Sierra Documentation

John Harwell

May 13, 2023

CONTENTS:

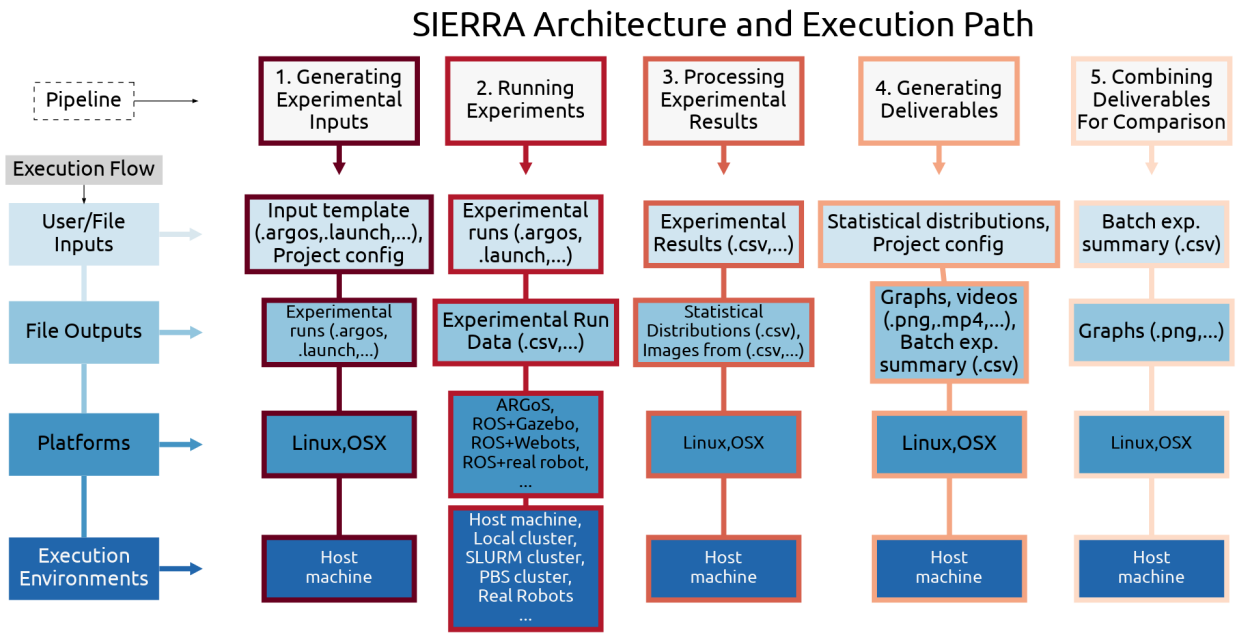


Fig. 1: SIERRA architecture, organized by pipeline stage. Stages are listed left to right, and an approximate joint architectural/functional stack is top to bottom for each stage. "…" indicates areas where SIERRA is designed via plugins to be easily extensible. "Host machine" indicates the machine SIERRA was invoked on.

SIERRA is a command line tool and plugin framework for:

- Automating scientific research, providing faculties for seamless experiment generation, execution, and results processing.
- Accelerating research cycles by allowing researchers to focus on the "science" aspects: developing new things and designing experiments to test them.
- Improving the reproducibility of scientific research, particular in AI.

It supports a wide range of platforms, execution environments, and experiment input/output formats–see *SIERRA Support Matrix* for details. A short overview of the research automation SIERRA provides is here: *SIERRA Pipeline*. To get started, see *Getting Started With SIERRA*.

CHAPTER

ONE

TRYING OUT SIERRA

If you just want to try SIERRA out with a pre-existing *Project* without first defining your own, the steps to do so below. I assume that all repositories are cloned into `$HOME/research`; adjust the paths accordingly if you clone things somewhere else.

1. Install OS packages (if you don't see your OS below you will have to find and install the equivalent packages).

   Ubuntu

   Install the following required packages with `apt install`:

   - `parallel`
   - `cm-super`
   - `texlive-fonts-recommended`
   - `texlive-latex-extra`
   - `dvipng`
   - `psmisc`

   Install the following optional packages with `apt install`:

   - `pssh`
   - `ffmpeg`
   - `xvfb`

   OSX

   Install the following required packages with `brew install`:

   - `parallel`
   - `--cask mactex`
   - `pssh`

   Install the following optional packages with `brew install`:

   - `--cask xquartz`
   - `pssh`
   - `ffmpeg`

   If you are on a different Linux distribution you will have to find and install the equivalent packages.

Important: SIERRA will not work correctly in all cases if the required packages (or their equivalent) are not installed! It may start, it might even not crash immediately depending on what you are using it to do. If you are missing an optional package for a feature you try to use, you will get an error.

2. Install SIERRA:

```
pip3 install sierra-research
```

3. Setup your chosen *Platform*:

   ARGoS

   Install *ARGoS* via your chosen method (from source or via .deb). Check that `argos3` is found by your shell.

   Important: If `argos3` is not found by your shell then you won't be able to do anything useful with SIERRA!

   ROS1+Gazebo

   1. Install ROS distribution by following one of (or an equivalent guide for OSX or an alternative linux distribution):

      - http://wiki.ros.org/noetic/Installation/Ubuntu
      - http://wiki.ros.org/noetic/Installation/Ubuntu

      SIERRA only supports kinetic,noetic.

   2. Install additional ROS packages for the turtlebot:

      - `ros-<distro>-turtlebot3-description`
      - `ros-<distro>-turtlebot3-msgs`
      - `ros-<distro>-turtlebot3-gazebo`
      - `ros-<distro>-turtlebot3-bringup`

      Where `<distro>` is replaced by your ROS distro.

   3. Install the SIERRA ROSBridge:

```
pip3 install catkin_tools
git clone https://github.com/jharwell/sierra_rosbridge.git
cd sierra_rosbridge
catkin init
catkin config --extend /opt/ros/<distro>
```

      Where `<distro>` is replaced by your ROS distro. Finally, set catkin to install at a common location (e.g., `$HOME/.local/ros/noetic`) and build the package:

```
catkin config --install -DCMAKE_INSTALL_PREFIX=$HOME/.local/ros/noetic
catkin build
```

4. Download and build the super-simple SIERRA sample project for your chosen *Platform*:

   ARGoS

   Based on the foraging example from the ARGoS website:

```
git clone https://github.com/jharwell/sierra-sample-project.git
cd sierra-sample-project/argos
git checkout devel
mkdir -p build && cd build
cmake -DARGOS_INSTALL_DIR=<path> ..
make
```

`ARGOS_INSTALL_DIR` should point to the directory you have installed the version of ARGoS you want to use for the trial (installed, not compiled!). This is used instead of the `FindARGoS()` cmake functionality to support having multiple versions of ARGoS installed in multiple directories.

ROS1+Gazebo

Based on one of the turtlebot3 intro tutorials:

```
git clone https://github.com/jharwell/sierra-sample-project.git
cd sierra-sample-project/ros1gazebo
git checkout devel
catkin init
catkin config --extend=$HOME/.local/ros/noetic
catkin build
```

Where `$HOME/.local/ros/noetic` is where I installed the SIERRA ROSBridge into.

5. Setup runtime environment:

   ARGoS

   1. Set *SIERRA_PLUGIN_PATH*:

      ```
      export SIERRA_PLUGIN_PATH=$HOME/research/sierra-sample-project/projects
      ```

   2. Set *ARGOS_PLUGIN_PATH*:

      ```
      export ARGOS_PLUGIN_PATH=$HOME/research/sierra-sample-project/argos/build:
      ↪<ARGOS_INSTALL_DIR>/lib/argos3
      ```

      Where <ARGOS_INSTALL_DIR> is the prefix that you installed ARGoS to.

   ROS1+Gazebo

   1. Set *SIERRA_PLUGIN_PATH*:

      ```
      export SIERRA_PLUGIN_PATH=$HOME/research/sierra-sample-project/projects/
      ↪ros1gazebo_project
      ```

   2. Source ROS environment to set *ROS_PACKAGE_PATH* (if you haven't already):

      ```
      . /path/to/setup.bash
      ```

6. Run SIERRA (invocation inspired by *SIERRA Usage By Example*). You can do this from any directory! (yay SIERRA!)

   ARGoS

   ```
   sierra-cli \
   --sierra-root=$HOME/research/exp \
   --template-input-file=exp/argos/template.argos \
   ```

(continues on next page)

(continued from previous page)

```
--n-runs=4 \
--platform=platform.argos \
--project=argos_project \
--physics-n-engines=1 \
--controller=foraging.footbot_foraging \
--scenario=LowBlockCount.10x10x1 \
--batch-criteria population_size.Log8 \
--with-robot-leds \
--with-robot-rab \
--exp-overwrite
```

This will run a batch of 4 experiments using the `argos_project.so` C++ library. The swarm size will be varied from 1..8, by powers of 2. Within each experiment, 4 copies of each simulation will be run (each with different random seeds), for a total of 16 ARGoS simulations. On a reasonable machine it should take about 1 minute or so to run. After it finishes, you can go to `$HOME/research/exp` and find all the simulation outputs, including camera ready graphs! For an explanation of SIERRA's runtime directory tree, see *SIERRA Runtime Directory Tree*. You can also run the same experiment again, and it will overwrite the previous one because you passed `--exp-overwrite`.

Note: The `--with-robot-rab` and `--with-robot-leds` arguments are required because robot controllers in the sample project use the RAB and LED sensor/actuators, and SIERRA strips those tags out of the robots <sensors> and <actuators> and <media> parent tags by default to increase speed and reduce the memory footprint of ARGoS simulations.

ROS1+Gazebo

```
sierra-cli \
--sierra-root=$HOME/research/exp \
--template-input-file=exp/ros1gazebo/turtlebot3_house.launch \
--n-runs=4 \
--platform=platform.ros1gazebo \
--project=ros1gazebo_project \
--controller=turtlebot3.wander \
--scenario=HouseWorld.10x10x1 \
--batch-criteria population_size.Log8 \
--robot turtlebot3 \
--exp-overwrite \
--pipeline 1 2
```

This will run a batch of 4 experiments. The swarm size will be varied from 1..8, by powers of 2. Within each experiment, 4 copies of each simulation will be run (each with different random seeds), for a total of 16 Gazebo simulations. Only the first two pipeline stages are run, because this controller does not produce any output. You can also run the same experiment again, and it will overwrite the previous one because you passed `--exp-overwrite`.

CHAPTER

TWO

GETTING STARTED WITH SIERRA

If you're looking for the "I just want to try out SIERRA without doing any work" quickstart, see *Trying Out SIERRA*. Otherwise, the steps to start using SIERRA are below.

2.1 Basic Setup

1. Browse through the SIERRA *Glossary* to get an overview of the terminology that SIERRA uses in the code and in the documentation–a little of this will go a long way towards quickly understanding how to get started and use SIERRA!

   Seriously–it will make the later steps make more sense.

2. Check requirements at *Requirements To Use SIERRA* to how to organize your experimental inputs/template input files so they can be used with SIERRA, and if you need to modify the way your code outputs data so that SIERRA can process it.

3. Install SIERRA

   - Install SIERRA packages by following *SIERRA Installation Reference*.
   - Install OS packages (if you don't see your OS below you will have to find and install the equivalent packages).

     Ubuntu

     Install the following required packages with `apt install`:

     - `parallel`
     - `cm-super`
     - `texlive-fonts-recommended`
     - `texlive-latex-extra`
     - `dvipng`
     - `psmisc`

     Install the following optional packages with `apt install`:

     - `pssh`
     - `ffmpeg`
     - `xvfb`

     OSX

     Install the following required packages with `brew install`:

- `parallel`
- `--cask mactex`
- `pssh`

Install the following optional packages with `brew install`:

- `--cask xquartz`
- `pssh`
- `ffmpeg`

If you are on a different Linux distribution you will have to find and install the equivalent packages.

Important: SIERRA will not work correctly in all cases if the required packages (or their equivalent) are not installed! It may start, it might even not crash immediately depending on what you are using it to do. If you are missing an optional package for a feature you try to use, you will get an error.

2.2 Project Plugin Setup

Now that you have some sense of what SIERRA is/how it works, you can define a *Project* plugin to interface between your code and SIERRA. The steps to do so are:

1. Select which *Platform* SIERRA should target (what platform it targets will partially define how you create your project plugin). See *Platform Plugins* for supported platforms. If your desired platform is not in the list, never fear! It's easy to create a new platform plugin, see *Creating a New Platform Plugin*.
2. Setup the interface between your code and SIERRA by defining a SIERRA by following *Creating a New SIERRA Project*.
3. Select an execution environment for SIERRA that matches your available computational resources: *HPC Execution Environment Plugins* or *Real Robot Execution Environment Plugins*, following the appropriate setup guide. If there is nothing suitable, never fear! It's easy to create a new execution environment plugin, see *Creating a New Execution Environment Plugin*.

2.3 Usage Setup

Now that you have created your project plugin, you are ready to start using SIERRA! The steps to do so are:

1. Decide what variable you are interested in investigating by consulting the *Batch Criteria* available for your project (i.e., what variable(s) you want to change across some range and see how system behavior changes, or doesn't change). Which criteria are available to use depends on your *Platform*; if you don't see something suitable, you can *Define A New Batch Criteria*.
2. Look at the *Command Line Interface* to understand how to invoke SIERRA in general.
3. Look at the *SIERRA Usage By Example* to get ideas on how to craft your own SIERRA invocation. If you get stuck, look at *FAQ* for answers to common questions.
4. Determine how to invoke SIERRA. At a minimum you need to tell it the following:
   - What platform you are targeting/want to run on: `--platform`. See *Platform Plugins* for details.
   - What project to load: `--project`. This is used to:
     - Configure runtime search paths (e.g., *ARGOS_PLUGIN_PATH*, *ROS_PACKAGE_PATH*).

  - Figure out the directory to load graph and *Experiment* data processing configuration from.

- What template input file to use: `--template-input-file`. See *Template Input Files* for requirements.
- How many variations of the main settings for each experiment to run: `--n-runs`.
- Where it is running/how to run experiments: `--exec-env`. See *HPC Execution Environment Plugins* for available plugins.
- What controller to run: `--controller`. See *Main Configuration* for details on how valid controllers are defined for a *Project*. Project dependent.
- How large the arena should be, what block distribution type to use (for example), etc. `--scenario`. Project dependent.
- What you are investigating; that is, what variable are you interested in varying: `--batch-criteria`.

If you try to invoke SIERRA with an (obviously) incorrect combination of command line options, it will refuse to do anything. For less obviously incorrect combinations, it will (hopefully) stop when an assert fails before doing anything substantial.

Full documentation of all command line options it accepts is in *Command Line Interface*, and there are many useful options that SIERRA accepts, so skimming the CLI docs is **very** worthwhile.

Important: Generally speaking, do not try to run SIERRA on HPC environments with a debug build of whatever project you are using. It will work but be obnoxiously/irritatingly slow. SIERRA is intended for *production* code (well, as close to production as research code gets) which is compiled with optimizations enabled.

---

5. Setup the cmdline environment you are going to invoke SIERRA in.

   - Set *SIERRA_PLUGIN_PATH* appropriately.

   Different platforms may require additional environments to be set.

6. Learn SIERRA's runtime *SIERRA Runtime Directory Tree*. When running, SIERRA will create a (rather) large directory structure for you, so reading the docs is worthwhile to understand what the structure means, and to gain intuition into where to look for the inputs/outputs of different stages, etc., without having to search exhaustively through the filesystem.
7. Invoke SIERRA! Again, look at the *SIERRA Usage By Example* for some ideas.

CHAPTER

THREE

REQUIREMENTS TO USE SIERRA

This page details the parameters you must meet in order to be able to use SIERRA in a more or less out-of-the-box fashion. Because SIERRA is highly modular, use cases which don't meet one or more of the parameters described below can likely still be accommodated with an appropriate plugin.

3.1 OS Requirements

One of the following:

- Recent linux. SIERRA is tested with Ubuntu 20.04+, though it will probably work on less recent versions/other distros as well.
- Recent OSX. SIERRA is tested with OSX 12+, though it *might* work on less recent versions.

Note: Windows is not supported currently. Not because it can't be supported, but because there are not current any platform plugins that which work on windows. SIERRA is written in pure python, and could be made to work on windows with a little work.

If windows support would be helpful for your intended usage of SIERRA, please get in touch with me.

3.2 Python Requirements

Python 3.8+. Tested with 3.8-3.9. It may work for newer versions, probably won't for older.

3.3 Experimental Definition Requirements

3.3.1 General

Experimental Runs within each *Experiment* are entirely defined by the contents of the `--template-input-file` (which is modified by SIERRA before being written out as one or more input XML files), so SIERRA restricts the content of this file in a few modest ways.

1. Experimental inputs are specified as a single XML file (`--template-input-file`); SIERRA uses this to generate multiple XML input files defining *Experiments*. If your experiments use require multiple XML input files, you will either have to consolidate them all into a single file, or point SIERRA to the "main" file in order to use SIERRA to generate experiments from some portion of your experimental definitions. As a result of this restriction, all experiments must read their definition from XML.

XML was chosen over other input formats because:

- It is not dependent on whitespace/tab/spaces for correctness, making it more robust to multiple platforms, simulators, parsers, users, etc.
- Mature manipulation libraries exist for python and C++, so it should be relatively straightforward for projects to read experimental definitions from XML.

2. No reserved XML tokens are present. See *XML Content Requirements* for details.
3. All experiments from which you want to generate statistics/graphs are:
   - The same length
   - Capture the same number of datapoints

   That is, experiments always obey `--exp-setup`, regardless if early stopping conditions are met. For *ROS1* platforms, the SIERRA timekeeper ensures that all experiments are the same length; it is up to you to make sure all experiments capture the same # of data points. For other *Platforms* (e.g., *ARGoS*) it is up to you to ensure both conditions.

   This is a necessary restriction for deterministic and non-surprising statistics generation during stage 3. Pandas (the underlying data processing library) can of course handle averaging/calculating statistics from dataframes of different sizes (corresponding to experiments which had different lengths), but the generated statistics may not be as reliable/meaningful. For example, if you are interested in the steady state behavior of the system, then you might want to use the *last* datapoint in a given column as a performance measure, averaged across all experiments. If not all experiments have the same # of datapoints/same length, then the resulting confidence intervals around the average value (for example) may be larger.

   If you need to "stop" an experiment early, simply tell all agents/robots to stop moving/stop doing stuff once the stopping conditions have been met and continue to collect data as you normally would until the end of the experiment.

   If you do **NOT** want to use SIERRA to generate statistics/graphs from experimental data (e.g., you want to use it to capture videos only), then you can ignore this requirement.

Platforms may have additional experiment requirements, as shown below.

3.3.2 ARGoS Platform

1. All swarms are homogeneous (i.e., only contain 1 type of robot) if the size of the swarm changes across experiments (e.g., 1 robot in exp0, 2 in exp1, etc.). While SIERRA does not currently support multiple types of robots with varying swarm sizes, adding support for doing so would not be difficult. As a result, SIERRA assumes that the type of the robots you want to use is already set in the template input file (e.g., `<entity/foot-bot>`) when using SIERRA to change the swarm size.
2. The distribution method via `<distribute>` in the `.argos` file is the same for all robots, and therefore only one such tag exists (not checked).
3. The `<XXX_controller>` tag representing the configuration for the `--controller` you want to use does not exist verbatim in the `--template-input-file`. Instead, a placeholder `__CONTROLLER__` is used so that SIERRA can unambiguously set the "library" attribute of the controller; the `__CONTROLLER__` tag will replaced with the ARGoS name of the controller you selected via `--controller` specified in the `controllers.yaml` configuration file by SIERRA. You should have something like this in your template input file:

```
<argos-configuration>
   ...
   <controllers>
      ...
```

(continues on next page)

(continued from previous page)

```
      <__CONTROLLER__>
         <param_set1>
            ...
         </param_set1>
         ...
      <__CONTROLLER__/>
      ...
   </controllers>
   ...
</argos-configuration>
```

See also *Main Configuration*.

3.3.3 ROS1-based Platforms

These requirements apply to any *Platform* which uses *ROS1* (e.g., *ROS1+Gazebo*, *ROS1+Robot*).

1. All robot systems are homogeneous (i.e., only contain 1 type of robot). While SIERRA does not currently support multiple types of robots in ROS, adding support for doing so would not be difficult.

2. Since SIERRA operates on a single template input file (`--template-input-file`) when generating experimental definitions, all XML parameters you want to be able to modify with SIERRA must be present in a single `.launch` file. Other parameters you don't want to modify with SIERRA can be present in other `.launch` or `.world` files, and using the usual <include> mechanism. See also *SIERRA Design Philosophy*.

3. Within the template `.launch` file (`--template-input-file`), the root XML tag must be <ros-configuration> . The <ros-configuration> tag is stripped out by SIERRA during generation, and exists solely for the purposes of conformance with the XML standard, which states that there can be only a single root element (i.e., you can't have a <params> element and a <launch> element both at the root level–see options below). See *Template Input Files* for details of required structure of passed `--template-input-file`, and what changes are applied to them by SIERRA to use with ROS.

   Projects can choose either of the following options for specifying controller parameters. See *Template Input Files* for further details of required structure of passed `--template-input-file`, and what changes are applied to them by SIERRA to use with ROS, depending on the option chosen.

   - Use the ROS Parameter Server

     All parameters are specified as you would expect under <launch>.

     > **Warning:** Using the ROS parameter server is generally discouraged for projects which have LOTS of parameters, because manipulating the XML becomes non-trivial, and can require extensive XPath knowledge (e.g., `//launch/group/[@ns='{ns}']`). For smaller projects it's generally fine.

   - Use the <params> tag under <ros-configuration> to specify an XML tree of controller parameters.

     This is recommended for large projects, as it allows cleaner XPath specifications (e.g., `.//params/task_alloc/mymethod/threshold`), and for those which use *ARGoS* for simulations and a ROS platform for real robots, as it maximizes code reuse. During stage 1 the modified <params> sub-tree is removed from the written `.launch` file if it exists and written to a *different* file in the same directory as the `.launch` file.

     All SIERRA configuration exposed via XML parameters uses the ROS parameter server. See *Template Input Files* for specifics.

4. ROS does not currently provide a way to shut down after a given # of iterations/timesteps, so SIERRA provides a ROS package with a node tracking the elapsed time in seconds, and which exits (and takes down the roslaunch when it does) after the specified experiment time has elapsed. This node is inserted into all `.launch` files. All ROS projects must depend on this ROS bridge package so the necessary nodes can be found by ROS at runtime.

3.4 Additional Platform Requirements

3.4.1 ROS1+Robot Platform

1. All data from multiple robots somehow ends up accessible through the filesystem on the host machine SIERRA is invoked on, as if the same experimental run was locally with a simulator. There are several ways to accomplish this:

   - Use SIERRA's ability to configure a "master" node on the host machine, and then setup streaming of robot data via ROS messages to this master node. Received data is processed as appropriate and then written out to the local filesystem so that it is ready for statistics generation during stage 3.

     Important: If you use this method, then you will need to handle robots starting execution at slightly different times in your code via (a) a start barrier triggered from the master node, or else timestamp the data from robots and marshal it on the master node in some fashion. The *SIERRA ROSBridge* provides some support for (a).

   - Mount a shared directory on each robot where it can write its data, and then after execution finishes but before your code exits you process the per-robot data if needed so it is ready for statistics generation during stage 3.
   - Record some/all messages sent and received via one or more ROSbag files, and then post-process these files into a set of dataframes which are written out to the local filesystem.
   - Record some/all messages sent and received via one or more ROSbag files, and use these files directly as a "database" to query during stage 3. This would require writing a SIERRA storage plugin (see *Creating a New Storage Plugin*).

     Important: This method requires that whatever is recorded into the ROSbag file is per-run, not per-robot; that is, if a given data source somehow built from messages sent from multiple robots, those messages need to be processed/averaged/etc and then sent to a dedicated topic to be recorded.

3.5 Requirements For Project Code

3.5.1 General

SIERRA makes a few assumptions about how *Experimental Runs* using your C/C++ library can be launched, and how they output data. If your code does not meet these assumptions, then you will need to make some (hopefully minor) modifications to it before you can use it with SIERRA.

1. Project code uses a configurable random seed. While this is not technically *required* for use with SIERRA, all research code should do this for reproducibility. See *Platform Plugins* for platform-specific details about random seeding and usage with SIERRA.

2. *Experimental Runs* can be launched from *any* directory; that is, they do not require to be launched from the root of the code repository (for example).

3. All outputs for a single *Experimental Run* will reside in a subdirectory in the directory that the run is launched from. For example, if a run is launched from `$HOME/exp/research/simulations/sim1`, then its outputs need to appear in a directory such as `$HOME/exp/research/simulations/sim1/outputs`. The directory within the experimental run root which SIERRA looks for simulation outputs is configured via YAML; see *Main Configuration* for details.

   For HPC execution environments (see *HPC Execution Environment Plugins*), this requirement is easy to meet. For real robot execution environments (see *Real Robot Execution Environment Plugins*), this can be more difficult to meet.

4. All experimental run outputs are in a format that SIERRA understands within the output directory for the run. See *Storage Plugins* for which output formats are currently understood by SIERRA. If your output format is not in the list, never fear! It's easy to create a new storage plugin, see *Creating a New Storage Plugin*.

3.5.2 ARGoS Platform

1. `--project` matches the name of the C++ library for the project (i.e. `--project.so`), unless `library_name` is present in `sierra.main.run` YAML config. See *Main Configuration* for details. For example if you pass `--project=project-awesome`, then SIERRA will tell ARGoS to search in `proj-awesome.so` for both loop function and controller definitions via XML changes, unless you specify otherwise in project configuration. You *cannot* put the loop function/controller definitions in different libraries.

2. *ARGOS_PLUGIN_PATH* is set up properly prior to invoking SIERRA.

3.5.3 ROS1+Gazebo Project Platform

1. *ROS_PACKAGE_PATH* is set up properly prior to invoking SIERRA.

3.5.4 ROS1+Robot Platform

1. *ROS_PACKAGE_PATH* is set up properly prior to invoking SIERRA on the local machine AND all robots are setup such that it is populated on login (e.g., an appropriate `setup.bash` is sourced in `.bashrc`).

