for a number of the major key steps in study setup as well. Those responsibilities include (but are not limited to):

- Setting up the workspace where a simulation campaign will be run.
- **Applying the StudyEnvionment to the abstract flow DAG:**
  - Creating the global workspace for a study.
  - Setting up the parameterized workspaces for each combination.
  - Acquiring dependencies as specified in the StudyEnvironment.
- **Intelligently constructing the expanded DAG to be able to:**
  - Recognize when a step executes in a parameterized workspace
  - Recognize when a step executes in the global workspace
- Expanding the abstract flow to the full set of specified parameters.

Future functionality that makes sense to add here:

- Metadata collection. If we're setting things up here, collect the

general information. We might even want to venture to say that a set of directives may be useful so that they could be placed into Dependency classes as hooks for dumping that data automatically. - A way of packaging an instance of the class up into something that is easy to store in the ExecutionDAG class so that an API can be designed in whatever class ends up managing all of this to have machine learning applications pipe messages to spin up new studies using the same environment.

- The current solution to this is VERY basic. Currently the plan is

to write a parameterized specification (not unlike the method of using parameterized .dat files for simulators) and just have the ML engine string replace those. It's crude because currently we'd have to just construct a new environment, with no way to manage injecting the new set into an existing workspace.

add_step(*step*)
Add a step to a study.

For this helper to be most effective, it recommended to apply steps in the order that they will be encountered. The method attempts to be intelligent and make the intended edge based on the 'depends' entry in a step. When adding steps out of order it's recommended to just use the base class DAG functionality and manually make connections.

param step A StudyStep instance to be added to the Study instance.

configure_study(*submission_attempts=1*, *restart_limit=1*, *throttle=0*, *use_tmp=False*, *hash_ws=False*, *dry_run=False*)
Perform initial configuration of a study. The method is used for going through and actually acquiring each dependency, substituting variables, sources and labels. :param submission_attempts: Number of attempted submissions before marking a step as failed. :param restart_limit: Upper limit on the number of times a step with a restart command can be resubmitted before it is considered failed. :param throttle: The maximum number of in-progress jobs allowed. [0 denotes no cap]. :param use_tmp: Boolean value specifying if the generated ExecutionGraph dumps its information into a temporary directory. :param dry_run: Boolean value that toggles dry run to just generate study workspaces and scripts without execution or status checking. :returns: True if the Study is successfully setup, False otherwise.

load_metadata()
Load metadata for the study.

output_path
Property method for the OUTPUT_PATH specified for the study.

Returns The string path stored in the OUTPUT_PATH variable.

setup_environment()
Set up the environment by acquiring outside dependencies.

setup_workspace()
Set up the study's main workspace directory.

stage()
Generate the execution graph for a Study.

Staging creates an ExecutionGraph based on the combinations generated by the ParameterGeneration object stored in an instance of a Study. The stage method also sets up individual working directories (or workspaces) for each node in the workflow that requires it.

Returns An ExecutionGraph object with the expanded workflow.

store_metadata()
Store metadata related to the study.

walk_study(*src='_source'*)
Walk the study and create a spanning tree.

Parameters **src** – Source node to start the walk.

Returns A generator of (parent, node name, node value) tuples.

class maestrowf.datastructures.core.study.**StudyStep**
Bases: object

Class that represents the data and API for a single study step.

This class is primarily a 1:1 mapping of a study step in the YAML spec in terms of data. The StudyStep's class API should capture all functions that a step can be expected to perform, including:

- Applying a combination of parameters to itself.

- Tests for equality and non-equality to check for changes.
- Other – WIP

apply_parameters(*combo*)
Apply a parameter combination to the StudyStep.

Parameters **combo** – A Combination instance to be applied to a StudyStep.

Returns A new StudyStep instance with combo applied to its members.

maestrowf.datastructures.core.studyenvironment module

Classes that represent the environment of a study.

class maestrowf.datastructures.core.studyenvironment.**StudyEnvironment**
Bases: object

StudyEnvironment for managing a study environment.

The StudyEnvironment provides the context where all study steps can find variables, sources, dependencies, etc.

acquire_environment()
Acquire any environment items that may be stored remotely.

add(*item*)
Add the item parameter to the StudyEnvironment.