2. All robots have `ROS_IP` or `ROS_HOSTNAME` populated, or otherwise can correctly report their address to the ROS master. You can verify this by trying to launch a ROS master on each robot: if it launches without errors, then these values are setup properly.

CHAPTER

FOUR

MODEL FRAMEWORK REQUIREMENTS

When running models during stage 4 (see *Adding Models to your SIERRA Project*) SIERRA requires that:

- All models return `pandas.DataFrame` (if they don't do this natively, then their python bindings will have to do it). This is enforced by the interfaces models must implement.

4.1 XML Content Requirements

4.1.1 Reserved Tokens

SIERRA uses some special XML tokens during stage 1, and although it is unlikely that including these tokens would cause problems, because SIERRA looks for them in *specific* places in the `--template-input-file`, they should be avoided.

- `__CONTROLLER__` - Tag used when as a placeholder for selecting which controller present in an input file (if there are multiple) a user wants to use for a specific *Experiment*. Can appear in XML attributes. This makes auto-population of the controller name based on the `--controller` argument and the contents of `controllers.yaml` (see *Main Configuration* for details) in template input files possible.
- `__UUID__` - XPath substitution optionally used when a *ROS1* platform is selected in `controllers.yaml` (see *Main Configuration*) when adding XML tags to force addition of the tag once for every robot in the experiment, with `__UUID__` replaced with the configured robot prefix concatenated with its numeric ID (0-based). Can appear in XML attributes.
- `sierra` - Used when the *ROS1+Gazebo* platform is selected. Should not appear in XML tags or attributes.

CHAPTER

FIVE

SIERRA SUPPORT MATRIX

SIERRA supports multiple *Platforms* which researchers can write code to target, as shown below. If your desired platform is not listed, see the *Configuration and Extension Tutorials* for how to add it via a plugin.

Important: In SIERRA terminology, platform != OS. If a SIERRA platform runs on a given OS, then SIERRA supports doing so; if it does not, then SIERRA does not. For example, SIERRA does not support running ARGoS on windows, because ARGoS does not support windows.

Platform	Description
ARGoS	Simulator for fast simulation of large swarms. Requires ARGoS >= 3.0.0-beta59.
ROS1	Using ROS1 with the Gazebo simulator. Requires Gazebo >= 11.9.0, ROS1 Noetic or later.
ROS1+Robot	Using ROS1 with a real robot platform of your choice. ROS1 Noetic or later is required.

SIERRA supports multiple HPC environments for execution of experiments in on HPC hardware (see *HPC Execution Environment Plugins*) and on real robots (see *Real Robot Execution Environment Plugins*). If your desired execution environment is not listed, see the *Configuration and Extension Tutorials* for how to add it via a plugin.

Execution Environment	Description
SLURM	An HPC cluster managed by the SLURM scheduler
Torque/MOAB	An HPC cluster managed by the Torque/MOAB scheduler
Adhoc	Miscellaneous collection of networked HPC compute nodes or random servers; not managed by a scheduler
HPC local	The SIERRA host machine,e.g., a researcher's laptop
Turtlebot3	Real turtlebot3 robots

Platform	Description
ARGoS	Simulator for fast simulation of large swarms. Requires ARGoS >= 3.0.0-beta59.
ROS1	Using ROS1 with the Gazebo simulator. Requires Gazebo >= 11.9.0, ROS1 Kinetic or later.
ROS1+Robot	Using ROS1 with a real robot platform of your choice. ROS1 Kinetic or later is required.

SIERRA also supports multiple output formats for experimental outputs. If the format for your experimental outputs

is not listed, see the docs for how to add it via a plugin. SIERRA currently only supports XML experimental inputs.

Experimental Output Format	Scope
CSV file	Raw experimental outputs, transforming into heatmap images
PNG file	Stitching images together into videos

SIERRA supports (mostly) mix-and-match between platforms, execution environments, experiment input/output formats as shown in its support matrix below. This is one of the most powerful features of SIERRA!

Execution Environment	Platform	Experimental Input Format	Experimental Output Format
SLURM	ARGoS, ROS1+Gazebo	XML	CSV, PNG
Torque/MOAB	ARGoS, ROS1+Gazebo	XML	CSV, PNG
ADHOC	ARGoS, ROS1+Gazebo	XML	CSV, PNG
Local	ARGoS, ROS1+Gazebo	XML	CSV, PNG
ROS1+Turtlebot3	ROS1+Gazebo, ROS1+robot	XML	CSV, PNG

CHAPTER

SIX

SIERRA USAGE BY EXAMPLE

This page contains reference examples of SIERRA usage to help you craft your own SIERRA invocation. These examples use the SIERRA project plugins from the SIERRA sample project repo: https://github.com:jharwell/sierra-sample-project.git. See *Trying Out SIERRA* or *Getting Started With SIERRA* for setting up the sample project repository.

In all examples:

- `$HOME/git/mycode` contains the ARGoS C++ library
- `$HOME/git/sierra-sample-project/projects` contains the SIERRA project plugin
- `$HOME/git/sierra-sample-project/projects` is on *SIERRA_PLUGIN_PATH*.

If your setup is different, adjust paths in the commands below as needed.

Important: The examples are grouped by platform, so they can be pasted into the terminal and executed directly. However, parts of many commands use common functionality in the SIERRA core; just because you don't see stage5 examples for the ROS1+Gazebo platform doesn't mean you can't run stage5 with that platform. Non-uniformities in which commands are below are more a limitation of the sample project than SIERRA itself.

6.1 ARGoS Examples

6.1.1 Basic Example

This example illustrates the simplest way to use ARGoS with SIERRA to run experiments.

```
sierra-cli \
--sierra-root=$HOME/exp \
--template-input-file=exp/your-experiment.argos \
--n-runs=3 \
--platform=platform.argos\
--project=argos_project \
--exec-env=hpc.local \
--physics-n-engines=1 \
--exp-setup=exp_setup.T10000 \
--controller=foraging.footbot_foraging\
--scenario=LowBlockCount.10x10x2 \
--batch-criteria population_size.Log64
```

This will run a batch of 7 experiments using a correlated random walk robot controller (CRW), across which the swarm size will be varied from 1..64, by powers of 2. Experiments will all be run in the same scenario: `LowBlockCount` in a 10x10x2 arena; the meaning of `LowBlockCount` is entirely up to the project. In this case, we can infer it has to do with the # of blocks available for robots to forage for. In fact, whatever is passed to `--scenario` is totally arbitrary from SIERRA's point of view.

Within each experiment, 3 copies of each simulation will be run (each with different random seeds), for a total of 21 ARGoS simulations. On a reasonable machine it should take about 10 minutes or so to run. After it finishes, you can go to `$HOME/exp` and find all the simulation outputs. For an explanation of SIERRA's runtime directory tree, see *SIERRA Runtime Directory Tree*.

6.1.2 HPC Example

In order to run on a SLURM managed cluster, you need to invoke SIERRA within a script submitted with `sbatch`, or via `srun` with the correspond cmdline options set.

```
#!/bin/bash -l
#SBATCH --time=01:00:00
#SBATCH --nodes 10
#SBATCH --tasks-per-node=6
#SBATCH --cpus-per-task=4
#SBATCH --mem-per-cpu=2G
#SBATCH --output=R-%x.%j.out
#SBATCH --error=R-%x.%j.err
#SBATCH -J argos-slurm-example

# setup environment
export ARGOS_INSTALL_PREFIX=/$HOME/.local
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ARGOS_INSTALL_PREFIX/lib/argos3
export ARGOS_PLUGIN_PATH=$ARGOS_INSTALL_PREFIX/lib/argos3:$HOME/git/mycode
export SIERRA_PLUGIN_PATH=$HOME/git/sierra-projects
export PARALLEL="--env ARGOS_PLUGIN_PATH --env LD_LIBRARY_PATH"

sierra-cli \
--sierra-root=$HOME/exp \
--template-input-file=exp/your-experiment.argos \
--n-runs=96 \
--platform=platform.argos\
--project=argos_project \
--exec-env=hpc.slurm \
--exp-setup=exp_setup.T10000 \
--controller=foraging.footbot_foraging \
--scenario=LowBlockCount.10x10x2 \
--batch-criteria population_size.Log64
```

In this example, the user requests 10 nodes with 24 cores each, and wants to run ARGoS with 4 physics engines (4 * 6 = 24), with 8GB memory per core. Note that we don't pass `--physics-n-engines` – SIERRA computes this from the SLURM parameters. SIERRA will run each of the 96 simulations per experiment in parallel, 6 at a time on each allocated node. Each simulation will be 10,000 seconds long and use `LowBlockCount` scenario in a 10x10x2 arena, as in the previous example.

Important: You need to export *`PARALLEL`* containing all necessary environment variables your code uses in addition to those needed by SIERRA before invoking it, otherwise some of them might not be transferred to the SLURM job

and/or the new shell GNU parallel starts each simulation in.

Note that if you compile ARGoS for different architectures within the same HPC environment, you can use a combination of conditionally setting *ARGOS_PLUGIN_PATH* with setting *SIERRA_ARCH* to some string to tell SIERRA to use a given version of ARGoS, depending on where you request resources from. For example, you could set `SIERRA_ARCH=x86` or `SIERRA_ARCH=arm` to link to an `argos3-x86` or `argos3-arm` executable and libraries, respectively.

6.1.3 Rendering Example

This example shows how to use ARGoS image capturing ability to create nice videos of simulations.

```
sierra-cli \
--sierra-root=$HOME/exp \
--template-input-file=exp/your-experiment.argos \
--platform=platform.argos\
--project=argos_project \
--controller=foraging.footbot_foraging \
--scenario=LowBlockCount.10x10x2 \
--exec-env=hpc.local \
--n-runs=3 \
--platform-vc \
--exp-graphs=none \
--physics-n-engines=1 \
--batch-criteria population_size.Log8
```

The runs 3 simulations in parallel with 1 physics engine each, and runs ARGoS under **Xvfb** to get it to render headless images. During stage 4, these images are stitched together using **ffmpeg** to create videos (see *SIERRA Runtime Directory Tree* for where the videos will appear). No graphs are generated during stage 4 in this example.

You may also be interested in the `--camera-config` option, which allows you to specify different static/dynamic camera arrangements (e.g., do a nice circular pan around the arena during simulation).

Note: Because LOTS of images can be captured by ARGoS to create videos, depending on simulation length, you usually want to have a very small `--n-runs` to avoid filling up the filesystem.

6.1.4 Bivariate Batch Criteria Example

This example shows how to use ARGoS with a bivariate batch criteria (i.e., with TWO variables/things you want to vary jointly):

```
::
```

sierra-cli –sierra-root=$HOME/exp –template-input-file=exp/your-experiment.argos –platform=platform.argos–project=argos_project –controller=foraging.footbot_foraging –scenario=LowBlockCount.10x10x2 –exec-env=hpc.local –n-runs=3 –platform-vc –exp-graphs=none –physics-n-engines=1 –batch-criteria population_size.Log8 max_speed.1.9.C5

The `max_speed.1.9.C5` is a batch criteria defined in the sample project, and corresponds to setting the maximum robot speed from 1…9 to make 5 experiments; i.e., 1,3,5,7,9. It can also be used on its own–just remove the first `population_size` batch criteria from the command to get a univariate example.

The generated experiments form a grid: population size on the X axis and max speed on the Y, for a total of 3 * 5 = 15 experiments. If the order of the batch criteria is switched, then so is which criteria/variable is on the X/Y axis. Experiments are run in sequence just as with univariate batch criteria. During stage 3/4, by default SIERRA generates discrete a set of heatmaps, one per capture interval of simulated time, because the experiment space is 2D instead of 1D, and you can't easily represent time AND two variables + time on a plot. This can take a loooonnnggg time, and can be disabled with `--project-no-HM`.

The generated sequence of heatmaps can be turned into a video–pass `--bc-rendering` during stage 4 to do so.

6.1.5 Stage 5 Scenario Comparison Example

This example shows how to run stage 5 to compare a single controller across different scenarios, assuming that stages 1-4 have been run successfully. Note that this stage does not require you to input the `--scenario`, or the `--batch-criteria`; SIERRA figures these out for you from the `--controller` and `--sierra-root`.

```
sierra-cli \
--sierra-root=$HOME/exp \
--project=argos_project \
--pipeline 5 \
--scenario-comparison \
--dist-stats=conf95 \
--bc-univar \
--controller=foraging.footbot_foraging \
--sierra-root=$HOME/exp
```

This will compare all scenarios that the `foraging.footbot_foraging` controller has been run on according to the configuration defined in `stage5.yaml`. SIERRA will plot the 95% confidence intervals on all generated graphs for the univariate batch criteria (whatever it was). If multiple batch criterias were used with this controller in the same scenario, SIERRA will process all of them and generate unique graphs for each scenario+criteria combination that the `foraging.footbot_foraging` controller was run on.

6.1.6 Stage 5 Controller Comparison Example

This example shows how to run stage 5 to compare multiple controllers in a single scenario, assuming that stages 1-4 have been run successfully. Note that this stage does not require you to input `--batch-criteria`; SIERRA figures these out for you from the `--controller-list` and `--sierra-root`.

```
sierra-cli \
--sierra-root=$HOME/exp \
--project=argos_project \
--pipeline 5 \
--controller-comparison \
--dist-stats=conf95 \
--bc-univar \
--controllers-list=foraging.footbot_foraging,foraging.footbot_foraging-slow \
--sierra-root=$HOME/exp
```

SIERRA will compute the list of scenarios that the `foraging.footbot_foraging` and the `foraging.footbot_foraging_slow` controllers have *all* been run. Comparison graphs for each scenario with the `foraging.footbot_foraging,foraging.footbot_foraging_slow` controllers will be generated according to the configuration defined in `stage5.yaml`. SIERRA will plot the 95% confidence intervals on all generated graphs for the univariate batch criteria (whatever it was). If multiple batch criterias were used with each controller in the same scenario, SIERRA

will process all of them and generate unique graphs for each scenario+criteria combination both controllers were run on.

6.2 ROS1+Gazebo Examples

6.2.1 Basic Example

This examples shows the simplest way to use SIERRA with the ROS1+gazebo platform plugin:

```
sierra-cli \
--platform=platform.ros1gazebo \
--project=ros1gazebo_project \
--n-runs=4 \
--exec-env=hpc.local \
--template-input-file=exp/your-experiment.launch \
--scenario=HouseWorld.10x10x1 \
--sierra-root=$HOME/exp/test \
--batch-criteria population_size.Log8 \
--controller=turtlebot3_sim.wander \
--exp-overwrite \
--exp-setup=exp_setup.T10 \
--robot turtlebot3
```

This will run a batch of 4 experiments using a correlated random walk controller (CRW) on the turtlebot3. Population size will be varied from 1..8, by powers of 2. Within each experiment, 4 copies of each simulation will be run (each with different random seeds), for a total of 16 Gazebo simulations. Each experimental run will be will be 10 seconds of simulated time. On a reasonable machine it should take about 10 minutes or so to run. After it finishes, you can go to `$HOME/exp` and find all the simulation outputs. For an explanation of SIERRA's runtime directory tree, see *SIERRA Runtime Directory Tree*.

6.2.2 HPC Example

In order to run on a SLURM managed cluster, you need to invoke SIERRA within a script submitted with `sbatch`, or via `srun` with the correspond cmdline options set.

```
#!/bin/bash -l
#SBATCH --time=01:00:00
#SBATCH --nodes 4
#SBATCH --tasks-per-node=6
#SBATCH --cpus-per-task=4
#SBATCH --mem-per-cpu=2G
#SBATCH --output=R-%x.%j.out
#SBATCH --error=R-%x.%j.err
#SBATCH -J ros1gazebo-slurm-example

# setup environment
export SIERRA_PLUGIN_PATH=$HOME/git/sierra-projects

sierra-cli \
--platform=platform.ros1gazebo \
--project=ros1gazebo_project \
```

(continues on next page)

(continued from previous page)

```
--n-runs=96 \
--exec-env=hpc.slurm \
--template-input-file=exp/your-experiment.launch \
--scenario=HouseWorld.10x10x1 \
--sierra-root=$HOME/exp/test \
--batch-criteria population_size.Log8 \
--controller=turtlebot3_sim.wander \
--exp-overwrite \
--exp-setup=exp_setup.T10000 \
--robot turtlebot3
```

In this example, the user requests 10 nodes with 24 cores each. SIERRA will run each of the 96 runs in parallel, 24 at a time on each allocated node. Each simulation will be 1,000 seconds long and use same scenario as before.

Important: You need to export *PARALLEL* containing all necessary environment variables your code uses in addition to those needed by SIERRA before invoking it, otherwise some of them might not be transferred to the SLURM job and/or the new shell GNU parallel starts each simulation in.

6.2.3 Bivariate Batch Criteria Example

This example shows how to use ROS1+gazebo with a bivariate batch criteria (i.e., with TWO variables/things you want to vary jointly):

```
::
```

> sierra-cli –sierra-root=$HOME/exp –template-input-file=exp/your-experiment.argos –platform=platform.ros1gazebo–project=ros1gazebo_project –controller=turtlebot3_sim.wander –scenario=HouseWorld.10x10x2 –exec-env=hpc.local –n-runs=3 –exp-graphs=none –batch-criteria population_size.Log8 max_speed.1.9.C5

The `max_speed.1.9.C5` is a batch criteria defined in the sample project, and corresponds to setting the maximum robot speed from 1...9 to make 5 experiments; i.e., 1,3,5,7,9. It can also be used on its own–just remove the first `population_size` batch criteria from the command to get a univariate example.

The generated experiments form a grid: population size on the X axis and max speed on the Y, for a total of 3 * 5 = 15 experiments. If the order of the batch criteria is switched, then so is which criteria/variable is on the X/Y axis. Experiments are run in sequence just as with univariate batch criteria. During stage 3/4, by default SIERRA generates discrete heatmaps of results instead of linegraphs, because the experiment space is 2D instead of 1D.

6.3 ROS1+Robot Examples

6.3.1 Basic Example

This examples shows the simplest way to use SIERRA with the ROS1+robot platform plugin:

```
::
```

> sierra-cli –platform=platform.ros1robot –project=ros1robot_project –n-runs=4 –template-input-file=exp/your-experiment.launch –scenario=OutdoorWorld.16x16x2 –sierra-root=$HOME/exp/test

–batch-criteria population_size.Linear6.C6 –controller=turtlebot3.wander –robot turtlebot3 –exp-setup=exp_setup.T100 –exec-env=robot.turtlebot3 –nodefile=turtlebots.txt –exec-inter-run-pause=60 –no-master-node

This will run a batch of 4 experiments using a correlated random walk controller (CRW) on the turtlebot3. Population size will be varied from 1,2,3,4,5,6. Within each experiment, 4 experimental runs will be conducted with each swarm size. SIERRA will pause for 60 seconds between runs so you can reset the robot's positions and environment before continuing with the next run. `turtlebots3.txt` contains the IP addresses of all 6 robots in the swarm (SIERRA may use different combinations of these if the swarm size is < 6). You could also omit `--nodefile` and set *SIERRA_NODEFILE* instead.

For these experiments, no master node is needed, so it is disabled. After all runs have completed and SIERRA finishes stages 3 and 4, you can go to `$HOME/exp` and find all the simulation outputs. For an explanation of SIERRA's runtime directory tree, see *SIERRA Runtime Directory Tree*.

CHAPTER

SEVEN

SIERRA OVERVIEW: A PRACTICAL SUMMARY

This page contains more in-depth documentation on how SIERRA works beyond the minimum needed to get started, and assumes that you have either gotten SIERRA working via *Trying Out SIERRA* or *Getting Started With SIERRA*. Once you've done that, the docs on this page provide a good way to understand how to get the most out of SIERRA.

7.1 SIERRA Pipeline

When invoked SIERRA will run one or more stages of its execution path, as specified via `--pipeline` on the cmdline. Only the first 4 pipeline stages will run by default. The pipeline stages are:

7.1.1 Stage 1: Experiment Generation

Experiments using the scientific method have an independent variable whose impact on results are measured through a series of trials. SIERRA allows you to express this as a research query on the command line, and then parses your query to make changes to a template input file to generate launch commands and experimental inputs to operationalize it. Switching from targeting platform A (e.g., ARGoS) to platform B (e.g., ROS1+Gazebo) is as easy as changing a single command line argument. SIERRA handles the "backend" aspects of defining experimental inputs allowing you to focus on their *content*, rather than the mechanics of how to turn the content into something that can be executed. See also:

- *SIERRA Support Matrix*
- *Command Line Interface*

Part of default pipeline.

7.1.2 Stage 2: Experiment Execution

SIERRA runs a previously generated *Batch Experiment*. Exactly which batch experiment SIERRA runs is determined by:

- `--controller`
- `--scenario`
- `--sierra-root`
- `--template-input-file`
- `--batch-criteria`

SIERRA currently supports two types of execution environments: simulators and real robots, which are handled seamlessly. For simulators, SIERRA will run multiple experimental runs from each experiment in parallel if possible. Similar to stage 1, switching between execution environments is as easy as changing a single command line argument. See also:

- *Execution Environment Plugins*
- *Platform Plugins*
- *SIERRA Support Matrix*

Part of default pipeline.

7.1.3 Stage 3: Experiment Post-Processing

SIERRA supports a number of data formats which simulations/real robot experiments can output their data, e.g., the number of robots engaged in a given task over time for processing. SIERRA post-processes experimental results after running the batch experiment; some parts of this can be done in parallel. This includes one or more of:

- Computing statistics over/about experimental data for stage 4 for use in graph generation in stage 4. See *Command Line Interface* documentation for `--dist-stats` for details.
- Creating images from project CSV files for rendering in stage 4. See *Project Rendering* for details.

Part of default pipeline.

7.1.4 Stage 4: Deliverable Generation

SIERRA can generate many deliverables from the processed experimental results automatically (independent of the platform/execution environment!), thus greatly simplifying reproduction of previous results if you need to tweak a given graph (for example). SIERRA currently supports generating the following deliverables:

- Camera-ready linegraphs, heatmaps, 3D surfaces, and scatterplots directly from averaged/statistically processed experimental data using matplotlib.
- Videos built from frames captured during simulation or real robot operation.
- Videos built from captured experimental output .csv files.

SIERRA also has a rich model framework allowing you to run arbitrary models, generate data, and plot it on the same figure as empirical results, automatically. See *Adding Models to your SIERRA Project* for details.

For some examples, see the "Generating Deliverables" section of 2022 AAMAS Demo. See *Rendering* for details about rendering capabilities.

Part of default pipeline.

7.1.5 Stage 5: Graph Generation for Controller/Scenario Comparison

SIERRA can perform additional graph generation *AFTER* graph generation for batch experiments has been run. This is extremely useful for generating graphs which can be dropped immediately into academic papers without modification. This can be used to compare:

- Different agent control algorithms which have all been run in the same `--scenario`. See *Intra-Scenario Comparison* for details.
- A single `--controller` across multiple scenarios. See *Inter-Scenario Comparison* for details.

Not part of default pipeline.

7.2 Command Line Interface

Unless an option says otherwise, it is applicable to all batch criteria. That is, option batch criteria applicability is only documented for options which apply to a subset of the available *Batch Criteria*.

If an option is given more than once, the last such occurrence is used.

7.2.1 SIERRA Core

These options are for all *Platforms*.

Bootstrap+Multi-stage Options

```
usage: __main__.py [-h] [--project PROJECT] [--version]
                   [--log-level {ERROR,INFO,WARNING,DEBUG,TRACE}]
                   [--platform PLATFORM] [--skip-pkg-checks]
                   [--exec-env EXEC_ENV] [--template-input-file filepath]
                   [--exp-overwrite] [--sierra-root dirpath]
                   [--batch-criteria [<category>.<definition>,...]
                   [[<category>.<definition>,...] ...]]
                   [--pipeline [PIPELINE ...]] [--exp-range EXP_RANGE]
                   [--platform-vc] [--processing-serial] [--n-runs N_RUNS]
                   [--skip-collate] [--plot-log-xscale]
                   [--plot-enumerated-xscale] [--plot-log-yscale]
                   [--plot-regression-lines PLOT_REGRESSION_LINES]
                   [--plot-primary-axis PLOT_PRIMARY_AXIS] [--plot-large-text]
                   [--plot-transpose-graphs]
```

Bootstrap options

Bare-bones options for bootstrapping SIERRA

--project Specify which *Project* to load.

> **Tip:** Used by stage {1,2,3,4,5}; can be omitted otherwise. If omitted: N/A.

--version, -v Display SIERRA version information on stdout and then exit.

Default: False

--log-level Possible choices: ERROR, INFO, WARNING, DEBUG, TRACE

The level of logging to use when running SIERRA.

> **Tip:** Used by stage {1,2,3,4,5}; can be omitted otherwise. If omitted: N/A.

Default: “INFO”

--platform This argument defines the *Platform* you want to run experiments on.

The value of this argument determines the execution environment for experiments; different platforms (e.g., simulator, real robots) will have different configuration options.

Valid values can be any folder name inside a folder on the *SIERRA_PLUGIN_PATH* (with / replaced with . as you would expect for using path names to address python packages). The platforms which come with SIERRA are:

- `platform.argos` - This directs SIERRA to run experiments using the *ARGoS* simulator. Selecting this platform assumes your code has been developed and configured for ARGoS.
- `platform.ros1gazebo` - This directs SIERRA to run experiments using the *Gazebo* simulator and *ROS1*. Selecting this platform assumes your code has been developed and configured for Gazebo and ROS1.

Default: "platform.argos"

--skip-pkg-checks Skip the usual startup package checks. Only do this if you are SURE you will never use the SIERRA functionality which requires packages you don't have installed/can't install.

Default: False

--exec-env This argument defines *how* experiments are going to be run, using the `--platform` you have selected.