Parameters **item** – EnvObject to be added to the environment.

apply_environment(*item*)
Apply the environment to the specified item.

Parameters **item** – String to apply environment to.

Returns String with the environment applied.

find(*key*)
Find the environment object labeled by the specified key.

Parameters **key** – Name of the environment object to find.

Returns The environment object labeled by key, None if key is not found.

is_set_up
Check that the StudyEnvironment is set up.

Returns True is the instance is set up, False otherwise.

remove(*key*)
Remove the environment object labeled by the specified key.

Parameters **key** – Name of the environment object to remove.

Returns The environment object labeled by key.

maestrowf.datastructures.environment package

Classes that represent the environment of a study.

class maestrowf.datastructures.environment.**GitDependency**(*name*, *value*, *path*, *token='$'*, ***kwargs*)

Bases: *maestrowf.abstracts.envobject.Dependency*

Environment GitDependency class for substituting a git dependency.

acquire(*substitutions=None*)

Acquire the dependency specified by the PathDependency.

The GitDependency will clone the remote repository specified by the instance's value to the local repository specified by path. If a commit hash is specified, acquire will attempt to rebase to the repository version described by the hash. Alternatively, if a tag is specfied acquire will attempt to checkout the version labeled by the tag.

Parameters **substitutions** – List of Substitution objects that can be applied.

get_var()

Get the variable representation of the dependency's name.

Returns String of the Dependencies's name in token form.

substitute(*data*)

Substitute the dependency's value for its notation.

Parameters **data** – String to substitute dependency into.

Returns String with the dependency's name replaced with its value.

class maestrowf.datastructures.environment.**PathDependency**(*name*, *value*, *token='$'*)

Bases: *maestrowf.abstracts.envobject.Dependency*

Environment PathDependency class for substituting a path dependency.

acquire(*substitutions=None*)

Acquire the dependency specified by the PathDependency.

The PathDependency is simply a path that already exists, so the method doesn't actually acquire anything, but it does verify that the path exists.

Parameters **substitutions** – List of Substitution objects that can be applied.

get_var()

Get the variable representation of the dependency's name.

Returns String of the Dependencies's name in token form.

substitute(*data*)

Substitute the dependency's value for its notation.

Parameters **data** – String to substitute dependency into.

Returns String with the dependency's name replaced with its value.

class maestrowf.datastructures.environment.**Script**(*source*)

Bases: *maestrowf.abstracts.envobject.Source*

Script class for applying changes to the execution environment.

apply(*cmds*)

Apply the Script source to the specified list of commands.

Parameters **cmds** – List of commands to add source to.

Returns List of commands with the source prepended.

class maestrowf.datastructures.environment.**Variable**(*name*, *value*, *token='$'*)
Bases: *maestrowf.abstracts.envobject.Substitution*

Environment Variable class capable of substituting itself into strings.

Derived from the Substitution EnvObject class which requires that a substitution be able to inject itself into data.

get_var()
Get the variable representation of the variable's name.

Returns String of the Variable's name in token form.

substitute(*data*)
Substitute the variable's value for its notation.

Parameters **data** – String to substitute variable into.

Returns String with the variable's name replaced with its value.

Submodules

maestrowf.datastructures.environment.gitdependency module

Classes that represent the environment of a study.

class maestrowf.datastructures.environment.gitdependency.**GitDependency**(*name*, *value*, *path*, *token='$'*, ***kwargs*)
Bases: *maestrowf.abstracts.envobject.Dependency*

Environment GitDependency class for substituting a git dependency.

acquire(*substitutions=None*)
Acquire the dependency specified by the PathDependency.

The GitDependency will clone the remote repository specified by the instance's value to the local repository specified by path. If a commit hash is specified, acquire will attempt to rebase to the repository version described by the hash. Alternatively, if a tag is specfied acquire will attempt to checkout the version labeled by the tag.

Parameters **substitutions** – List of Substitution objects that can be applied.

get_var()
Get the variable representation of the dependency's name.

Returns String of the Dependencies's name in token form.

substitute(*data*)
Substitute the dependency's value for its notation.

Parameters **data** – String to substitute dependency into.