Valid values can be any folder name inside a folder on the *SIERRA_PLUGIN_PATH* (with / replaced with . as you would expect for using path names to address python packages). The execution environments which come with SIERRA are:

- `hpc.local` - This directs SIERRA to run experiments on the local machine. See *Local HPC Plugin* for a detailed description.
- `hpc.pbs` - The directs SIERRA to run experiments spread across multiple allocated nodes in an HPC computing environment managed by TORQUE-PBS. See *PBS HPC Plugin* for a detailed description.
- `hpc.slurm` - The directs SIERRA to run experiments spread across multiple allocated nodes in an HPC computing environment managed by SLURM. See *SLURM HPC Plugin* for a detailed description.
- `hpc.adhoc` - This will direct SIERRA to run experiments on an ad-hoc network of computers. See *Adhoc HPC Plugin* for a detailed description.
- `robot.turtlebot3` - This will direct SIERRA to run experiments on real Turtlebots.

Not all platforms support all execution environments.

Tip: Used by stage {1,2}; can be omitted otherwise. If omitted: N/A.

Default: "hpc.local"

Multi-stage options

Options which are used in multiple pipeline stages

--template-input-file The template `.xml` input file for the batch experiment. Beyond the requirements for the specific `--platform`, the content of the file can be any valid XML, with the exception of the SIERRA requirements detailed in *Template Input Files*.

Tip: Used by stage {1,2,3,4}; can be omitted otherwise. If omitted: N/A.

--exp-overwrite When SIERRA calculates the batch experiment root (or any child path in the batch experiment root) during stage {1, 2}, if the calculated path already exists it is treated as a fatal error and no modifications to the filesystem are performed. This flag overwrides the default behavior. Provided to avoid accidentally overwrite input/output files for an experiment, forcing the user to be explicit with potentially dangerous actions.

Tip: Used by stage {1,2}; can be omitted otherwise. If omitted: N/A.

Default: False

--sierra-root Root directory for all SIERRA generated and created files.

Subdirectories for controllers, scenarios, experiment/experimental run inputs/outputs will be created in this directory as needed. Can persist between invocations of SIERRA.

Tip: Used by stage {1,2,3,4,5}; can be omitted otherwise. If omitted: N/A.

Default: "<home directory>/exp"

--batch-criteria Definition of criteria(s) to use to define the experiment.

Specified as a list of 0 or 1 space separated strings, each with the following general structure:

`<category>.<definition>`

`<category>` must be a filename from the `core/variables/` or from a `--project <project>/variables/` directory, and `<definition>` must be a parsable namne (according to the requirements of the criteria defined by the parser for `<category>`).

Tip: Used by stage {1,2,3,4,5}; can be omitted otherwise. If omitted: N/A.

Default: []

--pipeline Define which stages of the experimental pipeline to run:

- Stage1 - Generate the experiment definition from the template input file, batch criteria, and other command line options. Part of default pipeline.

- Stage2 - Run a previously generated experiment. Part of default pipeline.
- Stage3 - Post-process experimental results after running the batch experiment; some parts of this can be done in parallel. Part of default pipeline.
- Stage4 - Perform deliverable generation after processing results for a batch experiment, which can include shiny graphs and videos. Part of default pipeline.
- Stage5 - Perform graph generation for comparing controllers AFTER graph generation for batch experiments has been run. Not part of default pipeline.

Default: [1, 2, 3, 4]

--exp-range

Set the experiment numbers from the batch to run, average, generate intra-experiment graphs from, or generate inter-experiment graphs from (0 based). Specified in the form `min_exp_num:max_exp_num` (closed interval/inclusive). If omitted, runs, averages, and generates intra-experiment and inter-experiment performance measure graphs for all experiments in the batch (default behavior).

This is useful to re-run part of a batch experiment in HPC environments if SIERRA gets killed before it finishes running all experiments in the batch, or to redo a single experiment with real robots which failed for some reason.

Tip: Used by stage {2,3,4}; can be omitted otherwise. If omitted: N/A.

--platform-vc

For applicable `--platforms`, enable visual capturing of run-time data during stage 2. This data can be frames (i.e., .png files), or rendering videos, depending on the platform. If the captured data was frames, then SIERRA can render the captured frames into videos during stage 4. If the selected `--platform` does not support visual capture, then this option has no effect. See *Platform Visual Capture* for full details.

Tip: Used by stage {1,4}; can be omitted otherwise. If omitted: N/A.

Default: False

--processing-serial

If TRUE, then results processing/graph generation will be performed serially, rather than using parallellism where possible.

Tip: Used by stage {3,4}; can be omitted otherwise. If omitted: N/A.

Default: False

--n-runs

The # of experimental runs that will be executed and their results processed to form the result of a single experiment within a batch.

If `--platform` is a simulator and `--exec-env` is something other than `hpc.local` then this may be be used to determine the concurrency of experimental runs.

Tip: Used by stage {1,2}; can be omitted otherwise. If omitted: N/A.

--skip-collate

Specify that no collation of data across experiments within a batch (stage 4) or across runs within an experiment (stage 3) should be performed. Useful if collation takes a long time and multiple types of stage 4 outputs are desired. Collation is generally idempotent unless you change the stage3 options (YMMV).

Tip: Used by stage {3,4}; can be omitted otherwise. If omitted: N/A.

Default: False

--plot-log-xscale

Place the set of X values used to generate intra- and inter-experiment graphs into the logarithmic space. Mainly useful when the batch criteria involves large system sizes, so that the plots are more readable.

Tip: Applicable graphs:

- *SummaryLineGraph*

Tip: Used by stage {4,5}; can be omitted otherwise. If omitted: N/A.

Default: False

--plot-enumerated-xscale

Instead of using the values generated by a given batch criteria for the X values, use an enumerated list[0, ..., len(X value) - 1]. Mainly useful when the batch criteria involves large system sizes, so that the plots are more readable.

Tip: Applicable graphs:

- *SummaryLineGraph*

Tip: Used by stage {4,5}; can be omitted otherwise. If omitted: N/A.

Default: False

--plot-log-yscale

Place the set of Y values used to generate intra - and inter-experiment graphs into the logarithmic space. Mainly useful when the batch criteria involves large system sizes, so that the plots are more readable.

Tip: Applicable graphs:

- *SummaryLineGraph*
- *StackedLineGraph*

Tip: Used by stage {4,5}; can be omitted otherwise. If omitted: N/A.

Default: False

--plot-regression-lines For all 2D generated scatterplots, plot a linear regression line and the equation of the line to the legend.

Tip: Applicable graphs:

- *SummaryLineGraph*

Tip: Used by stage {4,5}; can be omitted otherwise. If omitted: N/A.

--plot-primary-axis

This option allows you to override the primary axis, which is normally is computed based on the batch criteria.

For example, in a bivariate batch criteria composed of

- *Population Size* on the X axis (rows)
- Another batch criteria which does not affect system size (columns)

Metrics will be calculated by *computing* across .csv rows and *projecting* down the columns by default, since system size will only vary within a row. Passing a value of 1 to this option will override this calculation, which can be useful in bivariate batch criteria in which you are interested in the effect of the OTHER non-size criteria on various performance measures.

0=criteria of interest varies across *rows*.

1=criteria of interest varies across *columns*.

This option only affects generating graphs from bivariate batch criteria.

Tip: Applicable graphs:

- *Heatmap*
- StackedLineGraph

Tip: Used by stage {4,5}; can be omitted otherwise. If omitted: N/A.

--plot-large-text

This option specifies that the title, X/Y axis labels/tick labels should be larger than the SIERRA default. This is useful when generating graphs suitable for two column paper format where the default text size for rendered graphs will be too small to see easily. The SIERRA defaults are generally fine for the one column/journal paper format.

Tip: Used by stage {4,5}; can be omitted otherwise. If omitted: N/A.

Default: False

--plot-transpose-graphs Transpose the X, Y axes in generated graphs. Useful as a general way to tweak graphs for best use of space within a paper.

Changed in version 1.2.20: Renamed from `--transpose-graphs` to make its relation to other plotting options clearer.

Tip: Applicable graphs:

- *Heatmap*

Tip: Used by stage {4,5}; can be omitted otherwise. If omitted: N/A.

Default: False

Stage1: Generating Experiments

None for the moment.

Stage2: Running Experiments

None for the moment.

Stage3: Processing Experiment Results

```
usage: SIERRA [--df-skip-verify] [--storage-medium {storage.csv}]
              [--dist-stats {none,all,conf95,bw}]
              [--processing-mem-limit PROCESSING_MEM_LIMIT]
              [--df-homogenize DF_HOMOGENIZE]
```

Stage3: General options for eprocessing experiment results

--df-skip-verify SIERRA generally assumes/relies on all dataframes with the same name having the same # of columns which are of equivalent length across *Experimental Runs* (different columns within a dataframe can of course have different lengths). This is strictly verified during stage 3 by default.

If passed, then the verification step will be skipped during experimental results processing, and outputs will be averaged directly.

If not all the corresponding CSV files in all experiments generated the same # rows, then SIERRA will (probably) crash during experiments exist and/or have the stage4. Verification can take a long time with large # of runs and/or dataframes per experiment.

Tip: Used by stage {3}; can be omitted otherwise. If omitted: N/A.

Default: False

--storage-medium Possible choices: storage.csv

Specify the storage medium for *Experimental Run* outputs, so that SIERRA can select an appropriate plugin to read them. Any plugin under `plugins/storage` can be used, but the ones that come with SIERRA are:

- `storage.csv` - Experimental run outputs are stored in a per-run directory as one or more CSV files.

Regardless of the value of this option, SIERRA always generates CSV files as it runs and averages outputs, generates graphs, etc.

Tip: Used by stage {3}; can be omitted otherwise. If omitted: N/A.

Default: "storage.csv"

--dist-stats Possible choices: none, all, conf95, bw

Specify what kinds of statistics, if any, should be calculated on the distribution of experimental data during stage 3 for inclusion on graphs during stage 4:

- `none` - Only calculate and show raw mean on graphs.
- `conf95` - Calculate standard deviation of experimental distribution and show 95% confidence interval on relevant graphs.
- `bw` - Calculate statistics necessary to show box and whisker plots around each point in the graph. :class: *~sierra.core.graphs.summary_line_graph.SummaryLineGraph* only).
- `all` - Generate all possible statistics, and plot all possible statistics on graphs.

Tip: Applicable graphs:

- *SummaryLineGraph*
- *StackedLineGraph*

Tip: Used by stage {3,4}; can be omitted otherwise. If omitted: N/A.

Default: "none"

--processing-mem-limit Specify, as a percent in [0, 100], how much memory SIERRA should try to limit itself to using. This is useful on systems with limited memory, or on systems which are shared with other users without per-user memory restrictions.

Tip: Used by stage {3,4}; can be omitted otherwise. If omitted: N/A.

Default: 90

--df-homogenize

SIERRA generally assumes/relies on all dataframes with the same name having the same # of columns which are of equivalent length across: term: *Experimental Runs* < *Experimental Run* > (different columns within a dataframe can of course have different lengths). This is checked during stage 3 unless `--df-skip-verify` is passed. If strict verification is skipped, then SIERRA provides the following options when processing dataframes during stage $\{3, 4\}$ to to homogenize them:

- `none` - Don't do anything. This may or may not produce crashes during stage 4, depending on what you are doing.
- `pad` - Project last valid value in columns which are too short down the column to make it match those which are longer.

  Note that this may result in invalid data/graphs if the filled columns are intervallic, interval average, or cumulative average data. If the data is a cumulative count of something, then this policy will have no ill effects.
- `zero` - Same as `pad`, but always fill with zeroes.

Homogenization is performed just before writing dataframes to the specified storage medium. Useful with real robot experiments if the number of datapoints captured per-robot is slightly different, depending on when they started executing relative to when the experiment started.

Default: "none"

Stage4: Deliverable Generation

```
usage: SIERRA [--exp-graphs {intra,inter,all,none}] [--project-no-LN]
              [--project-no-HM] [--models-enable]
              [--render-cmd-opts RENDER_CMD_OPTS] [--project-imagizing]
              [--project-rendering] [--bc-rendering]
```

Stage4: General options for generating graphs

--exp-graphs

Possible choices: intra, inter, all, none

Specify which types of graphs should be generated from experimental results:

- `intra` - Generate intra-experiment graphs from the results of a single experiment within a batch, for each experiment in the batch(this can take a long time with large batch experiments). If any intra-experiment models are defined and enabled, those are run and the results placed on appropriate graphs.
- `inter` - Generate inter-experiment graphs _across_ the results of all experiments in a batch. These are very fast to generate, regardless of batch experiment size. If any inter-experiment models are defined and enabled, those are run and the results placed on appropriate graphs.

- `all` - Generate all types of graphs.
- `none` - Skip graph generation; provided to skip graph generation if only video outputs are desired.

Tip: Used by stage {4}; can be omitted otherwise. If omitted: N/A.

Default: "all"

--project-no-LN

Specify that the intra-experiment and inter-experiment linegraphs defined in project YAML configuration should not be generated. Useful if you are working on something which results in the generation of other types of graphs, and the generation of those linegraphs is not currently needed only slows down your development cycle.

Model linegraphs are still generated, if applicable.

Default: False

--project-no-HM

Specify that the intra-experiment heatmaps defined in project YAML configuration should not be generated. Useful if:

- You are working on something which results in the generation of other types of graphs, and the generation of heatmaps only slows down your development cycle.
- You are working on stage5 comparison graphs for bivariate batch criteria, and re-generating many heatmaps during stage4 is taking too long.

Model heatmaps are still generated, if applicable.

New in version 1.2.20.

Default: False

Models

--models-enable

Enable running of all models; otherwise, no models are run, even if they appear in the project config file. The logic behind having models disabled by default is that most users won't have them.

Default: False

Stage4: Rendering (see also stage1 rendering options)

--render-cmd-opts

Specify the: program: *ffmpeg* options to appear between the specification of the input image files and the specification of the output file. The default is suitable for use with ARGoS frame grabbing set to a frames size of 1600x1200 to output a reasonable quality video.

Tip: Used by stage {4}; can be omitted otherwise. If omitted: N/A.

Default: "-r 10 -s:v 800x600 -c:v libx264 -crf 25 -filter:v scale=-2:956 -pix_fmt yuv420p"

--project-imagizing Enable generation of image files from CSV files captured during stage 2 and averaged during stage 3 for each experiment. See *Project Rendering* for details and restrictions.

Tip: Used by stage {3,4}; can be omitted otherwise. If omitted: N/A.

Default: False

--project-rendering Enable generation of videos from imagized CSV files created as a result of `--project-imagizing`. See *Project Rendering* for details.

Tip: Used by stage {4}; can be omitted otherwise. If omitted: N/A.

Default: False

--bc-rendering Enable generation of videos from generated graphs, such as heatmaps. Bivariate batch criteria only.

Tip: Used by stage {4}; can be omitted otherwise. If omitted: N/A.

Default: False

Stage5: Comparing Controllers

```
usage: SIERRA [--controllers-list CONTROLLERS_LIST]
              [--controllers-legend CONTROLLERS_LEGEND]
              [--scenarios-list SCENARIOS_LIST]
              [--scenarios-legend SCENARIOS_LEGEND] [--scenario-comparison]
              [--controller-comparison]
              [--comparison-type {LNraw,HMraw,HMdiff,HMscale,SUraw,SUscale,SUdiff}]
              [--bc-univar] [--bc-bivar]
```

Stage5: General options for controller comparison

--controllers-list Comma separated list of controllers to compare within `--sierra-root`.

The first controller in this list will be used for as the controller of primary interest if `--comparison-type` is passed.

Tip: Used by stage {5}; can be omitted otherwise. If omitted: N/A.

--controllers-legend Comma separated list of names to use on the legend for the generated comparison graphs, specified in the same order as the `--controllers-list`.

Tip: Used by stage {5}; can be omitted otherwise. If omitted: the raw controller names will be used.

--scenarios-list Comma separated list of scenarios to compare `--controller` across within `--sierra-root`.

Tip: Used by stage {5}; can be omitted otherwise. If omitted: N/A.

--scenarios-legend Comma separated list of names to use on the legend for the generated inter-scenario controller comparison graphs(if applicable), specified in the same order as the `--scenarios-list`.

Tip: Used by stage {5}; can be omitted otherwise. If omitted: the raw scenario names will be used.

--scenario-comparison Perform a comparison of `--controller` across `--scenarios-list` (univariate batch criteria only).

Tip: Used by stage {5}; can be omitted otherwise. Either `--scenario-comparison` or `--controller-comparison` must be passed.

Default: False

--controller-comparison Perform a comparison of `--controllers-list` across all scenarios at least one controller has been run on.

Tip: Used by stage {5}; can be omitted otherwise. Either `--scenario-comparison` or `--controller-comparison` must be passed.

Default: False

--comparison-type Possible choices: LNraw, HMraw, HMdiff, HMscale, SUraw, SUscale, SUdiff

Specify how controller comparisons should be performed.

If the batch criteria is univariate, the options are:

- `LNraw` - Output raw 1D performance measures using a single *SummaryLineGraph* for each measure, with all `--controllers-list` controllers shown on the same graph.

If the batch criteria is bivariate, the options are:

- `LNraw` - Output raw performance measures as a set of :class:`~sierra.core.graphs.summary_line_graph.SummaryLineGraph`s, where each line graph is constructed from the i-th row/column for the 2D dataframe for the performance results for all controllers.

  Note: SIERRA cannot currently plot statistics on the line-graphs built from slices of the 2D CSVs/heatmaps generated during stage4, because statistics generation is limited to stage3. This limitation may be removed in a future release.

- `HMraw` - Output raw 2D performance measures as a set of dual heatmaps comparing all controllers against the controller of primary interest(one per pair).
- `HMdiff` - Subtract the performance measure of the controller of primary interest against all other controllers, pairwise, outputting one 2D heatmap per comparison.
- `HMscale` - Scale controller performance measures against those of the controller of primary interest by dividing, outputing one 2D heatmap per comparison.
- `SUraw` - Output raw 3D performance measures as a single, stacked 3D surface plots comparing all controllers(identical plots, but viewed from different angles).
- `SUscale` - Scale controller performance measures against those of the controller of primary interest by dividing. This results in a single stacked 3D surface plots comparing all controllers(identical plots, but viewed from different angles).
- `SUdiff` - Subtract the performance measure of the controller of primary interest from each controller(including the primary). This results in a set single stacked 3D surface plots comparing all controllers(identical plots, but viewed from different angles), in which the controller of primary interest forms an(X, Y) plane at Z=0.

For all comparison types, `--controllers-legend` is used if passed for legend.

Tip: Used by stage {5}; can be omitted otherwise. If omitted: N/A.

--bc-univar

Specify that the batch criteria is univariate. This cannot be deduced from the command line `--batch-criteria` argument in all cases because we are comparing controllers *across* scenarios, and each scenario (potentially) has a different batch criteria definition, which will result in (potentially) erroneous comparisons if we don't re-generate the batch criteria for each scenaro we compare controllers within.

Tip: Used by stage {5}; can be omitted otherwise. If omitted: N/A.

Default: False

--bc-bivar

Specify that the batch criteria is bivariate. This cannot be deduced from the command line `--batch-criteria` argument in all cases because we are comparing controllers *across* scenarios, and each scenario(potentially) has a different batch criteria definition, which will result in (potentially) erroneous comparisons if we don't re-generate the batch criteria for each scenaro we compare controllers in .

Tip: Used by stage {5}; can be omitted otherwise. If omitted: N/A.

Default: False

7.2.2 ARGoS Platform

These options are enabled if `--platform=platform.argos` is passed.

Stage1: Generating Experiments

```
usage: SIERRA [--exp-setup EXP_SETUP] [--physics-engine-type2D {dynamics2d}]
              [--physics-engine-type3D {dynamics3d}]
              [--physics-n-engines {1,2,4,6,8,12,16,24}]
              [--physics-iter-per-tick PHYSICS_ITER_PER_TICK]
              [--physics-spatial-hash2D]
              [--camera-config {overhead,sw,sw+interp}] [--with-robot-rab]
              [--with-robot-leds] [--with-robot-battery] [--n-robots N_ROBOTS]
```

Stage1: Experiment generation

--exp-setup Defines experiment run length, *Ticks* per second for the experiment (<experiment> tag), # of datapoints to capture/capture interval for each simulation. See *Experiment Setup* for a full description.

Tip: Used by stage {1}; can be omitted otherwise. If omitted: N/A.

Default: "exp_setup.T5000.K5.N50"

Stage1: Configuring ARGoS physics engines

--physics-engine-type2D Possible choices: dynamics2d

The type of 2D physics engine to use for managing spatial extents within the arena, choosing one of the types that ARGoS supports. The precise 2D areas (if any) within the arena which will be controlled by 2D physics engines is defined on a per `--project` basis.

Tip: Used by stage {1}; can be omitted otherwise. If omitted: N/A.

Default: "dynamics2d"

--physics-engine-type3D Possible choices: dynamics3d

The type of 3D physics engine to use for managing 3D volumetric extents within the arena, choosing one of the types that ARGoS supports. The precise 3D volumes (if any) within the arena which will be controlled by 3D physics engines is defined on a per `--project` basis.

Tip: Used by stage {1}; can be omitted otherwise. If omitted: N/A.

Default: "dynamics3d"

--physics-n-engines Possible choices: 1, 2, 4, 6, 8, 12, 16, 24

# of physics engines to use during simulation (yay ARGoS!). If N > 1, the engines will be tiled in a uniform grid within the arena (X and Y spacing may not be the same depending on dimensions and how many engines are chosen, however), extending upward in Z to the height specified by `--scenario` (i.e., forming a set of "silos" rather that equal volumetric extents).

If 2D and 3D physics engines are mixed, then half of the specified # of engines will be allocated among all arena extents cumulatively managed by each type of engine. For example, if 4 engines are used, with 1/3 of the arena managed by 2D engines and 2/3 by 3D, then 2 2D engines will manage 1/3 of the arena, and 2 3D engines will manage the other 2/3 of the arena.

If `--exec-env` is something other than `hpc.local` then the # physics engines will be computed from the HPC environment, and the cmdline value (if any) will be ignored.

Important: When using multiple physics engines, always make sure that `# engines / arena dimension` (X **AND** Y dimensions) is always a rational number. That is,

- 24 engines in a `12x12` arena will be fine, because `12/24=0.5`, which can be represented reasonably well in floating point.
- 24 engines in a `16x16` arena will not be fine, because `16/24=0.666667`, which will very likely result in rounding errors and ARGoS being unable to initialize the space because it can't place arena walls.

This is enforced by SIERRA.

Tip: Used by stage {1}; can be omitted otherwise. If omitted: N/A.

--physics-iter-per-tick The # of iterations all physics engines should perform per *Tick* each time the controller loops are run (the # of ticks per second for controller control loops is set via `--exp-setup`).

Tip: Used by stage {1}; can be omitted otherwise. If omitted: N/A.

Default: 10

--physics-spatial-hash2D Specify that each 2D physics engine should use a spatial hash (only applies to `dynamics2d` engine type).

Default: False

Stage1: Rendering (see also stage4 rendering options)

--camera-config Possible choices: overhead, sw, sw+interp

Select the camera configuration for simulation. Ignored unless `--platform-vc` is passed. Valid values are:

- `overhead` - Use a single overhead camera at the center of the aren looking straight down at an appropriate height to see the whole arena.
- `sw` - Use the ARGoS camera configuration (12 cameras), cycling through them periodically throughout simulation without interpolation.
- `sw+interp` - Same as `sw`, but with interpolation between cameras.

Tip: Used by stage {1}; can be omitted otherwise. If omitted: N/A.

Default: "overhead"

Stage1: Configuring robots

--with-robot-rab If passed, do not remove the Range and Bearing (RAB) sensor, actuator, and medium XML definitions from `--template-input-file` before generating experimental inputs. Otherwise, the following XML tags are removed if they exist:

- `.//media/range_and_bearing`
- `.//actuators/range_and_bearing`
- `.//sensors/range_and_bearing`

Tip: Used by stage {1}; can be omitted otherwise. If omitted: N/A.

Default: False

--with-robot-leds If passed, do not remove the robot LED actuator XML definitions from the `--template-input-file` before generating experimental inputs. Otherwise, the following XML tags are removed if they exist:

- `.//actuators/leds`
- `.//medium/leds`
- `.//sensors/colored_blob_omnidirectional_camera`

Tip: Used by stage {1}; can be omitted otherwise. If omitted: N/A.

Default: False

--with-robot-battery If passed, do not remove the robot battery sensor XML definitions from `--template-input-file` before generating experimental inputs. Otherwise, the following XML tags are removed if they exist:

- `.//entity/*/battery`
- `.//sensors/battery`

Tip: Used by stage {1}; can be omitted otherwise. If omitted: N/A.

Default: False

--n-robots The # robots that should be used in the simulation. Can be used to override batch criteria, or to supplement experiments that do not set it so that manual modification of input file is unneccesary.

Tip: Used by stage {1}; can be omitted otherwise. If omitted: N/A.

Stage2: Running Experiments

```
usage: SIERRA [--exec-jobs-per-node EXEC_JOBS_PER_NODE] [--exec-devnull]
              [--exec-no-devnull] [--exec-resume] [--exec-strict]
```

HPC options

For platforms which are simulators (and can therefore be run in HPC environments).

--exec-jobs-per-node Specify the maximum number of parallel jobs to run per allocated node. By default this is computed from the selected HPC environment for maximum throughput given the desired `--n-runs` and CPUs per allocated node. However, for some environments being able to override the computed default can be useful.

Tip: Used by stage {2}; can be omitted otherwise. If omitted: N/A.

--exec-devnull Redirect ALL output from simulations to /dev/null. Useful for platform where you can't disable all INFO messages at compile time, and don't want to have to grep through lots of redundant stdout files to see if there were any errors.

Tip: Used by stage {1,2}; can be omitted otherwise. If omitted: N/A.

Default: True

--exec-no-devnull Don't redirect ALL output from simulations to /dev/null. Useful for platform where you can't disable all INFO messages at compile time, and don't want to have to grep through lots of redundant stdout files to see if there were any errors.

Tip: Used by stage {1,2}; can be omitted otherwise. If omitted: N/A.

Default: True

--exec-resume

Resume a batch experiment that was killed/stopped/etc last time SIERRA was run.

Tip: Used by stage {2}; can be omitted otherwise. If omitted: N/A.

Default: False

--exec-strict

If passed, then if any experimental commands fail during stage 2 SIERRA will exit, rather than try to keep going and execute the rest of the experiments.