Returns String with the dependency's name replaced with its value.

maestrowf.datastructures.environment.pathdependency module

Class representing a file system path dependency.

class maestrowf.datastructures.environment.pathdependency.**PathDependency**(*name*, *value*, *token='$'*)

Bases: *maestrowf.abstracts.envobject.Dependency*

Environment PathDependency class for substituting a path dependency.

acquire(*substitutions=None*)

Acquire the dependency specified by the PathDependency.

The PathDependency is simply a path that already exists, so the method doesn't actually acquire anything, but it does verify that the path exists.

Parameters **substitutions** – List of Substitution objects that can be applied.

get_var()

Get the variable representation of the dependency's name.

Returns String of the Dependencies's name in token form.

substitute(*data*)

Substitute the dependency's value for its notation.

Parameters **data** – String to substitute dependency into.

Returns String with the dependency's name replaced with its value.

maestrowf.datastructures.environment.script module

Class representing the sourcing of a script.

class maestrowf.datastructures.environment.script.**Script**(*source*)

Bases: *maestrowf.abstracts.envobject.Source*

Script class for applying changes to the execution environment.

apply(*cmds*)

Apply the Script source to the specified list of commands.

Parameters **cmds** – List of commands to add source to.

Returns List of commands with the source prepended.

maestrowf.datastructures.environment.variable module

Class for handling Variable substitutions.

class maestrowf.datastructures.environment.variable.**Variable**(*name*, *value*, *token='$'*)

Bases: *maestrowf.abstracts.envobject.Substitution*

Environment Variable class capable of substituting itself into strings.

Derived from the Substitution EnvObject class which requires that a substitution be able to inject itself into data.

get_var()
Get the variable representation of the variable's name.

Returns String of the Variable's name in token form.

substitute(*data*)
Substitute the variable's value for its notation.

Parameters **data** – String to substitute variable into.

Returns String with the variable's name replaced with its value.

Submodules

maestrowf.datastructures.dag module

Module that contains the implemenation of a Directed-Acyclic Graph.

class maestrowf.datastructures.dag.**DAG**
Bases: *maestrowf.abstracts.graph.Graph*

A directed acyclic graph (DAG) data structure.

The implementation of this DAG uses an adjacency map with a map to index the values (or objects) at each node.

add_edge(*src*, *dest*)
Add an edge to the DAG if edge (src, dest) is a valid edge.

Parameters

- **src** – Source vertex name.
- **dest** – Destination vertex name.

add_node(*name*, *obj*)
Add node 'name' to the DAG.

Parameters

- **name** – String identifier of the node.
- **obj** – An object representing the value of the node.

bfs_subtree(*src*)
Create a subtree of the DAG starting at src in BFS order.

Parameters **src** – Source node name to begin search.

Returns A list representing the path taken by BFS.

Returns A dictionary containing a mapping from node to parent node.

detect_cycle()
Detect if the DAG contains a cycle.

dfs_subtree(*src*, *par=None*)
Create a subtree of the DAG starting at src in DFS order.

Parameters

- **src** – Source node name to begin search.
- **par** – Name of parent node to the specified source node.

Returns A list representing the path taken by DFS.

Returns A dictionary containing a mapping from node to parent node.

remove_edge(*src*, *dest*)
Remove edge (src, dest) from the DAG.

Parameters

- **src** – Source vertex name.
- **dest** – Destination vertex name.

topological_sort()
Perform a topological ordering of the vertices in the DAG.

Returns A list of the vertices sorted in topological order.

maestrowf.datastructures.yamlspecification module

7.1.1.3 maestrowf.interfaces package

Collection of custom adapters for interfacing with various systems.

class maestrowf.interfaces.**ScriptAdapterFactory**
Bases: object

factories = {'flux': <class 'maestrowf.interfaces.script.fluxscriptadapter.FluxScript

classmethod get_adapter(*adapter_id*)

classmethod get_valid_adapters()

Subpackages

maestrowf.interfaces.script package

Module for interfaces that support various schedulers.

class maestrowf.interfaces.script.**CancellationRecord**(*cancel_status*, *retcode*)
Bases: maestrowf.abstracts.containers.Record

A container for data returned from a scheduler cancellation call.

add_status(*jobid*, *cancel_status*)
Add the cancellation status for a single job to a record.

Parameters

- **jobid** – Unique job identifier for the job status to be added.
- **cancel_status** – CancelCode designating how cancellation terminated.

cancel_status
Get the high level CancelCode status.

lookup_status(*cancel_status*)
Find the cancellation status of the job identified by jid.

Parameters **cancel_status** – The CancelCode to look up.

Returns Set of job identifiers that match the requested status.

return_code
Get the return code from the cancel command.

class maestrowf.interfaces.script.**SubmissionRecord**(*subcode*, *retcode*, *jobid=-1*)
Bases: maestrowf.abstracts.containers.Record

A container for data about return state upon scheduler submission.

add_info(*key*, *value*)
Set additional informational key-value information.

Parameters

- **key** – Record key identifying data.
- **value** – Data to be recorded.

job_identifier
Property for the job identifier for the record.

Returns A string representing the job identifer assigned by the scheduler.

return_code
Property for the raw return code returned from submission.

Returns An integer representing the state of the raw return code from submission.

submission_code
Property for submission state for the record.

Returns A SubmissionCode enum representing the state of the submission call.

Submodules

maestrowf.interfaces.script.localscriptadapter module

Local interface implementation.

class maestrowf.interfaces.script.localscriptadapter.**LocalScriptAdapter**(***kwargs*)
Bases: *maestrowf.abstracts.interfaces.scriptadapter.ScriptAdapter*

A ScriptAdapter class for interfacing for local execution.

cancel_jobs(*joblist*)
For the given job list, cancel each job.

Parameters **joblist** – A list of job identifiers to be cancelled.

Returns The return code to indicate if jobs were cancelled.

check_jobs(*joblist*)
For the given job list, query execution status.

Parameters **joblist** – A list of job identifiers to be queried.

Returns The return code of the status query, and a dictionary of job identifiers to their status.

key = 'local'

submit(*step*, *path*, *cwd*, *job_map=None*, *env=None*)
Execute the step locally.

If cwd is specified, the submit method will operate outside of the path specified by the 'cwd' parameter. If env is specified, the submit method will set the environment variables for submission to the specified values. The 'env' parameter should be a dictionary of environment variables.

Parameters

- **step** – An instance of a StudyStep.
- **path** – Path to the script to be executed.
- **cwd** – Path to the current working directory.
- **job_map** – A map of workflow step names to their job identifiers.
- **env** – A dict containing a modified environment for execution.

Returns The return code of the submission command and job identiifer.

maestrowf.interfaces.script.slurmscriptadapter module

Slurm Scheduler interface implementation.

class maestrowf.interfaces.script.slurmscriptadapter.**SlurmScriptAdapter**(***kwargs*)

Bases: *maestrowf.abstracts.interfaces.schedulerscriptadapter.SchedulerScriptAdapter*

A ScriptAdapter class for interfacing with the SLURM scheduler.

cancel_jobs(*joblist*)

For the given job list, cancel each job.

Parameters **joblist** – A list of job identifiers to be cancelled.

Returns The return code to indicate if jobs were cancelled.

check_jobs(*joblist*)

For the given job list, query execution status.

This method uses the scontrol show job <jobid> command and does a regex search for job information.

Parameters **joblist** – A list of job identifiers to be queried.

Returns The return code of the status query, and a dictionary of job identifiers to their status.

get_header(*step*)

Generate the header present at the top of Slurm execution scripts.

Parameters **step** – A StudyStep instance.

Returns A string of the header based on internal batch parameters and the parameter step.

get_parallelize_command(*procs*, *nodes=None*, ***kwargs*)

Generate the SLURM parallelization segement of the command line.

Parameters

- **procs** – Number of processors to allocate to the parallel call.
- **nodes** – Number of nodes to allocate to the parallel call (default = 1).

Returns A string of the parallelize command configured using nodes and procs.

key = 'slurm'

submit(*step*, *path*, *cwd*, *job_map=None*, *env=None*)

Submit a script to the Slurm scheduler.

Parameters

- **step** – The StudyStep instance this submission is based on.
- **path** – Local path to the script to be executed.
- **cwd** – Path to the current working directory.
- **job_map** – A dictionary mapping step names to their job identifiers.
- **env** – A dict containing a modified environment for execution.