Useful for:

- "Correctness by construction" experiments, where you know if SIERRA doesn't crash and it makes it to the end of your batch experiment then none of the individual experiments crashed.
- CI pipelines

Default: False

7.2.3 ROS1+Gazebo Platform

These options are enabled if `--platform=platform.ros1gazebo` is passed.

Stage1: Generating Experiments

```
usage: SIERRA [--exp-setup EXP_SETUP] [--robot ROBOT]
              [--robot-positions ROBOT_POSITIONS [ROBOT_POSITIONS ...]]
              [--physics-engine-type {ode,bullet,dart,simbody}]
              [--physics-iter-per-tick PHYSICS_ITER_PER_TICK]
              [--physics-n-threads PHYSICS_N_THREADS]
              [--physics-ec-threadpool PHYSICS_EC_THREADPOOL]
```

Stage1: Experiment generation

--exp-setup

Defines experiment run length, ticks per second for the experiment, # of datapoints to capture/capture interval for each simulation. See *Experiment Setup* for a full description.

Tip: Used by stage {1}; can be omitted otherwise. If omitted: N/A.

Default: "exp_setup.T1000.K5.N50"

--robot

The key name of the robot model, which must be present in the appropriate section of `main.yaml` for the *Project*. See *Main Configuration* for details.

Tip: Used by stage {1}; can be omitted otherwise. If omitted: N/A.

Stage1: Experiment setup

--robot-positions A list of space-separated "X,Y,Z" tuples (no quotes) passed on the command line as valid starting positions for the robots within the world.

Tip: Used by stage {1}; can be omitted otherwise. If omitted: effective arena dimensions must be given as part of `--scenario`.

Default: []

Stage1: Configuring Gazebo physics engines

--physics-engine-type Possible choices: ode, bullet, dart, simbody

The type of 3D physics engine to use for managing spatial extents within the arena, choosing one of the types that *Gazebo* supports. A single engine instance is used to manage all physics in the arena.

Tip: Used by stage {1}; can be omitted otherwise. If omitted: N/A.

Default: "ode"

--physics-iter-per-tick The # of iterations all physics engines should perform per tick each time the controller loops are run (the # of ticks per second for controller control loops is set via `--exp-setup`).

Tip: Used by stage {1}; can be omitted otherwise. If omitted: N/A.

Default: 1000

--physics-n-threads Gazebo can group non-interacting entities into computational "islands" and do the physics updates for those islands in parallel each timestep (islands) are recomputed after each timestep). Gazebo can also parallelize the computation of velocity/position updates with the computation of resolving collisions (i.e., the timestep impulse results in one entity "inside" another). You can assign multiple threads to a pool for cumulative use for these two parallelization methods (threads will be allocated evenly between them). The point at which adding more threads will start to DECREASE performance depends on the complexity of your world, the number and type of robots in it, etc., so don't just set this parameter to the # of cores for your machine as a default.

From the Gazebo Parallel Physics Report, setting the pool size to the # robots/# joint trees in your simulation usually gives good results, as long as you have more cores available than you allocate to this pool (Gazebo has other threads too).

This only applies if `--physics-engine-type=ode`.

A value of 0=no threads.

Tip: Used by stage {1}; can be omitted otherwise. If omitted: N/A.

Default: 0

--physics-ec-threadpool Gazebo can parallelize the computation of velocity/position updates with the computation of resolving collisions (i.e., the timestep impulse results in one entity "inside" another). You can assign multiple threads to a pool for cumulative use for this purpose. The point at which adding more threads will start to DECREASE performance depends on the complexity of your world, the number and type of robots in it, etc., so don't just set this parameter to the # of cores for your machine as a default.

From the Gazebo Parallel Physics Report, setting the pool size to the # robots/#joint trees in your simulation usually gives good results, as long as you have more cores than than you allocate to physics (Gazebo has other threads too).

This only applies if `--physics-engine-type=`ode.

A value of 0=no threads.

Tip: Used by stage {1}; can be omitted otherwise. If omitted: N/A.

Default: 0

Stage2: Running Experiments

```
usage: SIERRA [--exec-jobs-per-node EXEC_JOBS_PER_NODE] [--exec-devnull]
              [--exec-no-devnull] [--exec-resume] [--exec-strict]
```

HPC options

For platforms which are simulators (and can therefore be run in HPC environments).

--exec-jobs-per-node Specify the maximum number of parallel jobs to run per allocated node. By default this is computed from the selected HPC environment for maximum throughput given the desired `--n-runs` and CPUs per allocated node. However, for some environments being able to override the computed default can be useful.

Tip: Used by stage {2}; can be omitted otherwise. If omitted: N/A.

--exec-devnull Redirect ALL output from simulations to /dev/null. Useful for platform where you can't disable all INFO messages at compile time, and don't want to have to grep through lots of redundant stdout files to see if there were any errors.

Tip: Used by stage {1,2}; can be omitted otherwise. If omitted: N/A.

Default: True

--exec-no-devnull

Don't redirect ALL output from simulations to /dev/null. Useful for platform where you can't disable all INFO messages at compile time, and don't want to have to grep through lots of redundant stdout files to see if there were any errors.

Tip: Used by stage {1,2}; can be omitted otherwise. If omitted: N/A.

Default: True

--exec-resume

Resume a batch experiment that was killed/stopped/etc last time SIERRA was run.

Tip: Used by stage {2}; can be omitted otherwise. If omitted: N/A.

Default: False

--exec-strict

If passed, then if any experimental commands fail during stage 2 SIERRA will exit, rather than try to keep going and execute the rest of the experiments.

Useful for:

- "Correctness by construction" experiments, where you know if SIERRA doesn't crash and it makes it to the end of your batch experiment then none of the individual experiments crashed.
- CI pipelines

Default: False

7.2.4 ROS1+Robot Platform

These options are enabled if `--platform=platform.ros1robot` is passed.

Stage1: Generating Experiments

```
usage: SIERRA [--exp-setup EXP_SETUP] [--robot ROBOT] [--skip-sync]
```

Stage1: Experiment generation

--exp-setup

Defines experiment run length, ticks per second for the experiment, # of datapoints to capture/capture interval for each simulation. See *Experiment Setup* for a full description.

Tip: Used by stage {1}; can be omitted otherwise. If omitted: N/A.

Default: "exp_setup.T1000.K5.N50"

--robot The key name of the robot model, which must be present in the appropriate section of `main.yaml` for the *Project*. See *Main Configuration* for details.

Tip: Used by stage {1}; can be omitted otherwise. If omitted: N/A.

Stage1: Experiment generation

--skip-sync If passed, then the generated experiment will not be synced to robots. This is useful when:

- You are developing your *Project* and just want to check locally if the experiment is being generated properly.
- You have a lot of robots and/or the network connection from the SIERRA host machine to the robots is slow, and copying the experiment multiple times as you tweak parameters takes a long time.

Tip: Used by stage {1}; can be omitted otherwise. If omitted: N/A.

Default: False

Stage2: Running Experiments

```
usage: SIERRA [--exec-inter-run-pause SECONDS] [--exec-resume]
```

Stage2: Experiment executionFor running real robot experiments

--exec-inter-run-pause How long to pause between *Experimental Runs*, giving you time to reset the environment, move robots, etc.

Tip: Used by stage {2}; can be omitted otherwise. If omitted: N/A.

Default: 60

--exec-resume Resume a batch experiment that was killed/stopped/etc last time SIERRA was run.

Tip: Used by stage {2}; can be omitted otherwise. If omitted: N/A.

Default: False

7.3 SIERRA Runtime Directory Tree

Important: SIERRA **NEVER** deletes directories for you.

Subsequent experiments using the same values for the following cmdline arguments **WILL** result in the same calculated root directory for experimental inputs and outputs, even if other parameters change (or if you change the contents of the template input file):

- `--controller`
- `--scenario`
- `--sierra-root`
- `--template-input-file`
- `--batch-criteria`

SIERRA will abort stage {1,2} processing when this occurs in order to preserve data integrity; this behavior can be overwridden with `--exp-overwrite`, in which case the use assumes full responsibility for ensuring the integrity of the experiment.

Always better to check the arguments before hitting ENTER. Measure twice, cut once, as the saying goes.

7.3.1 Default Pipeline Directory Tree (Stages 1-4)

When SIERRA runs stages 1-4, it creates a directory structure under whatever was passed as `--sierra-root`. For the purposes of explanation, I will use the following partial SIERRA option set to explain the experiment tree:

```
--sierra-root=$HOME/exp\
--controller=CATEGORY.my_controller\
--scenario=SS.12x6\
--platform=platform.argos\
--batch-criteria=population_size.Log8\
--n-runs=4\
--template-input-file=~/my-template.argos\
--project=fordyca
```

This invocation will cause SIERRA to create the following directory structure as it runs:

- `$HOME/exp` - This is the root of the directory structure (`--sierra-root`), and is **NOT** deleted on subsequent runs.
  - `fordyca` - Each project gets their own directory, so you can disambiguate otherwise identical SIERRA invocations and don't overwrite the directories for a previously used project on subsequent runs.
    - `CATEGORY.my_controller` - Each controller gets their own directory in the project root, which is **NOT** deleted on subsequent runs.
      - `mytemplate-SS.12x6` - The directory for the *Batch Experiment* is named from a combination of the template input file used (`--template-input-file`) and the scenario (`--scenario`).
        - `exp-inputs` - Root directory for *Experimental* inputs; each experiment in the batch gets their own directory in here.
          - `exp0` - Within the input directory for each experiment in the batch (there are 4 such directories in this example), there will be an input file for each *Experimental Run* in the experiment, as well as a `commands.txt` used by GNU parallel to run them all in parallel. The leaf of the

`--template-input-file`, sans extension, has the experimental run # appended to it (e.g. `my-template_0.argos` is the input file for simulation 0).

- `commands.txt`
- `my-template_0.argos`
- `my-template_1.argos`
- `my-template_2.argos`
- `my-template_3.argos`

- `exp1`
  - `my-template_0.argos`
  - `my-template_1.argos`
  - `my-template_2.argos`
  - `my-template_3.argos`
- `exp2`
  - `...`

- `exp-outputs` - Root directory for experimental outputs; each experiment gets their own directory in here (just like for experiment inputs). Directory name is controlled by the main YAML configuration.
  - `exp0` - Within the output directory for each experiment in the batch (there are 3 such directories in this example), there will be a *directory* (rather than a file, as was the case for inputs) for each experimental run's output, including metrics, grabbed frames, etc., as configured in the XML input file.
    - `my-template_0_output`
    - `my-template_1_output`
    - `my-template_2_output`
    - `my-template_3_output`
  - `exp1`
    - `my-template_0_output`
    - `my-template_1_output`
    - `my-template_2_output`
    - `my-template_3_output`
  - `exp2`
    - `...`
  - `statistics` - Root directory for holding statistics calculated during stage3 for use during stage4.
    - `exp0` - Contains the results of statistics generation for exp0 (mean, stddev, etc., as configured).
    - `exp1`
    - `exp2`
    - `...`

        - `collated` - Contains *Collated .csv* files. During stage4, SIERRA will draw specific columns from .csv files under `statistics` according to configuration, and collate them under here for graph generation of *inter*-experiment graphs.
        - `exec` - Statistics about SIERRA runtime. Useful for capturing runtime of specific experiments to better plan/schedule time on HPC clusters.
    - `imagize` - Root directory for holding imagized files (averaged run outputs which have been converted to graphs) which can be patched together in stage 4 to generated videos. Each experiment will get its own directory under here, with unique sub-directories for each different type of *Experimental Run* data captured for imagizing. See *Project Rendering* for more details.
    - `videos` - Root directory for holding rendered videos generated during stage 4 from either captured simulator frames for imagized project files. Each experiment will get its own directory under here, with See *Rendering* for more details.
    - `models` - During stage4, the dataframes generated by all executed models are stored under this directory. Each experiment in the batch gets their own directory for *intra*-experiment models.
    - `graphs` - During stage4, all generated graphs are output under this directory. Each experiment in the batch gets their own directory for *intra*-experiment graphs.
        - `exp0`
        - `exp1`
        - `exp2`
        - `exp3`
        - `collated` - Graphs which are generated across experiments in the batch from collated .csv data, rather than from the averaged results within each experiment, are output here.

7.3.2 Stage 5 Directory Tree

When SIERRA runs stage 5, stages 1-4 must have already been successfully run, and therefore the directory tree shown above will exist. For the purposes of explanation, I will use the following partial SIERRA option sets to explain the additions to the experiment tree for stage 5.

First, the experiment tree for *scenario comparison*:

```
--pipeline 5\
--scenario-comparison\
--batch-criteria population_size.Log8\
--scenarios-list=RN.16x16x2,PL.16x16x2\
--sierra-root=$HOME/exp"
```

This invocation will cause SIERRA to create the following directory structure as it runs:

- `$HOME/exp`
    - `RN.16x16x2+PL.16x16x2-sc-graphs`

      This is the directory holding the comparison graphs for all controllers which were previously run on the scenarios `RN.16x16x2` and `PL.16x16x2` (scenario names are arbitrary for the purposes of stage 5 and entirely depend on the project). Inside this directory will be all graphs generated according to the configuration specified in *Stage 5 Configuration*.

Second, the experiment tree for *controller comparison*

```
--pipeline 5\
--controller-comparison\
--batch-criteria population_size.Log8\
--controllers-list d0.CRW,d0.DPO\
--sierra-root=$HOME/exp"
```

This invocation will cause SIERRA to create the following directory structure as it runs:

- `$HOME/exp`
  - `d0.CRW+d0.DPO-cc-graphs`

    This is the directory holding the comparison graphs for each scenario for which `d0.CRW` and `d0.DPO` were run (scenarios are computed by examining the directory tree for stages 1-4). Controller names are arbitrary for the purposes of stage 5 and entirely depend on the project). Inside this directory will be all graphs generated according to the configuration specified in *Stage 5 Configuration*.

7.4 SIERRA Subprograms

These are the shell programs which SIERRA *may* use internally when running, depending on what you are doing.

- **`parallel`** - GNU parallel. Used during stage 2 when running experiments (*ARGoS*, *ROS1+Gazebo*, *ROS1+Robot* platforms).
- **`ffmpeg`** - Used during stage 3 if imagizing is run. See *Platform Visual Capture*.
- **`Xvfb`** - Used during stage 1 when generating simulation inputs, and during stage 2 when running experiments for the *ARGoS Platform*. See also *Platform Visual Capture*.
- **`parallel-ssh`** - Used during stage 1 when generating experiments experiments (*ROS1+Robot* platform).
- **`parallel-rsync`** - Used during stage 1 when generating experiments experiments (*ROS1+Robot* platform).

7.5 Environment Variables

`SIERRA_PLUGIN_PATH`
: Used for locating *plugins*. The directory *containing* a plugin directory outside the SIERRA source tree must be on `SIERRA_PLUGIN_PATH`. Paths are added to `PYTHONPATH` as needed to load plugins correctly. For example, if you have a different version of the `--storage-medium` plugin you'd like to use, and you have but the directory containing the plugin in `$HOME/plugins`, then you need to add `$HOME/plugins` to your `SIERRA_PLUGIN_PATH` to so that SIERRA will find it. This variable is used in stages 1-5.

  Used for locating *projects*; all projects specifiable with `--project` are directories found within the directories on this path. For example, if you have a project `$HOME/git/projects/myproject`, then `$HOME/git` must be on `SIERRA_PLUGIN_PATH` in order for you to be able to specify `--project=myproject`. This variable is used in stages 1-5.

  You *cannot* just put the parent directory of your project on *`PYTHONPATH`* because SIERRA uses this path for other things internally (e.g., computing the paths to YAML config files).

`PYTHONPATH`
: Used for locating projects per the usual python mechanisms.

`ARGOS_PLUGIN_PATH`
: Must be set to contain the library directory where you installed/built ARGoS, as well as the library directory for your project `.so`. Checked to be non-empty before running stage 2 for all `--exec-env` plugins. SIERRA does

not modify this variable, so it needs to be setup properly prior to invoking SIERRA (i.e., the directory containing the *Project* `.so` file needs to be on it). SIERRA can't know, in general, where the location of the C++ code corresponding to the loaded *Project* is.

SIERRA_ARCH
Can be used to determine the names of executables launch in HPC environment, so that in environments with multiple queues/sub-clusters with different architectures simulators can be compiled natively for each for maximum performance. Can be any string. If defined, then instead of searching for the `foobar` executable for some platform on `PATH`, SIERRA will look for `foobar-$SIERRA_ARCH`.

Important: Not all platforms use this variable–see the docs for your platform of interest.

SIERRA_NODEFILE
Points to a file suitable for passing to **`parallel`** via `--sshloginfile`. See **`parallel`** docs for general content/formatting requirements.

Used by SIERRA to configure experiments during stage 1,2; if it is not defined and `--nodefile` is not passed SIERRA will throw an error.

PARALLEL
When running on some execution environments, such as `hpc.slurm,hpc.pbs`, any and all environment variables needed by your *Project* should be exported via the `PARALLEL` environment variable before invoking SIERRA, because GNU parallel does not export the environment of the node it is launched from to slave nodes (or even on the local machine). Something like:

```
export PARALLEL="--workdir . \
--env PATH \
--env LD_LIBRARY_PATH \
--env LOADEDMODULES \
--env _LMFILES_ \
--env MODULE_VERSION \
--env MODULEPATH \
--env MODULEVERSION_STACK
--env MODULESHOME \
--env OMP_DYNAMICS \
--env OMP_MAX_ACTIVE_LEVELS \
--env OMP_NESTED \
--env OMP_NUM_THREADS \
--env OMP_SCHEDULE \
--env OMP_STACKSIZE \
--env OMP_THREAD_LIMIT \
--env OMP_WAIT_POLICY \
--env SIERRA_ARCH \
--env SIERRA_PLUGIN_PATH"
```

Don't forget to include *ARGOS_PLUGIN_PATH*, *ROS_PACKAGE_PATH*, etc., depending on your chosen *Platform*.

PARALLEL_SHELL
SIERRA sets up the *Experiment* execution environments by running one or more shell commands in a subprocess (treated as a `shell`, which means that **`parallel`** can't determine `SHELL`, and therefore defaults to `/bin/sh`, which is not what users expect. SIERRA explicitly sets `PARALLEL_SHELL` to the result of `shutil.which('bash')` in keeping with the Principle Of Least Surprise.

ROS_PACKAGE_PATH
The list of directories which defines where ROS will search for packages. SIERRA does *not* modify this variable, so it needs to be setup properly prior to invoking SIERRA (i.e., sourcing the proper `setup.bash` script).

7.6 Configurable SIERRA Variables

Non-*Batch Criteria* variables which you can use to configure simulations. All batch criteria are variables, but not all variables are batch criteria.

- *Experiment Setup*

7.6.1 Experiment Setup

Configure *Experiment* time: length, controller cadence (*Tick* duration/timestep), and how many datapoints to capture per *Experimental Run*.

Cmdline Syntax

`T{duration}[.K{ticks_per_sec}][.N{n_datapoints}`

- `duration` - Duration of timesteps in *seconds* (not timesteps).
- `ticks_per_sec` - How many times each controller will be run per second.
- `n_datapoints` - # datapoints per *Experimental Run* to be captured; the capture interval (if configurable) should be adjusted in *Project*-derived class from the platform "Experiment setup class" (e.g., *`sierra.plugins.platform.argos.variables.exp_setup.ExpSetup`* for *ARGoS*).

Examples

- `exp_setup.T1000`: Experimental run will be 1,000 seconds long and have 1,000*5=5,000 timesteps, with default (50) # datapoints.
- `exp_setup.T2000.N100`: Experimental run will be 2,000 seconds long and have 2,000*5=10,000 timesteps, with 100 datapoints (1 every 20 seconds/100 timesteps).
- `exp_setup.T10000.K10`: Experimental run will be 10,000 seconds long, and have 10,000*10=100,000 timesteps with default (50) # datapoints.
- `exp_setup.T10000.K10.N100`: Experimental run will be 10,000 seconds long, and have 10,000*10=100,000 timesteps, with 100 datapoints (one every 100 seconds/1,000 timesteps).

7.7 Rendering

SIERRA's capabilities for rendering video outputs are detailed in this section. SIERRA can render frames (images) into videos from 3 sources:

- Those captured using `--platform-vc`, details *here*.
- Those imagized from project CSV output files via `--project-imagizing` using `--project-rendering` details *here*.
- Inter-experiment heatmaps from bivariate batch criteria `--bc-rendering`, details *here*.

Note: Using BOTH the platform and project rendering capabilities simultaneously IS possible (i.e., passing `--platform-vc` and `--project-rendering` during stage 3), but discouraged unless you have multiple terrabytes of disk space available. `--exp-range` is your friend.

7.7.1 Platform Visual Capture

SIERRA can direct some platforms to capture frames during experiments. `--platform-vc` assumes that:

- **ffmpeg** is installed/can be found by the shell. Checked during stage 3 if *imagizing* is run.

This is applicable to the following platforms:

- *ARGoS*, selected via `--platform=platform.argos`.

Important: If the selected platform usually runs headless, then this option will probably slow things down a LOT, so if you use it, `--n-runs` should probably be low, unless you have gobs of computing power available.

ARGos Visual Capture

Visual capture in *ARGoS* is done via frame capturing while running, and then the captured images stitched together into videos during stage 4.

During stage 1 `--platform-vc` causes the ARGoS Qt/OpenGL <visualization> subtree to be added to the `--template-input-file` when generating experimental inputs; it is removed otherwise. If <visualization> already exists, it is removed and re-created. During stage 1 SIERRA assumes that:

- **Xvfb** is installed/can be found by the shell (checked). This is needed to get ARGoS to "render" its simulation into an offscreen buffer which we can output to a file.

During stage 4, `--platform-vc` causes frames captured during stage 2 to be stitched together into a unique video file using **ffmpeg** (precise command configurable via `--render-cmd-opts`), and output to <batch_root>/videos/<exp>.

7.7.2 Project Rendering

Projects can generate CSV files residing in subdirectories within the `main.run_metrics_leaf` (see *Main Configuration*) directory (directory path set on a per `--project` basis) for each experimental run, in addition to generating CSV files residing directly in the `main.run_metrics_leaf.` directory. SIERRA can then render these CSV files into *Heatmap* graphs, and stitch these images together to make videos.

To use, do the following:

1. Pass `--project-imagizing` during stage 3. When passed, the CSV files residing each subdirectory under the `main.run_metrics_leaf` directory (no recursive nesting is allowed) in each run are treated as snapshots of 2D or 3D data over time, and will be averaged together across runs and then turn into image files suitable for video rendering in stage 4. The following restrictions apply:

   - A common stem with a unique numeric ID is required for each CSV must be present for each CSV.
   - The directory name within `main.run_metrics_leaf` must be the same as the stem for each CSV file in that directory. For example, if the directory name was `swarm-distribution` under `main.run_metrics_leaf` then all CSV files within that directory must be named according to `swarm-distribution/swarm-distributionXXXXX.csv`, where XXXXX is any length numeric prefix (possibly preceded by an underscore or dash).

   Important: Averaging the image CSV files and generating the images for each experiment does not happen automatically as part of stage 3 because it can take a LONG time and is idempotent. You should only pass

`--project-imagizing` the first time you run stage 3 after running stage 2 (unless you are getting paid by the hour).

2. Pass `--project-vc` during stage 4 after running imagizing via `--project-imagizing` during stage 3, either on the same invocation or a previous one. SIERRA will take the imagized CSV files previously created and generate a set of a videos in `<batch_root>/videos/<exp>` for each experiment in the batch which was run.

Important: Rendering the imagized CSV does not happen automatically every time as part of stage 4 because it can take a LONG time and is idempotent. You should only pass `--project-vc` the first time you run stage 4 after having run stage 3 with `--project-vc` (unless you are getting paid by the hour).

7.7.3 Batch Criteria Rendering

For bivariate batch criteria, if inter-experiment heatmaps are generated, they can be stitched together to make videos of how the two variables of interest affect some aspect of behavior over time.

To use, do the following:

1. Pass `--bc-rendering` during stage 4 when at least inter-experiment heatmap is generated. SIERRA will take the generated PNG files previously created and generate a set of a videos in `<batch_root>/videos/<heatmap name>` for each heatmap.

Important: Rendering the heatmaps CSV does not happen automatically every time as part of stage 4 because it can take a LONG time and is idempotent. You should only pass `--bc-rendering` the first time you run stage 4 (unless you are getting paid by the hour).

7.8 Pipeline Stage 5

The main idea of this pipeline stage is to "collate" the results of one or more *Summary .csv* files present in different *Batch Experiments* into a *Inter-Batch .csv* file, and then use that file to generate graphs. Any *Summary .csv* that is present in multiple *Batch Experiments* can be used during stage 5! This gives this pipeline stage tremendous flexibility as a camera-ready graph generation tool.

In general, stage 5 is always run separate from stages 1-4 (i.e., a separate SIERRA invocation), because the options are quite different, but you don't *have* to do this.

Important: You *cannot* use this stage before successfully running stage 4 for each of the *Batch Experiments* you want to include on the final graph.

Warning: Because SIERRA never deletes stuff for you, running stage 5 is *NOT* idempotent. Running the same stage 5 invocation comparing 3 controllers in a single scenario (for example) could result in linegraphs containing 3,6,9,..., lines with duplicated data. In general, you want to delete the directories generated by stage 5 between successive runs. See *SIERRA Runtime Directory Tree* for details on what directories are generated.

7.8.1 Intra-Scenario Comparison

Intra-scenario comparison compares the of experiments using one or more controllers on the same `--scenario`. To use it, you need to pass the following options to SIERRA (see *Command Line Interface* for documentation):

- `--scenario-comparison`
- `--bc-univar` or `--bc-bivar`
- `--dist-stats` (to get statistics generated during stage 3 to show up on the final graph).

Other `--plot-*` options providing for fine-grained control of the generated graphs may also be of interest.

For YAML configuration, see *Intra-Scenario Comparison*.

7.8.2 Inter-Scenario Comparison

Inter-scenario comparison compares the same `--controller` across multiple `--scenarios`. To use it, you need to pass the following options to SIERRA when running stage 5 (see *Command Line Interface* for documentation):

- `--controller-comparison`
- `--bc-univar` or `--bc-bivar`
- `--dist-stats` (to get statistics generated during stage 3 to show up on the final graph).

Other `--plot-*` options providing for fine-grained control of the generated graphs may also be of interest.

For YAML configuration, see *Inter-Scenario Comparison*.

CHAPTER

EIGHT

CONFIGURATION AND EXTENSION TUTORIALS

This page contains tutorials to setup and/or extend SIERRA according to your needs.

8.1 Creating a New SIERRA Project

1. Decide what *Platform* your *Project* will target. Currently, there is no way to share projects across platforms; any common code will have to be put into common python files and imported as needed.
2. Create the directory which will hold your *Project*. The directory your project must be on *SIERRA_PLUGIN_PATH* or SIERRA won't be able to find your project. For example, if your project is `proj-awesome`, and that directory is in `projects` as `/path/to/projects/proj-awesome`, then `/path/to/projects` needs to be on *SIERRA_PLUGIN_PATH*.
3. Create the following directory structure within your project directory (or copy and modify the one from an existing project).

   Important: Once you create the directory structure below you need to INSTALL your project with pip so that not only can SIERRA find it, but so can the python interpreter. If you don't want to do that, then you need to put your project plugin directory on *PYTHONPATH*. Otherwise, you won't be able to use your project plugin with SIERRA.

   - `config/` - Plugin YAML configuration root. This directory is required for all projects. Within this directory, the following files are used (not all files are required when running a stage that utilizes them):
     - `main.yaml` - Main SIERRA configuration file. This file is required for all pipeline stages. See *Main Configuration* for documentation.
     - `controllers.yaml` - Configuration for controllers (input file/graph generation). This file is required for all pipeline stages. See *Main Configuration* for documentation.
     - `intra-graphs-line.yaml` - Configuration for intra-experiment linegraphs. This file is optional. If it is present, graphs defined in it will be added to those specified in `<sierra>/core/config/intra-graphs-line.yaml`, and will be generated if stage 4 is run. See *Graph Configuration* for documentation.
     - `intra-graphs-hm.yaml` - Configuration for intra-experiment heatmaps. This file is optional. If it is present, graphs defined in it will be added to those specified in `<sierra>/core/config/intra-graphs-hm.yaml`, and will be generated if stage 4 is run. See *Graph Configuration* for documentation.
     - `inter-graphs.yaml` - Configuration for inter-experiment graphs. This file is optional. If it is present, graphs defined in it will be added to those specified in `<sierra>/core/config/`

`inter-graphs-line.yaml`, and will be generated if stage 4 is run. See *Graph Configuration* for documentation.

  - `stage5.yaml` - Configuration for stage5 controller comparisons. This file is required if stage 5 is run, and optional otherwise. See *Stage 5 Configuration* for documentation.

  - `models.yaml` - Configuration for intra- and inter-experiment models. This file is optional. If it is present, models defined and enabled in it will be run before stage 4 intra- and/or inter-experiment graph generation, if stage 4 is run. See *Adding Models to your SIERRA Project* for documentation.

- `generators/` - Classes to enable SIERRA to generate changes to template XML files needed by your project. This directory is required for all SIERRA projects.

  - `scenario_generator_parser.py` - Contains the parser for parsing the contents of `--scenario` into a dictionary which can be used to configure experiments. This file is required. See *Per-Scenario Configuration* for documentation.

  - `scenario_generators.py` - Specifies classes and functions to enable SIERRA to generate XML file modifications to the `--template-input-file` based on what is passed as `--scenario` on the cmdline. This file is required. See *Per-Scenario Configuration* for documentation.

  - `exp_generators.py` - Contains extensions to the per-*Experiment* and per-*Experimental Run* configuration that SIERRA performs. See *Per-Experimental Run Configuration* for documentation. This file is optional.

- `variables/` - Additional variables (including batch criteria) defined by the plugin/project that can be directly or indirectly used by the `--batch-criteria` and `--scenario` cmdline arguments. This directory is optional.

- `models/` - Theoretical models that you want to run against empirical data from experimental runs (presumably to compare predictions with). This directory is optional. See *Adding Models to your SIERRA Project* for documentation.

- `cmdline.py` - Specifies cmdline extensions specific to the plugin/project. This file is required. See *Extending the SIERRA Cmdline* for documentation.

#. Configure your project so SIERRA understands how to generate *Experimental Run* inputs and process outputs correctly by following *Main Configuration*.

1. Define graphs to be generated from *Experiment* outputs by following *Graph Configuration*. Strictly speaking this is optional, but automated graph generation during stage 4 is one of the most useful parts of SIERRA, so its kind of silly if you don't do this.

2. Setup your `--template-input-file` appropriately by following *Template Input Files*.

8.1.1 Optional Steps

1. Define additional batch criteria to investigate variables of interest specific to your project by following *Create A New Batch Criteria*.

2. Define one or more *Models* to run to compare with empirical data.

3. Add additional per-run configuration such as unique output directory names, random seeds, etc. in various python files referenced by `scenario_generators.py` and `exp_generators.py` beyond what is required for `--scenario`. SIERRA can't set stuff like this up in a project agnostic way.

8.2 Extending the SIERRA Cmdline

At a minimum, all *Projects* must define the `--scenario` and `--controller` cmdline arguments to interact with the SIERRA core; other cmdline arguments can be added in the same manner. For example, you might want to control aspects of experiment generation outside of those controlled by `--scenario`, but only some of the time. To add additional cmdline options to the SIERRA, follow the steps below.

1. Create `cmdline.py` in your *Project* directory.
2. Create a `Cmdline` class which inherits from the SIERRA cmdline as follows:

```
import typing as tp
import argparse

import sierra.core.cmdline as cmd

class Cmdline(cmd.CoreCmdline):
    def __init__(self,
                 bootstrap: tp.Optional[argparse.ArgumentParser],
                 stages: tp.List[int],
                 for_sphinx: bool):
        super().__init__(bootstrap, stages)

    def init_multistage(self):
        super().init_multistage(for_sphinx)

        self.multistage.add_argument("--scenario",
                                     help="""

                                     A cool scenario argument.

                                     """ + self.stage_usage_doc([1, 2, 3, 4]))

        self.multistage.add_argument("--controller",
                                     help="""

                                     A cool controller argument.

                                     """ + self.stage_usage_doc([1, 2, 3, 4]))

    def init_stage1(self):
        super().init_stage1(for_sphinx)

        self.stage1.add_argument("--my-stage1-argument",
                                   help="""

                                   An argument which is intended for stage 1 use only.

                                   """ + self.stage_usage_doc([1]),
                                   type=int,
                                   default=None)

    def init_stage2(self):
        ...
```

(continues on next page)

(continued from previous page)

```
    def init_stage3(self):
        ...
    def init_stage4(self):
        ...
    def init_stage5(self):
        ...

    @staticmethod
    def cmdopts_update(cli_args: argparse.Namespace, cmdopts: types.Cmdopts):
        updates = {
        'scenario': cli_args.scenario,
        'controller': cli_args.controller,
        'my_stage1_argument': cli_args.my_stage1_argument

  }
  cmdopts.update(updates)
```

All of the `init_XXstage()` functions are optional; if they are not given then SIERRA will use the version in *CoreCmdline*.

Important: Whichever `init_XXstage()` functions you define must have a call to `super().init__XXstage()` as the first statement, otherwise the cmdline arguments defined by SIERRA will not be setup properly.

The `cmdopts_update()` function inserts the parsed cmdline arguments into the main `cmdopts` dictionary used throughout SIERRA. Keys can have any name, though in general it is best to make them the same as the name of the argument (principle of least surprise).

The following command line arguments must be (a) present on all platforms and (b) inserted into `cmdopts` by the `cmdopts_update()` function, or SIERRA will crash/not work properly:

- `--exp-setup`

3. Create a `CmdlineValidator` class to validate the additional cmdline arguments you pass (can be empty class if no additional validation is needed). Generally this should be used for things like "if X is passed then Y must also be passed".

```
class CmdlineValidator(cmd.CoreCmdlineValidator):
    def __call__(self, args: argparse.Namespace) -> None:
        assert args.my_stage1_argument is not None,\
             "--my-stage1-argument must be passed!"
```

The `__call__()` function is passed the `argparse` object resulting from parsing the arguments, which can be used as you would expect to perform checks. All checks should be assertions.

8.3 Main Configuration

The three main required configuration files that you must define so that SIERRA knows how to interact with your project are shown below for each platform SIERRA supports:

`config/main.yaml`

ARGoS

An example main configuration file for the ARGoS platform:

```
# Per-project configuration for SIERRA core. This dictionary is
# mandatory.
sierra:
  # Configuration for each experimental run. This dictionary is
  # mandatory for all experiments.
  run:
    # The directory within each experimental run's working
    # directory which will contain the metrics output as by the
    # run. This key is mandatory for all experiments. Can be
    # anything; this is the interface link between where the
    # project code outputs things and where SIERRA looks for
    # outputs.
    run_metrics_leaf: 'metrics'

    # The name of the shared library where project code can be
    # found, sans file extension. Must include
    # 'lib'. Optional. If not present, then SIERRA will use
    # ``--project`` as the name of the library to tell ARGoS to
    # use.
    library_name: 'libawesome'

  # Configuration for performance measures. This key is mandatory
  # for all experiments. The value is the location of the .yaml
  # configuration file for performance measures. It is a separate
  # config file so that multiple scenarios within a single
  # project which define performance measures in different ways
  # can be easily accommodated without copy-pasting.
  perf: 'perf-config.yaml'
```

ROS1+Gazebo

An example main configuration file for the ROS1+Gazebo platform:

```
# Per-project configuration for SIERRA core. This dictionary is
# mandatory.
sierra:
  # Configuration for each experimental run. This dictionary is
  # mandatory for all experiments.
  run:
    # The directory within each experimental run's working
    # directory which will contain the metrics output as by the
    # run. This key is mandatory for all experiments. Can be
    # anything; this is the interface link between where the
    # project code outputs things and where SIERRA looks for
```

(continues on next page)

(continued from previous page)

```
    # outputs.
    run_metrics_leaf: 'metrics'


  # Configuration for performance measures. This key is mandatory
  # for all experiments. The value is the location of the .yaml
  # configuration file for performance measures. It is a separate
  # config file so that multiple scenarios within a single
  # project which define performance measures in different ways
  # can be easily accommodated without copy-pasting.
  perf: 'perf-config.yaml'

# Configuration specific to the ROS platforms. This
# dictionary is required if that platform is selected, and
# optional otherwise.
ros:
  # The list of robot configuration for the platform that you want
  # SIERRA to support (that actual list of robots supported by the
  # platform can be much larger).
  robots:
    # The name of a supported robot which can be passed to
    # ``--robot``. Can be any valid python string, and does not
    # have to match whatever the robot is called in its ROS
    # package.
    governator:
      # The ROS package that the robot can be found in. This
      # package must be on your ROS_PACKAGE_PATH or SIERRA will
      # fail at runtime. This key is required.
      pkg: "my_ros_package"

      # The name of your robot within its ROS package. Used by
      # SIERRA to add the ROS node to load its description. This
      # key is required.
      model: "terminator"

      # The name of a variation of the base robot model. This key
      # is optional. If present, the actual name of the robot in
      # the ROS package used to source the robot description is
      # constructed via <model>_<model_variant>
      model_variant: "T1000"

      # The robot prefix which will be prepended to the robot's
      # numeric ID to form its UUID. E.g., for robot 14, its UUID
      # will be <prefix>14. This is used by SIERRA to create
      # unique namespaces for each robot's nodes so that all their
      # ROS topics are unique.
      prefix: "T"

    myrobot2:
      ...
```

ROS1+Robot

An example main configuration file for the ROS1+Robot platform:

```
 # Per-project configuration for SIERRA core. This dictionary is
 # mandatory.
 sierra:
   # Configuration for each experimental run. This dictionary is
   # mandatory for all experiments.
   run:
     # The directory within each experimental run's working
     # directory which will contain the metrics output as by the
     # run. This key is mandatory for all experiments. Can be
     # anything; this is the interface link between where the
     # project code outputs things and where SIERRA looks for
     # outputs.
     run_metrics_leaf: 'metrics'

   # Configuration for performance measures. This key is mandatory
   # for all experiments. The value is the location of the .yaml
   # configuration file for performance measures. It is a separate
   # config file so that multiple scenarios within a single
   # project which define performance measures in different ways
   # can be easily accommodated without copy-pasting.
   perf: 'perf-config.yaml'

# Configuration specific to the ROS platforms. This
# dictionary is required if that platform is selected, and
# optional otherwise.
ros:
  # The list of robot configuration for the platform that you want
  # SIERRA to support (that actual list of robots supported by the
  # platform can be much larger).
  robots:
    # The name of a supported robot which can be passed to
    # ``--robot``. Can be any valid python string, and does not
    # have to match whatever the robot is called in its ROS
    # package.
    turtlebot3:
      # The robot prefix which will be prepended to the robot's
      # numeric ID to form its UUID. E.g., for robot 14, its UUID
      # will be <prefix>14. This is used by SIERRA to create
      # unique namespaces for each robot's nodes so that all their
      # ROS topics are unique (if desired).
      prefix: "tb3_"

      # The name of the setup script to source on login to each
      # robot to setup the ROS environment. This key is optional.
      setup_script: "$HOME/setup.bash"

    myrobot2:
      ...
```

ARGoS

An example main configuration file for the ARGoS platform:

```yaml
# Per-project configuration for SIERRA core. This dictionary is
# mandatory.
sierra:
  # Configuration for each experimental run. This dictionary is
  # mandatory for all experiments.
  run:
    # The directory within each experimental run's working
    # directory which will contain the metrics output as by the
    # run. This key is mandatory for all experiments. Can be
    # anything; this is the interface link between where the
    # project code outputs things and where SIERRA looks for
    # outputs.
    run_metrics_leaf: 'metrics'

    # The name of the shared library where project code can be
    # found, sans file extension. Must include
    # 'lib'. Optional. If not present, then SIERRA will use
    # ``--project`` as the name of the library to tell ARGoS to
    # use.
    library_name: 'libawesome'

  # Configuration for performance measures. This key is mandatory
  # for all experiments. The value is the location of the .yaml
  # configuration file for performance measures. It is a separate
  # config file so that multiple scenarios within a single
  # project which define performance measures in different ways
  # can be easily accommodated without copy-pasting.
  perf: 'perf-config.yaml'
```

ROS1+Gazebo

An example main configuration file for the ROS1+Gazebo platform:

```yaml
# Per-project configuration for SIERRA core. This dictionary is
# mandatory.
sierra:
  # Configuration for each experimental run. This dictionary is
  # mandatory for all experiments.
  run:
    # The directory within each experimental run's working
    # directory which will contain the metrics output as by the
    # run. This key is mandatory for all experiments. Can be
    # anything; this is the interface link between where the
    # project code outputs things and where SIERRA looks for
    # outputs.
    run_metrics_leaf: 'metrics'


  # Configuration for performance measures. This key is mandatory
  # for all experiments. The value is the location of the .yaml
  # configuration file for performance measures. It is a separate
  # config file so that multiple scenarios within a single
  # project which define performance measures in different ways
```

(continues on next page)

(continued from previous page)

```
   # can be easily accommodated without copy-pasting.
   perf: 'perf-config.yaml'

# Configuration specific to the ROS platforms. This
# dictionary is required if that platform is selected, and
# optional otherwise.
ros:
  # The list of robot configuration for the platform that you want
  # SIERRA to support (that actual list of robots supported by the
  # platform can be much larger).
  robots:
    # The name of a supported robot which can be passed to
    # ``--robot``. Can be any valid python string, and does not
    # have to match whatever the robot is called in its ROS
    # package.
    governator:
      # The ROS package that the robot can be found in. This
      # package must be on your ROS_PACKAGE_PATH or SIERRA will
      # fail at runtime. This key is required.
      pkg: "my_ros_package"

      # The name of your robot within its ROS package. Used by
      # SIERRA to add the ROS node to load its description. This
      # key is required.
      model: "terminator"

      # The name of a variation of the base robot model. This key
      # is optional. If present, the actual name of the robot in
      # the ROS package used to source the robot description is
      # constructed via <model>_<model_variant>
      model_variant: "T1000"

      # The robot prefix which will be prepended to the robot's
      # numeric ID to form its UUID. E.g., for robot 14, its UUID
      # will be <prefix>14. This is used by SIERRA to create
      # unique namespaces for each robot's nodes so that all their
      # ROS topics are unique.
      prefix: "T"

    myrobot2:
      ...
```

ROS1+Robot

An example main configuration file for the ROS1+Robot platform:

```
# Per-project configuration for SIERRA core. This dictionary is
# mandatory.
sierra:
  # Configuration for each experimental run. This dictionary is
  # mandatory for all experiments.
  run:
    # The directory within each experimental run's working
```

(continues on next page)

(continued from previous page)

```
    # directory which will contain the metrics output as by the
    # run. This key is mandatory for all experiments. Can be
    # anything; this is the interface link between where the
    # project code outputs things and where SIERRA looks for
    # outputs.
    run_metrics_leaf: 'metrics'

  # Configuration for performance measures. This key is mandatory
  # for all experiments. The value is the location of the .yaml
  # configuration file for performance measures. It is a separate
  # config file so that multiple scenarios within a single
  # project which define performance measures in different ways
  # can be easily accommodated without copy-pasting.
  perf: 'perf-config.yaml'

# Configuration specific to the ROS platforms. This
# dictionary is required if that platform is selected, and
# optional otherwise.
ros:
  # The list of robot configuration for the platform that you want
  # SIERRA to support (that actual list of robots supported by the
  # platform can be much larger).
  robots:
    # The name of a supported robot which can be passed to
    # ``--robot``. Can be any valid python string, and does not
    # have to match whatever the robot is called in its ROS
    # package.
    turtlebot3:
      # The robot prefix which will be prepended to the robot's
      # numeric ID to form its UUID. E.g., for robot 14, its UUID
      # will be <prefix>14. This is used by SIERRA to create
      # unique namespaces for each robot's nodes so that all their
      # ROS topics are unique (if desired).
      prefix: "tb3_"

      # The name of the setup script to source on login to each
      # robot to setup the ROS environment. This key is optional.
      setup_script: "$HOME/setup.bash"

    myrobot2:
      ...
```

`config/perf-config.yaml`

Configuration for summary performance measures. Does not have to be named `perf-config.yaml`, but must match whatever is specified in `main.yaml`.

```
# Root key is required.
perf:

  # Is the performance measure for the project inverted, meaning that
  # lower values are better. This key is optional; defaults to False if
  # omitted.
```

(continues on next page)

(continued from previous page)

```
  inverted: true

  # The CSV file under ``statistics/`` for each experiment which
  # contains the averaged performance information for the
  # experiment. This key is required.
  intra_perf_csv: 'block-transport.csv'

  # The CSV column within ``intra_perf_csv`` which is the
  # temporally charted performance measure for the experiment. This key
  # is required.
  intra_perf_col: 'cum_avg_transported'
```

Additional fields can be added to this dictionary as needed to support custom performance measures,graph generation, or batch criteria as needed. For example, you could add fields to this dictionary as a lookup table of sorts for a broader range of cmdline configuration (i.e., using it to make the cmdline syntax for the a new batch criteria much nicer).

Configuration for summary performance measures. Does not have to be named `perf-config.yaml`, but must match whatever is specified in `main.yaml`.

```
# Root key is required.
perf:

  # Is the performance measure for the project inverted, meaning that
  # lower values are better. This key is optional; defaults to False if
  # omitted.
  inverted: true

  # The CSV file under ``statistics/`` for each experiment which
  # contains the averaged performance information for the
  # experiment. This key is required.
  intra_perf_csv: 'block-transport.csv'

  # The CSV column within ``intra_perf_csv`` which is the
  # temporally charted performance measure for the experiment. This key
  # is required.
  intra_perf_col: 'cum_avg_transported'
```

Additional fields can be added to this dictionary as needed to support custom performance measures,graph generation, or batch criteria as needed. For example, you could add fields to this dictionary as a lookup table of sorts for a broader range of cmdline configuration (i.e., using it to make the cmdline syntax for the a new batch criteria much nicer).

`config/controllers.yaml`

Configuration for robot controllers.

Root level dictionaries: varies; project dependent. Each root level dictionary is treated as the name of a *Controller Category* when `--controller` is parsed. For example, if you pass `--controller=mycategory.FizzBuzz` to SIERRA, then you need to have a root level dictionary `mycategory` defined in `controllers.yaml`.

A complete YAML configuration for a *Controller Category* `mycategory` and a controller `FizzBuzz` is shown below, separated by platform. This configuration specifies that all graphs in the categories of `LN_MyCategory1`, `LN_MyCategory2`, `HM_MyCategory1`, `HM_MyCategory2` are applicable to `FizzBuzz`, and should be generated if the necessary *Experiment* output files exist. The `LN_MyCategory1`, `LN_MyCategory2` graph categories are common to multiple controllers in this project, while the `HM_MyCategory1`, `HM_MyCategory2` *graph categories* are specific to the `FizzBuzz` controller.

ARGoS

```
my_base_graphs:
  - LN_MyCategory1
  - LN_MyCategory2

mycategory:

  # Changes to existing XML attributes in the template ``.argos``
  # file for *all* controllers in the category, OR changes to
  # existing tags for *all* controllers in the template ``.xml``
  # file.  This is usually things like setting ARGoS loop functions
  # appropriately, if required. Each change is formatted as a list
  # with paths to parent tags specified in the XPath syntax.
  #
  # - [parent tag, attr, value] for changes to existing XML
  #   attributes.
  #
  # - [parent tag, child tag, value] for changes to existing tags
  #
  # - [parent tag, child tag, attr] for adding new tags. When adding
  #   tags the attr string is passed to eval() to turn it into a
  #   python dictionary.
  #
  # The ``xml`` section and subsections are optional. If
  # ``--platform-vc`` is passed, then this section should be used to
  # specify any changes to the XML needed to setup the selected
  # platform for frame capture/video rendering by specifying the QT
  # visualization functions to use.
  xml:
    tag_change:
      - ['.//loop-functions/parent', 'child', 'stepchild']
    attr_change:
      - ['.//loop-functions', 'label', 'my_category_loop_functions']
      - ['.//qt-opengl/user_functions', 'label', 'my_category_qt_loop_functions']
    tag_add:
      - ...
      - ...

  # Under ``controllers`` is a list of controllers which can be
  # passed as part of ``--controller`` when invoking SIERRA, matched
  # by ``name``. Any controller-specific XML attribute changes can
  # be specified here, with the same syntax as the changes for the
  # controller category (``mycategory`` in this example). As above,
  # you can specify sets of changes to existing XML attributes,
  # changes to existing XML tags to set things up for a specific
  # controller, or adding new XML tags.
  controllers:
    - name: FizzBuzz
      xml:
        attr_change:

          # The ``__CONTROLLER__`` tag in the
```

(continues on next page)

(continued from previous page)

```
        # ``--template-input-file`` is REQUIRED to allow SIERRA to
        # unambiguously set the "library" attribute of the
        # controller.
        - ['.//controllers', '__CONTROLLER__', 'FizzBuzz']


    # Sets of graphs common to multiple controller categories can
    # be inherited with the ``graphs_inherit`` dictionary (they
    # are added to the ``graphs`` dictionary). This dictionary is
    # optional, but handy to reduce repetitive declarations and
    # typing. see the YAML docs for details on how to include
    # named lists inside other lists.
    graphs_inherit:
      - *my_base_graphs

    # Specifies a list of graph categories from inter- or
    # intra-experiment ``.yaml`` configuration which should be
    # generated for this controller, if the necessary input CSV
    # files exist.
    graphs: &FizzBuzz_graphs
      - HM_MyCategory1
      - HM_MyCategory2
```

ROS1+Gazebo and ROS1+Robot

```
my_base_graphs:
  - LN_MyCategory1
  - LN_MyCategory2

mycategory:
  # Changes to existing XML attributes in the template ``.launch``
  # file for *all* controllers in the category, OR changes to
  # existing tags for *all* controllers in the template ``.launch``
  # file.  Each change is formatted as a list with paths to parent
  # tags specified in the XPath syntax.
  #
  # - [parent tag, attr, value] for changes to existing XML
  #   attributes.
  #
  # - [parent tag, child tag, value] for changes to existing tags
  #
  # - [parent tag, child tag, attr] for adding new tags. When adding
  #   tags the attr string is passed to eval() to turn it into a
  #   python dictionary.
  #
  # The ``xml`` section and subsections are optional. If
  # ``--platform-vc`` is passed, then this section should be used to
  # specify any changes to the XML needed to setup ROS1+Gazebo for
  # visual capture.
  #
  # When adding new tags the ``__UUID__`` string can be included in
  # the parent tag or child tag fields, which has two
```

(continues on next page)

(continued from previous page)

```
  # effects. First, it is expanded to the robot prefix (namespace in
  # ROS terminology) + the robot's ID to form a UUID for the
  # robot. Second, the tag is added not just once overall, but once
  # for each robot in each experimental run. This is useful to set
  # per-robot parameters specific to a given controller outside of
  # the parameters controller via batch criteria or SIERRA
  # variables (e.g., launching nodes to bringup sensors on the
  # robot that are not launched by default/by the controller entry
  # point).
  xml:
    tag_change:
      - ...
    attr_change:
      - ...
    tag_add:
      - ...

  # Under ``controllers`` is a list of controllers which can be
  # passed as part of ``--controller`` when invoking SIERRA, matched
  # by ``name``. Any controller-specific XML attribute changes can
  # be specified here, with the same syntax as the changes for the
  # controller category (``mycategory`` in this example). As above,
  # you can specify sets of changes to existing XML attributes,
  # changes to existing XML tags to set things up for a specific
  # controller, or adding new XML tags.
  #
  # When adding new tags the ``__UUID__`` string can be included in
  # the parent tag or child tag fields, which has two
  # effects. First, it is expanded to the robot prefix (namespace in
  # ROS terminology) + the robot's ID to form a UUID for the
  # robot. Second, the tag is added not just once overall, but once
  # for each robot in each experimental run. This is useful to set
  # per-robot parameters specific to a given controller outside of
  # the parameters controller via batch criteria or SIERRA variables
  # (e.g., launching nodes to bringup sensors on the robot that are
  # not launched by default/by the controller entry point).
  controllers:
    - name: FizzBuzz
      xml:
        tag_add:
          - [".//launch/group/[@ns='__UUID__']", 'param', "{'name': 'topic_name', 'value
↪':'mytopic'}"]



      # Sets of graphs common to multiple controller categories can
      # be inherited with the ``graphs_inherit`` dictionary (they
      # are added to the ``graphs`` dictionary). This dictionary is
      # optional, but handy to reduce repetitive declarations and
      # typing. see the YAML docs for details on how to include
      # named lists inside other lists.
      graphs_inherit:
```

(continues on next page)

(continued from previous page)

```
      - *my_base_graphs

    # Specifies a list of graph categories from inter- or
    # intra-experiment ``.yaml`` configuration which should be
    # generated for this controller, if the necessary input CSV
    # files exist.
    graphs: &FizzBuzz_graphs
      - HM_MyCategory1
      - HM_MyCategory2
```

Configuration for robot controllers.