Returns The return status of the submission command and job identiifer.

maestrowf.interfaces.script.lsfscriptadapter module

LSF Scheduler interface implementation.

class maestrowf.interfaces.script.lsfscriptadapter.**LSFScriptAdapter**(***kwargs*)

Bases: *maestrowf.abstracts.interfaces.schedulerscriptadapter.SchedulerScriptAdapter*

A ScriptAdapter class for interfacing with the LSF cluster scheduler.

NOJOB_REGEX = re.compile('^No\\s')

cancel_jobs(*joblist*)

For the given job list, cancel each job.

Parameters **joblist** – A list of job identifiers to be cancelled.

Returns The return code to indicate if jobs were cancelled.

check_jobs(*joblist*)

For the given job list, query execution status.

This method uses the scontrol show job <jobid> command and does a regex search for job information.

Parameters **joblist** – A list of job identifiers to be queried.

Returns The return code of the status query, and a dictionary of job identifiers to their status.

get_header(*step*)

Generate the header present at the top of LSF execution scripts.

Parameters **step** – A StudyStep instance.

Returns A string of the header based on internal batch parameters and the parameter step.

get_parallelize_command(*procs*, *nodes=None*, ***kwargs*)

Generate the LSF parallelization segement of the command line.

Parameters

- **procs** – Number of processors to allocate to the parallel call.
- **nodes** – Number of nodes to allocate to the parallel call (default = 1).

Returns A string of the parallelize command configured using nodes and procs.

key = 'lsf'

submit(*step*, *path*, *cwd*, *job_map=None*, *env=None*)

Submit a script to the LSF scheduler.

Parameters

- **step** – The StudyStep instance this submission is based on.
- **path** – Local path to the script to be executed.
- **cwd** – Path to the current working directory.
- **job_map** – A dictionary mapping step names to their job identifiers.
- **env** – A dict containing a modified environment for execution.

Returns The return status of the submission command and job identiifer.

maestrowf.interfaces.script.fluxscriptadapter module

Flux Scheduler interface implementation.

class maestrowf.interfaces.script.fluxscriptadapter.**FluxScriptAdapter**(***kwargs*)

Bases: *maestrowf.abstracts.interfaces.schedulerscriptadapter.SchedulerScriptAdapter*

Interface class for the flux scheduler (on Spectrum MPI).

cancel_jobs(*joblist*)

For the given job list, cancel each job.

Parameters **joblist** – A list of job identifiers to be cancelled.

Returns The return code to indicate if jobs were cancelled.

check_jobs(*joblist*)

For the given job list, query execution status.

This method uses the scontrol show job <jobid> command and does a regex search for job information.

Parameters **joblist** – A list of job identifiers to be queried.

Returns The return code of the status query, and a dictionary of job identifiers to their status.

get_header(*step*)

Generate the header present at the top of Flux execution scripts.

Parameters **step** – A StudyStep instance.

Returns A string of the header based on internal batch parameters and the parameter step.

get_parallelize_command(*procs*, *nodes=None*, ***kwargs*)

Generate the FLUX parallelization segement of the command line.

Parameters

- **procs** – Number of processors to allocate to the parallel call.
- **nodes** – Number of nodes to allocate to the parallel call (default = 1).

Returns A string of the parallelize command configured using nodes and procs.

key = 'flux'

submit(*step*, *path*, *cwd*, *job_map=None*, *env=None*)

Submit a script to the Flux scheduler.

Parameters

- **step** – The StudyStep instance this submission is based on.
- **path** – Local path to the script to be executed.

- **cwd** – Path to the current working directory.
- **job_map** – A dictionary mapping step names to their job identifiers.
- **env** – A dict containing a modified environment for execution.

Returns The return status of the submission command and job identiifer.

class maestrowf.interfaces.script.fluxscriptadapter.**SpectrumFluxScriptAdapter**(***kwargs*)

Bases: *maestrowf.abstracts.interfaces.schedulerscriptadapter.SchedulerScriptAdapter*

Interface class for the flux scheduler (on Spectrum MPI).

cancel_jobs(*joblist*)

For the given job list, cancel each job.