Root level dictionaries: varies; project dependent. Each root level dictionary is treated as the name of a *Controller Category* when `--controller` is parsed. For example, if you pass `--controller=mycategory.FizzBuzz` to SIERRA, then you need to have a root level dictionary `mycategory` defined in `controllers.yaml`.

A complete YAML configuration for a *Controller Category* `mycategory` and a controller `FizzBuzz` is shown below, separated by platform. This configuration specifies that all graphs in the categories of `LN_MyCategory1`, `LN_MyCategory2`, `HM_MyCategory1`, `HM_MyCategory2` are applicable to `FizzBuzz`, and should be generated if the necessary *Experiment* output files exist. The `LN_MyCategory1`, `LN_MyCategory2` graph categories are common to multiple controllers in this project, while the `HM_MyCategory1`, `HM_MyCategory2` *graph categories* are specific to the `FizzBuzz` controller.

ARGoS

```
my_base_graphs:
  - LN_MyCategory1
  - LN_MyCategory2

mycategory:

  # Changes to existing XML attributes in the template ``.argos``
  # file for *all* controllers in the category, OR changes to
  # existing tags for *all* controllers in the template ``.xml``
  # file.  This is usually things like setting ARGoS loop functions
  # appropriately, if required. Each change is formatted as a list
  # with paths to parent tags specified in the XPath syntax.
  #
  # - [parent tag, attr, value] for changes to existing XML
  #   attributes.
  #
  # - [parent tag, child tag, value] for changes to existing tags
  #
  # - [parent tag, child tag, attr] for adding new tags. When adding
  #   tags the attr string is passed to eval() to turn it into a
  #   python dictionary.
  #
  # The ``xml`` section and subsections are optional. If
  # ``--platform-vc`` is passed, then this section should be used to
  # specify any changes to the XML needed to setup the selected
  # platform for frame capture/video rendering by specifying the QT
  # visualization functions to use.
  xml:
    tag_change:
```

(continues on next page)

(continued from previous page)

```
      - ['.//loop-functions/parent', 'child', 'stepchild']
    attr_change:
      - ['.//loop-functions', 'label', 'my_category_loop_functions']
      - ['.//qt-opengl/user_functions', 'label', 'my_category_qt_loop_functions']
    tag_add:
      - ...
      - ...

  # Under ``controllers`` is a list of controllers which can be
  # passed as part of ``--controller`` when invoking SIERRA, matched
  # by ``name``. Any controller-specific XML attribute changes can
  # be specified here, with the same syntax as the changes for the
  # controller category (``mycategory`` in this example). As above,
  # you can specify sets of changes to existing XML attributes,
  # changes to existing XML tags to set things up for a specific
  # controller, or adding new XML tags.
  controllers:
    - name: FizzBuzz
      xml:
        attr_change:

          # The ``__CONTROLLER__`` tag in the
          # ``--template-input-file`` is REQUIRED to allow SIERRA to
          # unambiguously set the "library" attribute of the
          # controller.
          - ['.//controllers', '__CONTROLLER__', 'FizzBuzz']


      # Sets of graphs common to multiple controller categories can
      # be inherited with the ``graphs_inherit`` dictionary (they
      # are added to the ``graphs`` dictionary). This dictionary is
      # optional, but handy to reduce repetitive declarations and
      # typing. see the YAML docs for details on how to include
      # named lists inside other lists.
      graphs_inherit:
        - *my_base_graphs

      # Specifies a list of graph categories from inter- or
      # intra-experiment ``.yaml`` configuration which should be
      # generated for this controller, if the necessary input CSV
      # files exist.
      graphs: &FizzBuzz_graphs
        - HM_MyCategory1
        - HM_MyCategory2
```

ROS1+Gazebo and ROS1+Robot

```
my_base_graphs:
  - LN_MyCategory1
  - LN_MyCategory2

mycategory:
```

(continues on next page)

(continued from previous page)

```
  # Changes to existing XML attributes in the template ``.launch``
  # file for *all* controllers in the category, OR changes to
  # existing tags for *all* controllers in the template ``.launch``
  # file.  Each change is formatted as a list with paths to parent
  # tags specified in the XPath syntax.
  #
  # - [parent tag, attr, value] for changes to existing XML
  #   attributes.
  #
  # - [parent tag, child tag, value] for changes to existing tags
  #
  # - [parent tag, child tag, attr] for adding new tags. When adding
  #   tags the attr string is passed to eval() to turn it into a
  #   python dictionary.
  #
  # The ``xml`` section and subsections are optional. If
  # ``--platform-vc`` is passed, then this section should be used to
  # specify any changes to the XML needed to setup ROS1+Gazebo for
  # visual capture.
  #
  # When adding new tags the ``__UUID__`` string can be included in
  # the parent tag or child tag fields, which has two
  # effects. First, it is expanded to the robot prefix (namespace in
  # ROS terminology) + the robot's ID to form a UUID for the
  # robot. Second, the tag is added not just once overall, but once
  # for each robot in each experimental run. This is useful to set
  # per-robot parameters specific to a given controller outside of
  # the parameters controller via batch criteria or SIERRA
  # variables (e.g., launching nodes to bringup sensors on the
  # robot that are not launched by default/by the controller entry
  # point).
  xml:
    tag_change:
      - ...
    attr_change:
      - ...
    tag_add:
      - ...

  # Under ``controllers`` is a list of controllers which can be
  # passed as part of ``--controller`` when invoking SIERRA, matched
  # by ``name``. Any controller-specific XML attribute changes can
  # be specified here, with the same syntax as the changes for the
  # controller category (``mycategory`` in this example). As above,
  # you can specify sets of changes to existing XML attributes,
  # changes to existing XML tags to set things up for a specific
  # controller, or adding new XML tags.
  #
  # When adding new tags the ``__UUID__`` string can be included in
  # the parent tag or child tag fields, which has two
  # effects. First, it is expanded to the robot prefix (namespace in
  # ROS terminology) + the robot's ID to form a UUID for the
```

(continues on next page)

(continued from previous page)

```
  # robot. Second, the tag is added not just once overall, but once
  # for each robot in each experimental run. This is useful to set
  # per-robot parameters specific to a given controller outside of
  # the parameters controller via batch criteria or SIERRA variables
  # (e.g., launching nodes to bringup sensors on the robot that are
  # not launched by default/by the controller entry point).
  controllers:
    - name: FizzBuzz
      xml:
        tag_add:
          - [".//launch/group/[@ns='__UUID__']", 'param', "{'name': 'topic_name', 'value
↪':'mytopic'}"]



      # Sets of graphs common to multiple controller categories can
      # be inherited with the ``graphs_inherit`` dictionary (they
      # are added to the ``graphs`` dictionary). This dictionary is
      # optional, but handy to reduce repetitive declarations and
      # typing. see the YAML docs for details on how to include
      # named lists inside other lists.
      graphs_inherit:
        - *my_base_graphs

      # Specifies a list of graph categories from inter- or
      # intra-experiment ``.yaml`` configuration which should be
      # generated for this controller, if the necessary input CSV
      # files exist.
      graphs: &FizzBuzz_graphs
        - HM_MyCategory1
        - HM_MyCategory2
```

8.4 Graph Configuration

This page has the following sections:

- How to create a new *Graph Category*
- How to define a new graph within a *Graph Category* to generate from *Experimental Run* outputs.
- How to "activate" the new graph so that it will be generated from experimental run outputs where applicable.
- How to generate additional graphs during stage 4 beyond those possible with the SIERRA core.

8.4.1 Create A New Graph Category

Add a root level dictionary to one of the following YAML configuration files:

- `<project>/config/intra-graphs-line.yaml` for intra-experiment line graphs. Dictionaries must start with `LN_`.
- `<project>/config/intra-graphs-hm.yaml` for intra-experiment heatmaps. Dictionaries must start with `HM_`.
- `<project>/config/inter-graphs-line.yaml` for inter-experiment line graphs. Dictionaries must start with `LN_`.
- `<project>/config/inter-graphs-hm.yaml` for inter-experiment heatmaps. Dictionaries must start with `HM_`.

An example `intra-graphs-line.yaml`, defining two categories of linegraphs:

```
graphs:
  LN_mycategory1:
    - ...
    - ...
    - ...

  LN_mycategory2:
    - ...
    - ...
    - ...
```

`intra-graphs-hm.yaml` and `inter-graphs-line.yaml` have identical structures.

Important: The `graphs` dictionary *must* be at the root of all `.yaml` files containing graph configuration.

Important: Because SIERRA tells matplotlib to use LaTeX internally to generate graph labels, titles, etc., the standard LaTeX character restrictions within strings apply to all fields (e.g., '#' is illegal but '#' is OK).

8.4.2 Add A New Intra-Experiment Graph To An Existing Category

There are two types of intra-experiment graphs: linegraphs and heatmaps, and each has their own config file (details of each is below).

Linegraphs

Linegraphs are appropriate if:

- The data you want to graph can be represented by a line (i.e. is one dimensional in some way).
- The data you want to graph can be obtained from a single .csv file (multiple columns in the same CSV file can be graphed simultaneously).

LN_XXX YAML Config

Unless stated otherwise, all keys are mandatory.

```yaml
LN_mycategory:
  # The filename (no path) of the CSV within the experimental run output
  # directory for an experimental run, sans the CSV extension.
  - src_stem: 'foo'

  # The filename (no path) of the graph to be generated
  # (extension/image type is determined elsewhere). This allows for multiple
  # graphs to be generated from the same CSV file by plotting different
  # combinations of columns.
  - dest_stem: 'bar'

  # List of names of columns within the source CSV that should be
  # included on the plot. Must match EXACTLY (i.e. no fuzzy matching). Can be
  # omitted to plot all columns within the CSV.
  - cols:
      - 'col1'
      - 'col2'
      - 'col3'
      - '...'

  # The title the graph should have. LaTeX syntax is supported (uses
  # matplotlib after all). Optional.
  - title: 'My Title'

  # List of names of the plotted lines within the graph. Can be
  # omitted to set the legend for each column to the name of the column
  # in the CSV.
  - legend:
      - 'Column 1'
      - 'Column 2'
      - 'Column 3'
      - '...'

  # The label of the X-axis of the graph. Optional.
  - xlabel: 'X'

  # The label of the Y-axis of the graph. Optional.
  - ylabel: 'Y'
```

Heatmaps

Heatmaps are appropriate if:

- The data you want to graph is two dimensional (e.g. a spatial representation of the arena is some way).

`HM_XXX` YAML Config

Unless stated otherwise, all keys are mandatory.

```
graphs:
  # The filename (no path) of the CSV within the output directory
  # for an experimental run to look for the column(s) to plot, sans the CSV
  # extension.
  - src_stem: 'fooCSV'

  # The title the graph should have. LaTeX syntax is supported (uses
  # matplotlib after all). Optional.
  - title: 'My Title'

  # The type of interpolation to use. Defaults to 'nearest' if omitted.
  - interpolation: 'nearest'

  # The Z colorbar label to use. Optional.
  - zlabel: 'My colorbar label'
```

8.4.3 How to Add A New Inter-Experiment Graph

Linegraphs

Inter-experiment linegraphs are appropriate if:

- The data you want to graph can be represented by a line (i.e. is one dimensional in some way).
- The data you want to graph can be obtained from a single column from a single CSV file.
- The data you want to graph requires comparison between multiple experiments in a batch.

`LN_XXX` YAML Config

See same as intra-experiment linegraphs, EXCEPT:

Each inter-experiment linegraph has an additional optional boolean field `summary` which determines if the generated graph is a *SummaryLineGraph* or a *StackedLineGraph* (default if omitted).

Heatmaps

Inter-experiment heatmaps are appropriate if:

- You are using bivariate batch criteria.
- The data you want to graph can be represented by a line (i.e. is one dimensional in some way).
- The data you want to graph can be obtained from a single column from a single CSV file.
- The data you want to graph requires comparison between multiple experiments in a batch.

New in version 1.2.20.

`HM_XXX` YAML Config

Same as intra-experiment heatmaps.

8.4.4 How to Activate New Graph Category

If you added a new *Graph Category*, it will not automatically be used to generate graphs for existing or new controllers. You will need to modify the `<project>/config/controllers.yaml` file to specify which controllers your new category of graphs should be generated for. See *Main Configuration* for details.

8.5 Stage 5 Configuration

This page has the following sections:

- *Intra-Scenario Comparison*: How to generate comparison graphs for a set of controllers which have all been run on the *same* scenario(s).
- *Inter-Scenario Comparison*: How to generate comparison graphs for a single controller which has been run across *multiple* scenarios.

All configuration for stage 5 is in `<project>/config/stage5.yaml` file. This file is mandatory for running stage 5, and optional otherwise.

8.5.1 Intra-Scenario Comparison

Intra-scenario comparison compares the results of multiple controllers on the same `--scenario`. An example `stage5.yaml`, defining two different comparison graphs is shown below. Supports univariate and bivariate batch criteria.

Note: Any collated CSV/graph can be used as a comparison graph! This includes any additional CSVs that a project creates on its own/by extending SIERRA via hooks.

Graph YAML Config

Unless stated otherwise, all keys are required.

```yaml
# Intra-scenario comparison: For a set of controllers which have all been run
# in the same scenario (or set of scenarios).
intra_scenario:
  # Which intra-scenario comparison graphs should be generated.
  graphs:
    # The filename (no path, extension) of the .csv within the collated .csv
    # output directory for each batch experiment which contains the
    # information to collate across controllers/scenarios.
    #
    # The src_stem must match the dest_stem from an inter-experiment line
    # graph in order to generate the comparison graph!
    #
    # Note that if you are using bivariate batch criteria + intra-scenario
    # comparison, you *may* have to append the interval # to the end of the
    # stem, because collated 2D CSV files form a temporal sequence over the
    # duration of the experiment. If you forget to do this, you will get a
    # warning and no graph will be generated, because SIERRA won't know which
    # 2D csv you want to us as source. Frequently you are interested in
    # steady state behavior, so putting 'n_intervals - 1' is desired.
    #
    # If the src_stem is from a CSV you generated outside of the SIERRA
    # core/via a hook, then this restriction does not apply.
    - src_stem: PM-ss-raw

      # The filename (no path, extent) of the .csv file within the
      # controller/scenario comparison directory in ``--sierra-root``
      # (outside of the directories for each controller/scenario!) which
      # should contain the data collated from each batch experiment. I
      # usually put a prefix such as ``cc`` (controller comparison) to help
      # distinguish these graphs from the collated graphs in stage 4.
      dest_stem: cc-PM-ss-raw

      # The title the graph should have. This cannot be computed from the
      # batch criteria in general in stage 5, because you can comparison
      # results across scenarios which use different batch criteria. This key
      # is optional.
      title: ''

      # The Y or Z label of the graph (depending on the type of comparison
      # graph selected on the cmdline). This key is optional.
      label: 'Avg. Object Collection Rate'

      # For bivariate batch criteria, select which criteria should be on the
      # X axis if the comparison graphs are linegraphs. 0=criteria1,
      # 1=criteria2. Ignored for univariate batch criteria or if heatmaps are
      # selected for as the comparison type. This key is optional, and
      defaults to 0 if omitted.
      primary_axis: 0
```

(continues on next page)

(continued from previous page)

```
    # The experiments from each batch experiment which should be included on
    # the comparison graph. This is useful to exclude exp0 (for example), if
    # you are interested in behavior under non-ideal conditions and exp0
    # contains behavior under ideal conditions as the "base case" in the
    # batch. Syntax is parsed as a python slice, so as ``X:Y`` as you would
    # expect. This key is optional, and defaults to ``:`` if omitted.
     include_exp: '2:'

   # scalability
   - src_stem: PM-ss-scalability-parallel-frac
     dest_stem: cc-PM-ss-scalability-parallel-frac
     title: ''
     label: 'Scalability Value'
     primary_axis: 0
     include_exp: '2:'
```

8.5.2 Inter-Scenario Comparison

Inter-scenario comparison compares the same `--controller` across multiple `--scenarios`. An example `stage5.yaml`, defining a comparison graphs is shown below. Only supports univariate batch criteria.

Note: Any collated CSV/graph can be used as a comparison graph! This includes any additional CSVs that a project creates on its own/by extending SIERRA via hooks.

Graph YAML Config

Same syntax and meaning as the configuration for intra-scenario comparison graphs.

```
inter_scenario:
  graphs:
    # raw performance
    - src_stem: PM-ss-raw
      dest_stem: cc-PM-ss-raw
      title: ''
      label: 'Avg. Object Collection Rate'
      primary_axis: 0
      include_exp: '2:'
```

8.6 Template Input Files

8.6.1 Template Input Files Passed to SIERRA

Examples of the structure/required content of the XML file passed to SIERRA via `--template-input-file` for each supported *Platform* are below.

ARGoS

For the purposes of illustration we will use `--template-input-file=sample.argos` and a controller `MyController`:

```
<argos-configuration>
   ...
   <controllers>
      <__CONTROLLER__>
         ...
         <params>
            <task_alloc>
               <mymethod threshold="17"/>
            </task_alloc>
         </params>
      </__CONTROLLER__>
   </controllers>
   ...
<argos-configuration>
```

See *XML Content Requirements* for usage/description of the `__CONTROLLER__` tag.

ROS1 (Using parameter server)

This is for the following ROS1-based platforms:

- ROS1+Gazebo
- ROS1+Robot

For the purposes of illustration we will use `--template-input-file=sample.launch`:

```
<ros-configuration>
   <master>
      ...
      <param name="metrics/directory" value="/path/to/dir"/>
      ...
   </master>
   <robot>
      ...
      <param name="task_alloc/mymethod/threshold" value="17"/>
      <param name="motion/random_walk" value="0.1"/>
      ...
   </robot>
<ros-configuration>
```

ROS1 (Using <params> tag)

This is for the following ROS-based platforms:

- ROS1+Gazebo
- ROS1+Robot

For the purposes of illustration we will use `--template-input-file=sample.launch`:

```
<ros-configuration>
  <master>
     ...
  </master>
```

(continues on next page)

(continued from previous page)

```
   <robot>
      ...
   </robot>
   <params>
      <metrics directory="/path/to/dir"/>
      <task_alloc>
         <mymethod threshold="17"/>
      </task_alloc>
      <motion>
         <random_walk prob="0.1"/>
      </motion>
   </params>
</ros-configuration>
```

8.6.2 Post-Processed Template Input Files

SIERRA may insert additional XML tags and split the processed template input file into multiple template files, depending on the platform. The results of this processing are shown below for each supported *Platform*. No additional modifications beyond those necessary to use the platform with SIERRA are shown (i.e., no *Batch Criteria* modifications).

Any of the following may be inserted:

- A new tag for the configured random seed.
- A new tag for the configured experiment length in seconds.
- A new tag for the configured # robots.
- A new tag for the controller rate (ticks per second).
- A new tag for the path to a second XML file containing all controller XML configuration.

ARGoS

```
<argos-configuration>
   ...
   <controllers>
      <MyController>
         ...
         <params>
            <task_alloc>
               <mymethod threshold="17"/>
            </task_alloc>
         </params>
      </MyController>
   </controllers>
   ...
<argos-configuration>
```

No tags are insert by SIERRA input the input `.argos` file.

ROS (Using parameter server)

Input `sample.launch` file is split into multiple files:

- `sample_runX_robotY.launch` containing the `<robot>` tag in the original input file, which is changed to `<launch>`. This has all nodes and configuration which is robot-specific and/or will be launched on each robot `Y` for each run `X`.
- `sample_master_runX.launch` containing the `<master>` tag in the original input file, which is changed to `<launch>`. This has all the nodes and configuration which is ROS master-specific and will be launched on the SIERRA host machine (which where the ROS master will be set to) for each run `X`.

`sample_run0_robot0.launch` file:

```
<launch>
   ...
   <param name="task_alloc/mymethod/threshold" value="17"/>
   <param name="motion/random_walk" value="0.1"/>
   ...
   <group ns='sierra'>
      <param name="experiment/length" value="1234"/>
      <param name="experiment/random_seed" value="5678"/>
      <param name="experiment/param_file" value="/path/to/file"/>
      <param name="experiment/n_robots" value="123"/>
      <param name="experiment/ticks_per_sec" value="5"/>
   </group>
   ...
</launch>
```

`sample_master_run0.launch` file:

```
<launch>
   ...
   <param name="metrics/directory" value="/path/to/dir"/>
   ...
   <group ns='sierra'>
      <node
         name="sierra_timekeeper"
         pkg="sierra_rosbridge"
         type="sierra_timekeeper.py"
         required="true"
         output="screen"/>
      <param name="experiment/length" value="1234"/>
      <param name="experiment/random_seed" value="5678"/>
      <param name="experiment/param_file" value="/path/to/file"/>
      <param name="experiment/n_robots" value="123"/>
      <param name="experiment/ticks_per_sec" value="5"/>
   </group>
   ...
</launch>
```

ROS (Not using parameter server)

Input `sample.launch` file is split into multiple files:

- `sample_runX_robotY.launch` containing the `<robot>` tag in the original input file, which is changed to `<launch>`. This has all nodes and configuration which is robot-specific and/or will be launched on each robot `Y` for each run `X`.
- `sample_master_runX.launch` containing the `<master>` tag in the original input file, which is changed to `<launch>`. This has all the nodes and configuration which is ROS master-specific and will be launched on the

SIERRA host machine (which where the ROS master will be set to) for each run `X`.

- `sample_runX.params` containing the <params> tag in the original input file, which is written out as a common file to use to share parameters between the robots and the ROS master for each run `X`.

Processed `sample_run0_robot0.launch` file:

```
<launch>
   ...
   <group ns='sierra'>
      <param name="experiment/length" value="1234"/>
      <param name="experiment/random_seed" value="5678"/>
      <param name="experiment/param_file" value="/path/to/file"/>
      <param name="experiment/n_robots" value="123"/>
      <param name="experiment/ticks_per_sec" value="5"/>
   </group>
   ...
</launch>
```

Processed `sample_run0_master.launch` file:

```
<launch>
   ...
   <group ns='sierra'>
      <node
        name="sierra_timekeeper"
        pkg="sierra_rosbridge"
        type="sierra_timekeeper.py"
        required="true"
        output="screen"/>
      <param name="experiment/length" value="1234"/>
      <param name="experiment/random_seed" value="5678"/>
      <param name="experiment/param_file" value="/path/to/file"/>
      <param name="experiment/n_robots" value="123"/>
      <param name="experiment/ticks_per_sec" value="5"/>
   </group>
   ...
</launch>
```

Processed `sample_run0.params` file:

```
<params>
   <metrics directory="/path/to/dir"/>
   <motion>
      <random_walk prob="0.1"/>
   </motion>
   <task_alloc>
      <mymethod threshold="17"/>
   </task_alloc>
</params>
```

8.7 Generator Configuration

8.7.1 Per-Scenario Configuration

To enable SIERRA to generate experiment definitions based on the `--scenario` you specify, you need to:

1. Create `generators/scenario_generator_parser.py` in your `--project` directory.

   Within this file, you must define the `ScenarioGeneratorParser` class with the following signature:

```
import typing as tp

class ScenarioGeneratorParser():
    def __init__(self):
        ...

    def to_scenario_name(self, args) -> tp.Optional[str]:
        """
        Parse the scenario generator from cmdline arguments into a
        string. Should return None if ``args`` is None (stage5).
        """
        ...

    def to_dict(self, scenario_name: str) -> str:
        """
        Given a string (presumably a result of an earlier cmdline parse),
        parse it into a dictionary of components: arena_x, arena_y,
        arena_z, scenario_tag

        which specify the X,Y,Z dimensions of the arena a unique tag/short
        scenario name unique among all scenarios for the project, which is
        used which creating the SIERRA runtime directory structure.
        """
        ...
```

2. Create `generators/scenario_generators.py` in your `--project` directory.

   Within this file, you must define at least one function (and presumably one or more classes representing the scenarios you want to test with), which is `gen_generator_name()`, which takes the `--scenario` argument and returns the string of the class name within `scenario_generators.py` that SIERRA should use to generate scenario definitions for your experiments:

```
def gen_generator_name(scenario_name: str) -> str:
    ...
```

   Each generator class within `scenario_generators.py` must define the `generate()` function like so:

```
class MyScenarioGenerator():
    ...

def generate(self):
    ...
```

8.7.2 Per-Experimental Run Configuration

In order to hook into SIERRA stage 1 experiment generation (doing so is optional), you need to:

1. Create `generators/exp_generators.py` in your `--project` directory.
2. Define a `ExpRunDefUniqueGenerator` class in this file, overriding the `generate()` function with your customizations. Your class really should be derived from a platform generator (e.g., *PlatformExpRunDefUniqueGenerator*) and override the `generate()` function, though you don't have to.

8.8 Create A New Batch Criteria

If you have a new experimental variable that you have added to your C++ library, **AND** which is exposed via the input `.xml` file, then you need to do the following to get it to work with SIERRA as a *Batch Criteria*:

1. Make your variable (`MyVar` in this tutorial) inherit from *sierra.core.variables.batch_criteria.UnivarBatchCriteria* and place your `my_var.py` file under `<project>/variables/`. The class defined in `my_var.py` should be a "base class" version of your variable, and therefore should take in parameters, and NOT have any hardcoded values in it anywhere (i.e., rely on dynamic class creation via the `factory()` function). This is to provide maximum flexibility to those using SIERRA, so that they can create *any* kind of instance of your variable, and not just the ones you have made pre-defined classes for.
2. Define the abstract functions from *sierra.core.variables.batch_criteria.UnivarBatchCriteria*. Most are straight forward to understand from the documentation, but the XML manipulation ones warrant more explanation.

   In order to change attributes, add/remove tags, you will need to understand the XPath syntax for search in XML files; tutorial is *here _ln-xpath*.

   `get_attr_changelist()` - Given whatever parameters that your variable was passed during initialization (e.g. the boundaries of a range you want to vary it within), produce a list of sets, where each set is all changes that need to be made to the `.xml` template file in order to set the value of your variable to something. Each change is a *AttrChange* object, that takes the following arguments in its constructor:

   1. XPath search path for the **parent** of the attribute that you want to modify.
   2. Name of the attribute you want to modify within the parent element.
   3. The new value as a string (integers will throw an exception).

   `gen_tag_rmlist()` - Given whatever parameters that your variable was passed during initialization, generate a list of sets, where each set is all tags that need to be removed from the `.xml` template file in order to set the value of your variable to something.

   Each change is a *TagRm* object that takes the following arguments in its constructor:

   1. XPath search path for the **parent** of the tag that you want to remove.
   2. Name of the tag you want to remove within the parent element.

   `gen_tag_addlist()` - Given whatever parameters that your variable was passed during initialization, generate a list of sets, where each set is all tags that need to be added to the `.xml` template file.

   Each change is a *TagAdd* object that takes the following arguments in its constructor:

   1. XPath search path for the **parent** of the tag that you want to add.
   2. Name of the tag you want to add within the parent element.
   3. A dictionary of (attribute, value) pairs to create as children of the tag when creating the tag itself.

3. Define the parser for your variable in order to parse the command line string defining your batch criteria into a dictionary of attributes that can then be used by the `factory()`. The parser can be defined anywhere, though it must be able to be used in the `factory()` function. The parse class must conform to the following interface:

```
class MyVarParser():
    ...
def __call__(self, cli_arg: str) -> dict:
    ...
```

   It must be callable with a single argument which is whatever was passed to `--batch-criteria`. See *sierra.core.variables.population_size.Parser* for a simple example of this.

4. Define a factory function to dynamically create classes from the base class definition of `MyVar` in `my_var.py`. It must have the following signature:

```
import pathlib

def factory(cli_arg: str,
            main_config: dict,
            batch_input_root: pathlib.path,
            **kwargs) -> MyVar:
"""
Arguments:

    cli_arg: The string of the your batch criteria/variable you
             have defined that was passed on the command line via
             ``--batch-criteria``.
    main_config: The main YAML configuration dictionary
    (``<project>/config/main.yaml``).

    batch_input_root: The directory where the experiment directories are
                      to be created.

    **kwargs: Additional arguments required by this batch criteria. This
    may be used during stage 5 to pass the ``--scenario`` if needed.

"""
```

   This function should do the following:

   1. Call the parser for your variable, as defined above.
   2. Return a custom instance of your class that is named according to the specific batch criteria string passed on the command line which inherits from `MyVar` variable base class you defined above, and that has an `__init__()` function that calls the `__init__()` function of your base variable. To dynamically create a new class which is derived from your `MyVar` class, you can use the `type()` function.

   See `<sierra>/plugins/argos/variables/population_size.py` for a simple example of this.

8.9 SIERRA Hooks

SIERRA allows a number of different elements of its pipeline to be customized and extend by a *Project*. To override part of a stage of the SIERRA pipeline, you must create a `pipeline/stageX` directory structure in your project directory, where `stageX` is the stage you want to override part of.

Note: This is the most advanced feature of SIERRA, and strange things may happen or things might not work right if you don't call the `super()` version of whatever functions you are overriding in your hook.

Warning: Don't try to override parts of the pipeline which are not listed below. It will probably not work, and even if it does there is every chance that it will break stuff.

8.9.1 Multi-Stage Hooks

Base Batch Criteria

Suppose you want to extend one of the following core SIERRA classes to add additional attributes/methods you want to be accessible to all *Batch Criteria* in your project:

- *BatchCriteria*
- *UnivarBatchCriteria*
- *BivarBatchCriteria*

To do this, do the following:

1. Create `variables/batch_criteria.py` in the root directory for your project.
2. Override one or more of the classes above, and SIERRA will then select your version of said override classes when running. Exactly where SIERRA is looking/what module it uses when a given class is requested can be seen with `--log-level=TRACE`.

Warning: Don't override or extend any of the interfaces! It will causes static analysis and/or runtime errors.

8.9.2 Stage 3 Hooks

Experimental Run Collation

In order to generate additional inter-experiment graphs, you have to also collate additional CSV files by:

1. Create `pipeline/stage3/run_collator.py`.
2. Override the *sierra.core.pipeline.stage3.run_collator.ExperimentalRunCSVGatherer* class:

```
import sierra.core.pipeline.stage3.run_collator as run_collator
import pathlib

class ExperimentalRunCSVGatherer(run_collator.ExperimentalRunCSVGatherer):
    def gather_csvs_from_run(self,
```

(continues on next page)

(continued from previous page)

```
                                run_output_root: pathlib.Path) -> tp.Dict[tp.Tuple[str,
↪ str], pd.DataFrame]:
        ...
```

8.9.3 Stage 4 Hooks

Tiered YAML Config

Suppose you have some graphs which are common to multiple SIERRA projects, and you don't want to have to duplicate the graph definitions in the `.yaml` files. You can put those definitions in a single location and then add them to the `.yaml` graph definitions that are unique to the *Project* as follows:

1. Create `pipeline/stage4/yaml_config_loader.py`.
2. Override the *sierra.core.pipeline.stage4.yaml_config_loader.YAMLConfigLoader* class:

```
import sierra.core.pipeline.stage4.yaml_config_loader as ycl
from sierra.core import types

class YAMLConfigLoader(ycl.YAMLConfigLoader):
    def __call__(self, cmdopts: types.Cmdopts) -> tp.Dict[str, types.YAMLDict]:
        ...
```

Intra-Experiment Graph Generation

You way want to extend the set of graphs which is generated for each experiment in the batch, based on what batch criteria is selected, or for some other reason. To do so:

1. Create `pipeline/stage4/intra_exp_graph_generator.py`.
2. Override the *sierra.core.pipeline.stage4.inter_exp_graph_generator.InterExpGraphGenerator* class:

```
import sierra.core.pipeline.stage4 as stage4

class IntraExpGraphGenerator(stage4.intra_exp_graph_generator.
↪IntraExpGraphGenerator):
    def __call__(self, criteria: bc.IConcreteBatchCriteria) -> None:
        ...
```

Inter-Experiment Graph Generation

You way want to extend the set of graphs which is generated across each each experiment in the batch (e.g., to create graphs of summary performance measures). To do so:

1. Create `pipeline/stage4/inter_exp_graph_generator.py`.
2. Override the *sierra.core.pipeline.stage4.inter_exp_graph_generator.InterExpGraphGenerator* class:

```
import sierra.core.pipeline.stage4 as stage4
import sierra.core.batch_criteria as bc

class InterExpGraphGenerator(stage4.inter_exp_graph_generator.
↪InterExpGraphGenerator):
    def __call__(self, criteria: bc.IConcreteBatchCriteria) -> None:
        ...
```

8.10 Adding Models to your SIERRA Project

Models can be written in any language, but if they aren't python, you will have to write some python bindings to translate the inputs/outputs into things that SIERRA can understand/is expecting.

8.10.1 Create A New Intra-Experiment Model

1. Look at:

   - *IConcreteIntraExpModel1D*
   - *IConcreteIntraExpModel2D*
   - *IConcreteInterExpModel1D*

   to determine if one of the model types SIERRA already supports will work for you. If one will, great! Otherwise, you'll probably need to add a new one.

2. Define your models and/or their bindings in one or more `.py` files under `<project>/models`. Not all python files in `<project>/models` have to contain models! See *Model File Requirements* below for details about what needs to be in python files which *do* contain models/model bindings.

3. Enable your model. If you add a new intra-experiment model, it will not automatically be run during stage 4. You will need to modify the `<project>/config/models.yaml` file to enable your model. See *Model Configuration* below for details.

4. Run your model during stage 4. You also need to pass `--models-enable` (models are not enabled by default); see *Command Line Interface* for more details.

Model File Requirements

1. Must be under `<project>/models`.

2. All model `.py` files must define an `available_models()` function which takes a string argument for the type of model [`intra`, `inter`] and returns a list of the names of the intra- and inter-experiment models present in the file. This allows the user flexibility to group multiple related models together in the same file, rather than requiring 1 model per `.py` file.

3. All model classes in the model `.py` must implement an appropriate interface interface of `<sierra>/models/interface.py`, depending on whether the model is 1D or 2D, and whether or not it is an intra-experiment or inter-experiment model.

Model Configuration

With <project>/config/models.yaml, each model has the following YAML fields under a root models dictionary:

- pyfile - The name of the python file with the models/ directory for the project where the model name be found. This also serves as the name of the model within SIERRA.

Each model specified in models.yaml can take any number of parameters of any type specified as extra fields in the YAML file; they will be parsed and passed to the model constructor as part of config. See below for an example.

config/models.yaml

Root level dictionaries:

- models - List of enabled models. This dictionary is mandatory for all experiments during stage 4 (if models are enabled via --models-enabled``, that is).

Example YAML Config

```
models:

  # The name of the python file under ``project/models`` containing one or
  # more models meeting the requirements of one of the model interfaces:
  # :class:`~sierra.core.models.interface.IConcreteIntraExpModel1D`
  # :class:`~sierra.core.models.interface.IConcreteIntraExpModel2D`
  # :class:`~sierra.core.models.interface.IConcreteInterExpModel1D`

  - pyfile: 'my_model1'
    fooparam: 42
    barparam: "also 42"

  - pyfile: 'my_model2'
    param1: 17
    param2: "abc"
    ...
  - ...
```

Any other parameters/dictionaries/etc needed by a particular model can be added to the list above and they will be passed through to the model's constructor.

8.10.2 Creating A New Inter-Experiment Model

TODO!

8.11 HPC Cluster Setup

These instructions assume you already have SIERRA working with your project on your local machine. If you haven't done that yet–shoo!

This setup applies to the following SIERRA HPC cluster environments:

- *PBS HPC Plugin*
- *SLURM HPC Plugin*

8.11.1 ARGoS Setup on HPC Clusters

The steps to properly configure the C++ libraries for *ARGoS* and your *Project* for use with SIERRA in one of the above environments are:

1. Build ARGoS natively for your cluster for maximum efficiency.

   Note: If your HPC cluster is 1/2 Intel chips and 1/2 AMD chips, you may want to compile ARGoS twice, natively on each chipset. If you do this, you can set *SIERRA_ARCH* prior to invoking SIERRA so that the correct ARGoS commands can be generated, depending on what the chipset is for the nodes you request for your HPC job.

2. Your project `.so` should be built natively on each different type of compute node SIERRA might be run on, just like ARGOS, for maximum efficiency with large swarms. You can use *ARGOS_PLUGIN_PATH* (which is not modified by SIERRA) to specify where the library should be loaded from (e.g., using *SIERRA_ARCH* as the switch in your script which invokes SIERRA).

Once ARGoS/your C++ code has been built, you can setup SIERRA:

1. Install SIERRA package by following the instructions in *SIERRA Installation Reference*.
2. Verify GNU **`parallel`** is installed; if it is not installed, ask your cluster admin to install it for you.
3. Clone plugin for whatever project you are going to use somewhere on your *SIERRA_PLUGIN_PATH*. SIERRA will refuse to do anything useful if there is no project selected. The repository should be cloned into a directory with the EXACT name you want it to be callable with on the cmdline via `--project`.
4. Read the documentation for *HPC Execution Environment Plugins*, and select and appropriate plugin to use. Be sure to define all necessary environment variables!!

8.11.2 GazeboS Setup on HPC Clusters

TBD.

8.12 HPC Local Setup

To set up SIERRA for HPC on your local machine, follow *Getting Started With SIERRA*.

8.13 Creating a New Platform Plugin

For the purposes of this tutorial, I will assume you are creating a new *Platform Plugin* `matrix`, and the code for that plugin lives in `$HOME/git/plugins/platform/matrix`.

If you are creating a new platform, you have two options.

1. Create a stand-alone platform, providing your own definitions for all of the necessary functions/classes below.
2. Derive from an existing platform by simply calling the "parent" platform's functions/calling in your derived definitions except when you need to actually extend them (e.g., to add support for a new HPC plugin)

In either case, the steps to actually create the code are below.

8.13.1 Create the Code

Create the following filesystem structure and content in `$HOME/git/plugins/platform/matrix`. Each file is required; any number of additional files can be included.

`plugin.py`

Within this file, you may define the following classes, which must be named **EXACTLY** as specified, otherwise SIERRA will not detect them. If you omit a required class, you will get an error on SIERRA startup. If you try to use a part of SIERRA which requires an optional class you omitted, you will get a runtime error.

Table 1: Platform Plugin Classes

Class	Required?	Conforms to interface?
ExpConfigurer	Yes	*IExpConfigurer*
CmdlineParserGenerator	Yes	*ICmdlineParserGenerator*
ParsedCmdlineConfigurer	No	*IParsedCmdlineConfigurer*
ExpRunShellCmdsGenerator	No	*IExpRunShellCmdsGenerator*
ExpShellCmdsGenerator	No	*IExpShellCmdsGenerator*
ExecEnvChecker	No	*IExecEnvChecker*

Within this file, you may define the following functions, which must be named **EXACTLY** as specified, otherwise SIERRA will not detect them. If you try to use a part of SIERRA which requires an optional function you omitted, you will get a runtime error.

Table 2: Platform Plugin Functions

Function	Required?	Purpose
population_size_from_def()	Yes	During stage 2, on some platforms (e.g., ROS) you need to be able to extract the # of robots that will be used for a given *Experiment*/*Experimental Run* in order to correctly setup the execution environment. So, given the experimental definition object, extract the # robots that will be used.
population_size_from_pickle()	Yes	During stage 5, there is no way for SIERRA to know how many robots were used in a cross-platform way, because different platforms can write different XML tags capturing the # robots used for a specific experiment. So, given an unpickled experiment definition, extract the # robots used.
robot_prefix_extract()	No	Return the alpha-numeric prefix that will be prepended to each robot's numeric ID to create a UUID for the robot. Not needed by all platforms; if not needed by your platform, return None.
arena_dims_from_criteria()	No	Get a list of the arena dimensions used in each generated experiment. Only needed if the dimensions are not specified on the cmdline, which can be useful if the batch criteria involves changing them; e.g., evaluating behavior with different arena shapes.
pre_exp_diagnostics()	No	Log any INFO-level diagnostics to stdout before a given *Experiment* is run. Useful to echo important execution environment configuration to the terminal as a sanity check.

Below is a sample/skeleton `plugin.py` to use as a starting point.

```
import typing as tp
import argparse

import implements

from sierra.core.experiment import bindings, xml, definition
from sierra.core.variables import batch_criteria as bc

@implements.implements(bindings.IParsedCmdlineConfigurer)
class CmdlineParserGenerator():
  def __call__() -> argparse.ArgumentParser:
      """A class that conforms to
      :class:`~sierra.core.experiment.bindings.ICmdlineParserGenerator`.
      """
      # As an example, assuming this platform can run on HPC
      # environments. Initialize all stages and return the initialized
      # parser to SIERRA for use.
      parser = hpc.HPCCmdline([-1, 1, 2, 3, 4, 5]).parser
      return cmd.PlatformCmdline(parents=[parser],
                                 stages=[-1, 1, 2, 3, 4, 5]).parser


@implements.implements(bindings.IParsedCmdlineConfigurer)
class ParsedCmdlineConfigurer():
    """A class that conforms to
    :class:`~sierra.core.experiment.bindings.IParsedCmdlineConfigurer`.
    """
```

(continues on next page)

(continued from previous page)

```
@implements.implements(bindings.IExpShellCmdsGenerator)
class ExpShellCmdsGenerator():
    """A class that conforms to
    :class:`~sierra.core.experiment.bindings.IExpShellCmdsGenerator`.
    """

@implements.implements(bindings.IExpRunShellCmdsGenerator)
class ExpRunShellCmdsGenerator():
    """A class that conforms to
    :class:`~sierra.core.experiment.bindings.IExpRunShellCmdsGenerator`.
    """

@implements.implements(bindings.IExecEnvChecker)
class ExecEnvChecker():
    """A class that conforms to
    :class:`~sierra.core.experiment.bindings.IExecEnvChecker`.
    """

@implements.implements(bindings.IExpConfigurer)
class ExpConfigurer():
    """A class that conforms to
    :class:`~sierra.core.experiment.bindings.IExpConfigurer`.
    """

@implements.implements(bindings.IExpRunConfigurer)
class ExpRunConfigurer():
    """A class that conforms to
    :class:`~sierra.core.experiment.bindings.IExpRunConfigurer`.
    """

def population_size_from_pickle(exp_def: tp.Union[xml.AttrChangeSet,
                                                  xml.TagAddList]) -> int:
    """
    Size can be obtained from added tags or changed attributes; platform
    specific.

    Arguments:

        exp_def: *Part* of the pickled experiment definition object.

  """

def population_size_from_def(exp_def: definition.XMLExpDef) -> int:
    """

    Arguments:

        exp_def: The *entire* experiment definition object.

  """
```

(continues on next page)

(continued from previous page)

```
def robot_prefix_extract(main_config: types.YAMLDict,
                         cmdopts: types.Cmdopts) -> str:
    """

    Arguments:

        main_config: Parsed dictionary of main YAML configuration.

        cmdopts: Dictionary of parsed command line options.
  """

def pre_exp_diagnostics(cmdopts: types.Cmdopts,
                        logger: logging.Logger) -> None:
    """
    Arguments:

        cmdopts: Dictionary of parsed command line options.

        logger: The logger to log to.

  """

def arena_dims_from_criteria(criteria: bc.BatchCriteria) -> tp.List[utils.ArenaExtent]:
    """
    Arguments:

       criteria: The batch criteria built from cmdline specification
    """
```

Within this file, you may define the following classes, which must be named **EXACTLY** as specified, otherwise SIERRA will not detect them. If you omit a required class, you will get an error on SIERRA startup. If you try to use a part of SIERRA which requires an optional class you omitted, you will get a runtime error.

Table 3: Platform Plugin Classes

Class	Required?	Conforms to interface?
ExpConfigurer	Yes	*IExpConfigurer*
CmdlineParserGenerator	Yes	*ICmdlineParserGenerator*
ParsedCmdlineConfigurer	No	*IParsedCmdlineConfigurer*
ExpRunShellCmdsGenerator	No	*IExpRunShellCmdsGenerator*
ExpShellCmdsGenerator	No	*IExpShellCmdsGenerator*
ExecEnvChecker	No	*IExecEnvChecker*

Within this file, you may define the following functions, which must be named **EXACTLY** as specified, otherwise SIERRA will not detect them. If you try to use a part of SIERRA which requires an optional function you omitted, you will get a runtime error.

Table 4: Platform Plugin Functions

Function	Required?	Purpose
population_size_from_def()	Yes	During stage 2, on some platforms (e.g., ROS) you need to be able to extract the # of robots that will be used for a given *Experiment*/*Experimental Run* in order to correctly setup the execution environment. So, given the experimental definition object, extract the # robots that will be used.
population_size_from_pickle()	Yes	During stage 5, there is no way for SIERRA to know how many robots were used in a cross-platform way, because different platforms can write different XML tags capturing the # robots used for a specific experiment. So, given an unpickled experiment definition, extract the # robots used.
robot_prefix_extract()	No	Return the alpha-numeric prefix that will be prepended to each robot's numeric ID to create a UUID for the robot. Not needed by all platforms; if not needed by your platform, return None.
arena_dims_from_criteria()	No	Get a list of the arena dimensions used in each generated experiment. Only needed if the dimensions are not specified on the cmdline, which can be useful if the batch criteria involves changing them; e.g., evaluating behavior with different arena shapes.
pre_exp_diagnostics()	No	Log any INFO-level diagnostics to stdout before a given *Experiment* is run. Useful to echo important execution environment configuration to the terminal as a sanity check.

Below is a sample/skeleton `plugin.py` to use as a starting point.

```
import typing as tp
import argparse

import implements

from sierra.core.experiment import bindings, xml, definition
from sierra.core.variables import batch_criteria as bc

@implements.implements(bindings.IParsedCmdlineConfigurer)
class CmdlineParserGenerator():
  def __call__() -> argparse.ArgumentParser:
      """A class that conforms to
      :class:`~sierra.core.experiment.bindings.ICmdlineParserGenerator`.
      """
      # As an example, assuming this platform can run on HPC
      # environments. Initialize all stages and return the initialized
      # parser to SIERRA for use.
      parser = hpc.HPCCmdline([-1, 1, 2, 3, 4, 5]).parser
      return cmd.PlatformCmdline(parents=[parser],
                                 stages=[-1, 1, 2, 3, 4, 5]).parser


@implements.implements(bindings.IParsedCmdlineConfigurer)
class ParsedCmdlineConfigurer():
    """A class that conforms to
    :class:`~sierra.core.experiment.bindings.IParsedCmdlineConfigurer`.
    """
```

(continues on next page)

(continued from previous page)

```
@implements.implements(bindings.IExpShellCmdsGenerator)
class ExpShellCmdsGenerator():
    """A class that conforms to
    :class:`~sierra.core.experiment.bindings.IExpShellCmdsGenerator`.
    """

@implements.implements(bindings.IExpRunShellCmdsGenerator)
class ExpRunShellCmdsGenerator():
    """A class that conforms to
    :class:`~sierra.core.experiment.bindings.IExpRunShellCmdsGenerator`.
    """

@implements.implements(bindings.IExecEnvChecker)
class ExecEnvChecker():
    """A class that conforms to
    :class:`~sierra.core.experiment.bindings.IExecEnvChecker`.
    """

@implements.implements(bindings.IExpConfigurer)
class ExpConfigurer():
    """A class that conforms to
    :class:`~sierra.core.experiment.bindings.IExpConfigurer`.
    """

@implements.implements(bindings.IExpRunConfigurer)
class ExpRunConfigurer():
    """A class that conforms to
    :class:`~sierra.core.experiment.bindings.IExpRunConfigurer`.
    """

def population_size_from_pickle(exp_def: tp.Union[xml.AttrChangeSet,
                                                  xml.TagAddList]) -> int:
    """
    Size can be obtained from added tags or changed attributes; platform
    specific.

    Arguments:

        exp_def: *Part* of the pickled experiment definition object.

  """

def population_size_from_def(exp_def: definition.XMLExpDef) -> int:
    """

    Arguments:

        exp_def: The *entire* experiment definition object.

  """
```

(continues on next page)

(continued from previous page)

```
def robot_prefix_extract(main_config: types.YAMLDict,
                         cmdopts: types.Cmdopts) -> str:
    """

    Arguments:

        main_config: Parsed dictionary of main YAML configuration.

        cmdopts: Dictionary of parsed command line options.
  """

def pre_exp_diagnostics(cmdopts: types.Cmdopts,
                        logger: logging.Logger) -> None:
    """
    Arguments:

        cmdopts: Dictionary of parsed command line options.

        logger: The logger to log to.

  """

def arena_dims_from_criteria(criteria: bc.BatchCriteria) -> tp.List[utils.ArenaExtent]:
    """
    Arguments:

       criteria: The batch criteria built from cmdline specification
    """
```

cmdline.py

Within this file you must define the `PlatformCmdline` class. A sample implementation is shown below to use as a starting bound. All member functions are optional.

```
import typing as tp
import argparse

from sierra.core import types
from sierra.core import config
import sierra.core.cmdline as cmd
import sierra.core.hpc as hpc

class PlatformCmdline(cmd.BaseCmdline):
    """
    Defines cmdline extensions to the core command line arguments
    defined in :class:`~sierra.core.cmdline.CoreCmdline` for the
    ``matrix`` platform. Any projects using this platform should
    derive from this class.

    Arguments:

        parents: A list of other parsers which are the parents of
                 this parser. This is used to inherit cmdline options
```

(continues on next page)

(continued from previous page)

```
                from the selected ``--exec-env`` at runtime. If
                None, then we are generating sphinx documentation
                from cmdline options.

        stages: A list of pipeline stages to add cmdline arguments
                for (1-5; -1 for multistage arguments). During
                normal operation, this will be [-1, 1, 2, 3, 4, 5].

    """

    def __init__(self,
                 parents: tp.Optional[tp.List[argparse.ArgumentParser]],
                 stages: tp.List[int]) -> None:

        # Normal operation when running sierra-cli
        if parents is not None:
            self.parser = argparse.ArgumentParser(prog='sierra-cli',
                                                  parents=parents,
                                                  allow_abbrev=False)
        else:
            # Optional--only needed for generating sphinx documentation
            self.parser = argparse.ArgumentParser(prog='sierra-cli',
                                                  allow_abbrev=False)

        # Initialize arguments according to configuration
        self.init_cli(stages)

    def init_cli(self, stages: tp.List[int]) -> None:
        if -1 in stages:
            self.init_multistage()

        if 1 in stages:
            self.init_stage1()

        # And so on...

    def init_stage1(self) -> None:
        # Experiment options
        experiment = self.parser.add_argument_group(
            'Stage1: Red pill or blue pill')

        experiment.add_argument("--pill-type",
                                choices=["red", "blue"],
                                help="""Red or blue""",
                                default="red")

    def init_multistage(self) -> None:
        neo = self.parser.add_argument_group('Neo Options')

        neo.add_argument("--using-powers",
                         help="""Do you believe you're the one or not?""",
                         action='store_true')
```