Parameters **joblist** – A list of job identifiers to be cancelled.

Returns The return code to indicate if jobs were cancelled.

check_jobs(*joblist*)

For the given job list, query execution status.

This method uses the scontrol show job <jobid> command and does a regex search for job information.

Parameters **joblist** – A list of job identifiers to be queried.

Returns The return code of the status query, and a dictionary of job identifiers to their status.

get_header(*step*)

Generate the header present at the top of Flux execution scripts.

Parameters **step** – A StudyStep instance.

Returns A string of the header based on internal batch parameters and the parameter step.

get_parallelize_command(*procs*, *nodes=None*, ***kwargs*)

Generate the FLUX parallelization segement of the command line.

Parameters

- **procs** – Number of processors to allocate to the parallel call.
- **nodes** – Number of nodes to allocate to the parallel call (default = 1).

Returns A string of the parallelize command configured using nodes and procs.

key = 'flux-spectrum'

submit(*step*, *path*, *cwd*, *job_map=None*, *env=None*)

Submit a script to the Flux scheduler.

Parameters

- **step** – The StudyStep instance this submission is based on.
- **path** – Local path to the script to be executed.
- **cwd** – Path to the current working directory.
- **job_map** – A dictionary mapping step names to their job identifiers.
- **env** – A dict containing a modified environment for execution.

Returns The return status of the submission command and job identiifer.

maestrowf.interfaces.script.fluxscriptadapter.**get_environment**()

Filter environment variables based on a naming filter.

7.1.2 Submodules

7.1.3 maestrowf.conductor module

A script for launching the Maestro conductor for study monitoring.

class maestrowf.conductor.**Conductor**(*study*)
Bases: object

A class that provides an API for interacting with the Conductor.

cleanup()

classmethod get_status(*output_path*)
Retrieve the status of the study rooted at 'out_path'.

Parameters **out_path** – A string containing the patht to a study root.

Returns A dictionary containing the status of the study.

initialize(*batch_info*, *sleeptime=60*)
Initializes the Conductor instance based on the stored study.

Parameters

- **batch_info** – A dict containing batch information.
- **sleeptime** – The amount of sleep time between polling loops [Default: 60s].

classmethod load_batch(*out_path*)
Load the batch information for the study rooted in 'out_path'.

Parameters **out_path** – A string containing the path to a study root.

Returns A dict containing the batch information for the study.

classmethod load_study(*out_path*)
Load the Study instance in the study root specified by 'out_path'.

Parameters **out_path** – A string containing the patht to a study root.

Returns A string containing the path to the study's root.

classmethod mark_cancelled(*output_path*)
Mark the study rooted at 'out_path'.

Parameters **out_path** – A string containing the patht to a study root.

Returns A dictionary containing the status of the study.

monitor_study()
Monitor a running study.

output_path
Return the path representing the root of the study workspace.

Returns A string containing the path to the study's root.

classmethod store_batch(*out_path*, *batch*)
Store the specified batch information to the study in 'out_path'.

Parameters **out_path** – A string containing the patht to a study root.

classmethod store_study(*study*)
Store a Maestro study instance in a way the Conductor can read it.

study_name
Return the name of the study this Conductor instance is managing.

Returns A string containing the name of the study.

maestrowf.conductor.**main**()
Run the main segment of the conductor.

maestrowf.conductor.**setup_logging**(*name*, *output_path*, *log_lvl=2*, *log_path=None*, *log_stdout=False*, *log_format=None*)
Set up logging in the Main class. :param args: A Namespace object created by a parsed ArgumentParser. :param name: The name of the log file.

maestrowf.conductor.**setup_parser**()
Set up the Conductors's argument parser.

Returns A ArgumentParser that's initialized with the conductor's CLI.

7.1.4 maestrowf.maestro module

A script for launching a YAML study specification.

maestrowf.maestro.**cancel_study**(*args*)
Flag a study to be cancelled.

maestrowf.maestro.**load_parameter_generator**(*path*, *env*, *kwargs*)
Import and load custom parameter Python files.

Parameters

- **path** – Path to a Python file containing the function 'get_custom_generator'.
- **env** – A StudyEnvironment object containing custom information.
- **kwargs** – Dictionary containing keyword arguments for the function 'get_custom_generator'.

Returns A populated ParameterGenerator instance.

maestrowf.maestro.**main**()
Execute the main program's functionality.

This function uses command line arguments to locate the study description. It makes use of the maestrowf core data structures as a high level class inerface.

maestrowf.maestro.**run_study**(*args*)
Run a Maestro study.

maestrowf.maestro.**setup_argparser**()
Set up the program's argument parser.

maestrowf.maestro.**status_study**(*args*)
Check and print the status of an executing study.

7.1.5 maestrowf.utils module

A collection of more general utility functions.

class maestrowf.utils.**LoggerUtility**(*logger*)
Bases: object

Utility class for setting up logging consistently.

add_file_handler(*log_path*, *log_format*, *log_lvl=2*)
Add a file handler to logging.

Parameters

- **log_path** – String containing the file path to store logging.
- **log_format** – String containing the desired logging format.
- **log_lvl** – Integer level (1-5) to set the logger to.

add_stream_handler(*log_format*, *log_lvl=2*)
Add a stream handler to logging.

Parameters

- **log_format** – String containing the desired logging format.
- **log_lvl** – Integer level (1-5) to set the logger to.

configure(*log_format*, *log_lvl=2*, *colors=True*)
Configures the general logging facility.

Parameters

- **log_format** – String containing the desired logging format.
- **log_lvl** – Integer level (1-5) to set the logger to.

static map_level(*log_lvl*)
Map level 1-5 to their respective logging enumerations.

Parameters **log_lvl** – Integer level (1-5) representing logging verbosity.

maestrowf.utils.**apply_function**(*item*, *func*)
Apply a function to items depending on type.

Parameters

- **item** – A Python primitive to apply a function to.
- **func** – Function that returns takes item as a parameter and returns item modified in some way.

maestrowf.utils.**create_dictionary**(*list_keyvalues*, *token=':'*)
Create a dictionary from a list of key-value pairs.

Parameters

- **list_keyvalues** – List of token separates key-values.
- **token** – The token to split each key-value by.

Returns A dictionary containing the key-value pairings in list_keyvalues.

maestrowf.utils.**create_parentdir**(*path*)
Recursively create parent directories.

Parameters **path** – Path to a directory to be created.

maestrowf.utils.**csvtable_to_dict**(*fstream*)
Convert a csv file stream into an in memory dictionary.

Parameters **fstream** – An open file stream to a csv table (with header)

Returns A dictionary with a key for each column header and a list of column values for each key.

maestrowf.utils.**generate_filename**(*path*, *append_time=True*)
Generate a non-conflicting file name.

Parameters

- **path** – Path to file.
- **append_time** – Setting to append a timestamp.

maestrowf.utils.**get_duration**(*time_delta*)
Convert durations to HH:MM:SS format.

Params time_delta A time difference in datatime format.

Returns A formatted string in HH:MM:SS

maestrowf.utils.**make_safe_path**(*base_path*, **args*)
Construct a subpath that is path safe.

Params base_path The base path to append args to.

Params args Path components to join into a path.

Returns A joined subpath with invalid characters stripped.

maestrowf.utils.**ping_url**(*url*)
Load a webpage to test that it is accessible.

Parameters **url** – URL string to be loaded.

maestrowf.utils.**round_datetime_seconds**(*input_datetime*)
Round datetime to the nearest whole second.

Solution referenced from: https://stackoverflow.com/questions/47792242/ rounding-time-off-to-the-nearest-second-python.

Params input_datetime A datetime in datatime format.

Returns `input_datetime` rounded to the nearest whole second

maestrowf.utils.**start_process**(*cmd*, *cwd=None*, *env=None*, *shell=True*)
Start a new process using a specified command.

Parameters

- **cmd** – A string or a list representing the command to be run.
- **cwd** – Current working path that the process will be started in.
- **env** – A dictionary containing the environment the process will use.
- **shell** – Boolean that determines if the process will run a shell.

7.2 setup module

Used by autodoc_mock_imports.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

s

Index

A

B

C

D

E

F

G

J

K

L

M

N

O

P

Q

R

S

T

U

V

W