Within this file you must define the `PlatformCmdline` class. A sample implementation is shown below to use as a starting bound. All member functions are optional.

```
import typing as tp
import argparse

from sierra.core import types
from sierra.core import config
import sierra.core.cmdline as cmd
import sierra.core.hpc as hpc

class PlatformCmdline(cmd.BaseCmdline):
    """
    Defines cmdline extensions to the core command line arguments
    defined in :class:`~sierra.core.cmdline.CoreCmdline` for the
    ``matrix`` platform. Any projects using this platform should
    derive from this class.

    Arguments:

        parents: A list of other parsers which are the parents of
                 this parser. This is used to inherit cmdline options
                 from the selected ``--exec-env`` at runtime. If
                 None, then we are generating sphinx documentation
                 from cmdline options.

         stages: A list of pipeline stages to add cmdline arguments
                 for (1-5; -1 for multistage arguments). During
                 normal operation, this will be [-1, 1, 2, 3, 4, 5].

    """

     def __init__(self,
                  parents: tp.Optional[tp.List[argparse.ArgumentParser]],
                  stages: tp.List[int]) -> None:

         # Normal operation when running sierra-cli
         if parents is not None:
             self.parser = argparse.ArgumentParser(prog='sierra-cli',
                                                   parents=parents,
                                                   allow_abbrev=False)
         else:
             # Optional--only needed for generating sphinx documentation
             self.parser = argparse.ArgumentParser(prog='sierra-cli',
                                                   allow_abbrev=False)

         # Initialize arguments according to configuration
         self.init_cli(stages)

     def init_cli(self, stages: tp.List[int]) -> None:
         if -1 in stages:
             self.init_multistage()
```

(continues on next page)

(continued from previous page)

```
        if 1 in stages:
            self.init_stage1()

        # And so on...

    def init_stage1(self) -> None:
        # Experiment options
        experiment = self.parser.add_argument_group(
            'Stage1: Red pill or blue pill')

        experiment.add_argument("--pill-type",
                                choices=["red", "blue"],
                                help="""Red or blue""",
                                default="red")

    def init_multistage(self) -> None:
        neo = self.parser.add_argument_group('Neo Options')

        neo.add_argument("--using-powers",
                         help="""Do you believe you're the one or not?""",
                         action='store_true')
```

generators/platform_generators.py

Within this file you must define the `PlatformExpDefGenerator` and `PlatformExpRunDefGenerator` classes to generate XML changes common to all experiment runs for your platform and per-run changes, respectively.

```
from sierra.core.experiment import definition

class PlatformExpDefGenerator():
    """
    Create an experiment definition from the
    ``--template-input-file`` and generate XML changes to input files
    that are common to all experiments on the platform. All projects
    using this platform should derive from this class for `their`
    project-specific changes for the platform.

    Arguments:

        spec: The spec for the experimental run.
        controller: The controller used for the experiment, as passed
                    via ``--controller.
    cmdopts: Dictionary of parsed cmdline parameters.
    kwargs: Additional arguments.
    """

    def __init__(self,
                 spec: ExperimentSpec,
                 controller: str,
                 cmdopts: types.Cmdopts,
                 **kwargs) -> None:
        pass
```

(continues on next page)

(continued from previous page)

```
    def generate(self) -> definition.XMLExpDef:
        pass

class PlatformExpRunDefUniqueGenerator:
    """
    Generate XML changes unique to a experimental run within an
    experiment for the matrix platform.

    Arguments:

        run_num: The run # in the experiment.

        run_output_path: Path to run output directory within
                         experiment root (i.e., a leaf).

        launch_stem_path: Path to launch file in the input directory
                          for the experimental run, sans extension
                          or other modifications that the platform
                          can impose.

        random_seed: The random seed for the run.

        cmdopts: Dictionary containing parsed cmdline options.
    """
    def __init__(self,
                 run_num: int,
                 run_output_path: pathlib.Path,
                 launch_stem_path: pathlib.Path,
                 random_seed: int,
                 cmdopts: types.Cmdopts) -> None:
        pass
```

Within this file you must define the `PlatformExpDefGenerator` and `PlatformExpRunDefGenerator` classes to generate XML changes common to all experiment runs for your platform and per-run changes, respectively.

```
from sierra.core.experiment import definition

class PlatformExpDefGenerator():
    """
    Create an experiment definition from the
    ``--template-input-file`` and generate XML changes to input files
    that are common to all experiments on the platform. All projects
    using this platform should derive from this class for `their`
    project-specific changes for the platform.

    Arguments:

        spec: The spec for the experimental run.
        controller: The controller used for the experiment, as passed
                    via ``--controller.
    cmdopts: Dictionary of parsed cmdline parameters.
    kwargs: Additional arguments.
```

(continues on next page)

(continued from previous page)

```
    """

    def __init__(self,
                 spec: ExperimentSpec,
                 controller: str,
                 cmdopts: types.Cmdopts,
                 **kwargs) -> None:
        pass

    def generate(self) -> definition.XMLExpDef:
        pass

class PlatformExpRunDefUniqueGenerator:
    """
    Generate XML changes unique to a experimental run within an
    experiment for the matrix platform.

    Arguments:

        run_num: The run # in the experiment.

        run_output_path: Path to run output directory within
                         experiment root (i.e., a leaf).

        launch_stem_path: Path to launch file in the input directory
                          for the experimental run, sans extension
                          or other modifications that the platform
                          can impose.

        random_seed: The random seed for the run.

        cmdopts: Dictionary containing parsed cmdline options.
    """
    def __init__(self,
                 run_num: int,
                 run_output_path: pathlib.Path,
                 launch_stem_path: pathlib.Path,
                 random_seed: int,
                 cmdopts: types.Cmdopts) -> None:
        pass
```

8.13.2 Connect to SIERRA

1. Put `$HOME/git/plugins/platform/matrix` on your *SIERRA_PLUGIN_PATH* so that your platform can be selected via `--platform=platform.matrix`.

Note: Platform names have the same constraints as python package names (e.g., no dots).

8.14 Creating a New Execution Environment Plugin

For the purposes of this tutorial, I will assume you are creating a new HPC *Plugin* `HAL`, and the code for that plugin lives in `$HOME/git/plugins/hpc/HAL`.

If you are creating a new HPC plugin for an existing platform that comes with SIERRA (e.g., *ARGoS*) you have two options:

1. Following *Creating a New Platform Plugin* to create a new platform to add support for your execution environment within the existing platform.
2. Open a pull request for SIERRA with your created HPC plugin to get it into the main repo. This should be the preferred option, as most execution environment plugins have utility beyond whatever group initially wrote them.

In either case, the steps to actually create the code are below.

8.14.1 Create The Code

Create the following filesystem structure and content in `$HOME/git/plugins/hpc/HAL`. Each file is required; any number of additional files can be included.

`plugin.py`

Within this file, you may define the following classes, which must be named **EXACTLY** as specified, otherwise SIERRA will not detect them. If you omit a required class, you will get an error on SIERRA startup. If you try to use a part of SIERRA which requires an optional class you omitted, you will get a runtime error.

Table 5: Platform Plugin Classes

Class	Required?	Conforms to interface?
ParsedCmdlineConfigurer	No	*IParsedCmdlineConfigurer*
ExpRunShellCmdsGenerator	No	*IExpRunShellCmdsGenerator*
ExpShellCmdsGenerator	No	*IExpShellCmdsGenerator*
ExecEnvChecker	No	*IExecEnvChecker*

Below is a sample/skeleton `plugin.py` to use as a starting point.

```
from sierra.core.experiment import bindings

@implements.implements(bindings.IParsedCmdlineConfigurer)
class ParsedCmdlineConfigurer():
    """A class that conforms to
    :class:`~sierra.core.experiment.bindings.IParsedCmdlineConfigurer`.
    """

@implements.implements(bindings.IExpRunShellCmdsGenerator)
class ExpRunShellCmdsGenerator():
   """
   A class that conforms to
   :class:`sierra.core.experiment.bindings.IExpRunShellCmdsGenerator`.

   """

@implements.implements(bindings.IRunShellCmdsGenerator)
```

(continues on next page)

(continued from previous page)

```
class ExpShellCmdsGenerator():
   """
   A class that conforms to
   :class:`sierra.core.experiment.bindings.IExpShellCmdsGenerator`.

   """

@implements.implements(bindings.IExecEnvChecker)
class ExecEnvChecker():
    """A class that conforms to
    :class:`~sierra.core.experiment.bindings.IExecEnvChecker`.
 """
```

Within this file, you may define the following classes, which must be named **EXACTLY** as specified, otherwise SIERRA will not detect them. If you omit a required class, you will get an error on SIERRA startup. If you try to use a part of SIERRA which requires an optional class you omitted, you will get a runtime error.

Table 6: Platform Plugin Classes

Class	Required?	Conforms to interface?
ParsedCmdlineConfigurer	No	*IParsedCmdlineConfigurer*
ExpRunShellCmdsGenerator	No	*IExpRunShellCmdsGenerator*
ExpShellCmdsGenerator	No	*IExpShellCmdsGenerator*
ExecEnvChecker	No	*IExecEnvChecker*

Below is a sample/skeleton `plugin.py` to use as a starting point.

```
from sierra.core.experiment import bindings

@implements.implements(bindings.IParsedCmdlineConfigurer)
class ParsedCmdlineConfigurer():
    """A class that conforms to
    :class:`~sierra.core.experiment.bindings.IParsedCmdlineConfigurer`.
    """

@implements.implements(bindings.IExpRunShellCmdsGenerator)
class ExpRunShellCmdsGenerator():
   """
   A class that conforms to
   :class:`sierra.core.experiment.bindings.IExpRunShellCmdsGenerator`.

   """

@implements.implements(bindings.IRunShellCmdsGenerator)
class ExpShellCmdsGenerator():
   """
   A class that conforms to
   :class:`sierra.core.experiment.bindings.IExpShellCmdsGenerator`.

   """
```

(continues on next page)

(continued from previous page)

```
@implements.implements(bindings.IExecEnvChecker)
class ExecEnvChecker():
    """A class that conforms to
    :class:`~sierra.core.experiment.bindings.IExecEnvChecker`.
 """
```

8.14.2 Connect to SIERRA

1. Put `$HOME/git/plugins` on your *SIERRA_PLUGIN_PATH*. Then your plugin can be selected as `--exec-env=hpc.HAL`.

Note: Execution environment plugin names have the same constraints as python package names (e.g., no dots).

8.15 Creating a New Storage Plugin

For the purposes of this tutorial, I will assume you are creating a new storage *Plugin* `infinite`, and the code for that plugin lives in `$HOME/git/plugins/storage/infinite`.

8.15.1 Create the Code

1. Create the following filesystem structure in `$HOME/git/plugins/storage/infinite`:

   `plugin.py`

   Within this file, you must define the following classes, which must be named **EXACTLY** as specified, otherwise SIERRA will not detect them.

```
import pandas as pd
import pathlib

def df_read(path: pathlib.Path, **kwargs) -> pd.DataFrame:
    """
    Return a dataframe containing the contents of the CSV at the
    specified path. For other storage methods (e.g. database), you can
    use a function of the path way to uniquely identify the file in the
    database (for example).

    """


def df_write(df: pd.DataFrame, path: pathlib.Path, **kwargs) -> None:
    """
    Write a dataframe containing to the specified path. For other
    storage methods (e.g. database), you can use a function of the path
    way to uniquely identify the file in the database (for example) when
    you add it.

    """
```

Within this file, you must define the following classes, which must be named **EXACTLY** as specified, otherwise SIERRA will not detect them.

```
import pandas as pd
import pathlib

def df_read(path: pathlib.Path, **kwargs) -> pd.DataFrame:
    """
    Return a dataframe containing the contents of the CSV at the
    specified path. For other storage methods (e.g. database), you can
    use a function of the path way to uniquely identify the file in the
    database (for example).

    """


def df_write(df: pd.DataFrame, path: pathlib.Path, **kwargs) -> None:
    """
    Write a dataframe containing to the specified path. For other
    storage methods (e.g. database), you can use a function of the path
    way to uniquely identify the file in the database (for example) when
    you add it.

    """
```

8.15.2 Connect to SIERRA

1. Put `$HOME/git/plugins` on your *SIERRA_PLUGIN_PATH*. Then your plugin can be selected as `--exec-env=storage.infinite`.

Note: Storage plugin names have the same constraints as python package names (e.g., no dots).

CHAPTER

NINE

SIERRA PLUGINS

9.1 Platform Plugins

SIERRA supports a number of *platforms*, all of which can be used transparently for running experiments; well, transparent from SIERRA's point of view; you probably will still have to make code modifications to switch between platforms.

9.1.1 ARGoS Platform

This platform can be selected via `--platform=platform.argos`.

This is the default platform on which SIERRA will run experiments, and uses the *ARGoS* simulator. It cannot be used to run experiments on real robots.

Batch Criteria

See *Batch Criteria* for a thorough explanation of batch criteria, but the short version is that they are the core of SIERRA–how to get it to DO stuff for you. The following batch criteria are defined which can be used with any *Project*.

- *Population Size*
- *Constant Population Density*
- *Variable Population Density*

Population Size

Changing the population size to investigate behavior across scales within a static arena size (i.e., variable density). This criteria is functionally identical to *Variable Population Density* in terms of changes to the template XML file, but has a different semantic meaning which can make generated deliverables more immediately understandable, depending on the context of what is being investigated (e.g., population size vs. population density on the X axis).

Cmdline Syntax

`population_size.{model}{N}[.C{cardinality}]`

- `model` - The population size model to use.
  - `Log` - Population sizes for each experiment are distributed 1...N by powers of 2.
  - `Linear` - Population sizes for each experiment are distributed linearly between 1...N, split evenly into 10 different sizes.
- `N` - The maximum population size.
- `cardinality` - If the model is `Linear`, then this can be used to specify how many experiments to generate; i.e, it defines the *size* of the linear increment. Defaults to 10 if omitted.

Examples

- `population_size.Log1024`: Static population sizes 1...1024
- `population_size.Linear1000`: Static population sizes 100...1000 (10)
- `population_size.Linear3.C3`: Static population sizes 1...3 (3)
- `population_size.Linear10.C2`: Static population sizes 5...10 (2)

Constant Population Density

Changing the population size and arena size together to maintain the same population size/arena size ratio to investigate behavior across scales.

Note: This criteria is for *constant* density of robots as population sizes increase. For *variable* robot density, use *Population Size* or *Variable Population Density*.

Cmdline Syntax

`population_constant_density.{density}.I{Arena Size Increment}.C{cardinality}`

- `density` - <integer>p<integer> (i.e. 5p0 for 5.0)
- `Arena Size Increment` - Size in meters that the X and Y dimensions should increase by in between experiments. Larger values here will result in larger arenas and more robots being simulated at a given density. Must be an integer.
- `cardinality` How many experiments should be generated?

Examples

- `population_constant_density.1p0.I16.C4`: Constant density of 1.0. Arena dimensions will increase by 16 in both X and Y for each experiment in the batch (4 total).

Variable Population Density

Changing the population size to investigate behavior across scales within a static arena size. This criteria is functionally identical to *Population Size* in terms of changes to the template XML file, but has a different semantic meaning which can make generated deliverables more immediately understandable, depending on the context of what is being investigated (e.g., population density vs. population size on the X axis).

Note: This criteria is for *variable* density of robots as population sizes increase. For *constant* robot density, use *Constant Population Density*.

Cmdline Syntax

`population_variable_density.{density_min}.{density_max}.C{cardinality}`

- `density_min` - <integer>p<integer> (i.e. 5p0 for 5.0)
- `density_max` - <integer>p<integer> (i.e. 5p0 for 5.0)
- `cardinality` How many experiments should be generated? Densities for each experiment will be linearly spaced between the min and max densities.

Examples

- `population_variable_density.1p0.4p0.C4`: Densities of 1.0,2.0,3.0,4.0.

Environment Variables

This platform respects *SIERRA_ARCH*.

Random Seeding For Reproducibility

ARGoS provides its own random seed mechanism under `<experiment>` which SIERRA uses to seed each experiment. *Project* code should use this mechanism or a similar random seed generator manager seeded by the same value so that experiments can be reproduced exactly. By default SIERRA does not overwrite its generated random seeds for each experiment once generated; you can override with `--no-preserve-seeds`. See *Template Input Files* and *Experimental Definition Requirements* for details on the format of the provided seed.

9.1.2 ROS1+Gazebo Platform

This platform can be selected via `--platform=platform.ros1gazebo`.

This is the platform on which SIERRA will run experiments using the *Gazebo* simulator and *ROS1*. It cannot be used to run experiments on real robots. To use this platform, you must setup the *SIERRA ROSBridge*.

Worlds within ROS1+Gazebo are infinite from the perspective of physics engines, even though a finite area shows up in rendering. So, to place robots randomly in the arena at the start of simulation across *Experimental Runs* (if you want to do that) "dimensions" for a given world must be specified as part of the `--scenario` argument. If you don't specify dimensions as part of the `--scenario` argument, then you need to supply a list of valid robot positions via `--robot-positions` which SIERRA will choose from randomly for each robot.

Batch Criteria

See *Batch Criteria* for a thorough explanation of batch criteria, but the short version is that they are the core of SIERRA–how to get it to DO stuff for you. The following batch criteria are defined which can be used with any *Project*.

- *System Population Size*

System Population Size

Changing the system size to investigate behavior across scales within a static arena size (i.e., variable density). Systems are homogeneous.

Cmdline Syntax

`population_size.{model}{N}[.C{cardinality}]`

- `model` - The population size model to use.
  - `Log` - Population sizes for each experiment are distributed 1...N by powers of 2.
  - `Linear` - Population sizes for each experiment are distributed linearly between 1...N, split evenly into 10 different sizes.
- `N` - The maximum population size.
- `cardinality` - If the model is `Linear`, then this can be used to specify how many experiments to generate; i.e, it defines the *size* of the linear increment. Defaults to 10 if omitted.

Examples

- `population_size.Log1024`: Static population sizes 1...1024
- `population_size.Linear1000`: Static population sizes 100...1000 (10)
- `population_size.Linear3.C3`: Static population sizes 1...3 (3)
- `population_size.Linear10.C2`: Static population sizes 5...10 (2)

Environment Variables

This platform ignores *SIERRA_ARCH*.

Random Seeding For Reproducibility

ROS1+Gazebo do not provide a random number generator manager, but SIERRA provides random seeds to each *Experimental Run* which *Project* code should use to manage random number generation, if needed, to maximize reproducability. See *Template Input Files* and *Experimental Definition Requirements* for details on the format of the provided seed. By default SIERRA does not overwrite its generated random seeds for each experiment once generated; you can override with `--no-preserve-seeds`.

9.1.3 ROS1+Robot Platform

This platform can be selected via `--platform=platform.ros1robot`.

This is the platform on which SIERRA will run experiments using *ROS1* on a real robot of your choice. To use this platform, you must setup the *SIERRA ROSBridge*. This is a generic platform meant to work with most real robots which *ROS1* supports, and as a starting point to derive more specific platform configuration for a given robot (if needed). For all execution environments using this platform (see *Real Robot Execution Environment Plugins* for examples), SIERRA will run experiments spread across multiple robots using GNU parallel.

SIERRA designates the host machine as the ROS master, and allows you to (optionally) specify configuration for running one or more nodes on it in the `--template-input-file` to gather data from robots (see below). This is helpful in some situations (e.g., simple robots which can't manage network mounted filesystems).

Batch Criteria

See *Batch Criteria* for a thorough explanation of batch criteria, but the short version is that they are the core of SIERRA–how to get it to DO stuff for you. The following batch criteria are defined which can be used with any *Project*.

- *System Population Size*

System Population Size

Changing the system size to investigate behavior across scales within a static arena size (i.e., variable density). Systems are homogeneous.

`population_size.{model}{N}[.C{cardinality}]`

- `model` - The population size model to use.
  - `Log` - Population sizes for each experiment are distributed 1...N by powers of 2.
  - `Linear` - Population sizes for each experiment are distributed linearly between 1...N, split evenly into 10 different sizes.
- `N` - The maximum population size.
- `cardinality` - If the model is `Linear`, then this can be used to specify how many experiments to generate; i.e, it defines the *size* of the linear increment. Defaults to 10 if omitted.

Examples

- `population_size.Log1024`: Static population sizes 1...1024
- `population_size.Linear1000`: Static population sizes 100...1000 (10)
- `population_size.Linear3.C3`: Static population sizes 1...3 (3)
- `population_size.Linear10.C2`: Static population sizes 5...10 (2)

Environment Variables

This platform ignores *SIERRA_ARCH*.

Random Seeding For Reproducibility

ROS do not provide a random number generator manager, but SIERRA provides random seeds to each *Experimental Run* which *Project* code should use to manage random number generation, if needed, to maximize reproducability. See *Template Input Files* and *Experimental Definition Requirements* for details on the format of the provided seed. By default SIERRA does not overwrite its generated random seeds for each experiment once generated; you can override with `--no-preserve-seeds`.

Real Robot Considerations

SIERRA makes the following assumptions about the robots it is allocated each invocation:

- No robots will die/run out of battery during an *Experimental Run*.
- Password-less ssh is setup to each robot SIERRA is handed to use (can be as a different user than the one which is invoking SIERRA on the host machine).
- The robots have static IP addresses, or are always allocated an IP from a known set so you can pass the set of IPs to SIERRA to use. This set of IP address/hostnames can be explicitly passed to SIERRA via cmdline (see *Command Line Interface*) or implicitly passed via *SIERRA_NODEFILE*.
- The ROS environment is setup either in the .bashrc for the robot login user, or the necessary bits are in a script which SIERRA sources on login to each robot (this is a configuration parameter–see *Main Configuration*).
- ROS does not provide a way to say "Run this experiment for X seconds", so SIERRA inserts its own timekeeper node into each robot which will exit after X seconds and take the roslaunch process with it on each robot and/or the master node.

See also *ROS1+Robot Platform*.

9.2 Execution Environment Plugins

9.2.1 HPC Execution Environment Plugins

SIERRA is capable of adapting its runtime infrastructure to a number of different HPC environments so that experiments can be run efficiently on whatever computational resources a researcher has access to. Supported environments that come with SIERRA are listed on this page.

These plugins tested with the following platforms (they may work on other platforms out of the box too):

- *ARGoS Platform*

- *ROS1+Gazebo Platform*

SIERRA makes the following assumptions about the HPC environments corresponding to the plugins listed on this page:

Table 1: HPC Environment Assumptions

Assumption	Rationale
All nodes allocated to SIERRA have the same # of cores (can be less than the total # available on each compute node). Note that this may be *less* than the actual number of cores available on each node, if the HPC environment allows node sharing, and the job SIERRA runs in is allocated less than the total # cores on a given node.	Simplicity: If allocated nodes had different core counts, SIERRA would have to do more of the work of an HPC scheduler, and match jobs to nodes. May be an avenue for future improvement.
All nodes have a shared filesystem.	Standard feature on HPC environments. If for some reason this is not true, stage 2 outputs will have to be manually placed such that it is as if everything ran on a common filesystem prior to running any later stages.

Local HPC Plugin

This HPC environment can be selected via `--exec-env=hpc.local`.

This is the default HPC environment in which SIERRA will run all experiments on the same computer from which it was launched using GNU parallel. The # simultaneous simulations will be determined by:

```
# cores on machine / # threads per experimental run
```

If more simulations are requested than can be run in parallel, SIERRA will start additional simulations as currently running simulations finish.

No additional configuration/environment variables are needed with this HPC environment for use with SIERRA.

ARGoS Considerations

The # threads per *experimental run* is defined with `--physics-n-engines`, and that option is required for this HPC environment during stage 1.

PBS HPC Plugin

This HPC environment can be selected via `--exec-env=hpc.pbs`.

In this HPC environment, SIERRA will run experiments spread across multiple allocated nodes by a PBS compatible scheduler such as Moab. The following table describes the PBS-SIERRA interface. Some PBS environment variables are used by SIERRA to configure experiments during stage 1,2 (see TOQUE-PBS docs for meaning); if they are not defined SIERRA will throw an error.

Table 2: PBS-SIERRA interface

PBS environment variable	SIERRA context
PBS_NUM_PPN	Used to calculate # threads per experimental run for each allocated compute node via: `floor(PBS_NUM_PPN / --exec-jobs-per-node)` That is, `--exec-jobs-per-node` is required for PBS HPC environments.
PBS_NODEFILE	Obtaining the list of nodes allocated to a job which SIERRA can direct GNU parallel to use for experiments.
PBS_JOBID	Creating the UUID nodelist file passed to GNU parallel, guaranteeing no collisions (i.e., simultaneous SIERRA invocations sharing allocated nodes) if multiple jobs are started from the same directory.

The following environmental variables are used in the PBS HPC environment:

Environment variable	Use
SIERRA_ARCH	Used to enable architecture/OS specific builds of simulators for maximum speed at runtime on clusters.
PARALLEL	Used to transfer environment variables into the GNU parallel environment. This must be always done because PBS doesn't transfer variables automatically, and because GNU parallel starts another level of child shells.

SLURM HPC Plugin

https://slurm.schedmd.com/documentation.html

This HPC environment can be selected via `--exec-env=hpc.slurm`.

In this HPC environment, SIERRA will run experiments spread across multiple allocated nodes by the SLURM scheduler. The following table describes the SLURM-SIERRA interface. Some SLURM environment variables are used by SIERRA to configure experiments during stage 1,2 (see SLURM docs for meaning); if they are not defined SIERRA will throw an error.

Table 3: SLURM-SIERRA interface

SLURM environment variable	SIERRA context	Command line override
SLURM_CPUS_PER_TASK	Used to set # threads per experimental node for each allocated compute node.	N/A
SLURM_TASKS_PER_NODE	Used to set # parallel jobs per allocated compute node.	`--exec-jobs-per-node`
SLURM_JOB_NODELIST	Obtaining the list of nodes allocated to a job which SIERRA can direct GNU parallel to use for experiments.	N/A
SLURM_JOB_ID	Creating the UUID nodelist file passed to GNU parallel, guaranteeing no collisions (i.e., simultaneous SIERRA invocations sharing allocated nodes if multiple jobs are started from the same directory).	N/A

The following environmental variables are used in the SLURM HPC environment:

Environment variable	Use
SIERRA_ARCH	Used to enable architecture/OS specific builds of simulators for maximum speed at runtime on clusters.
PARALLEL	Used to transfer environment variables into the GNU parallel environment. This must be done even though SLURM can transfer variables automatically, because GNU parallel starts another level of child shells.

Adhoc HPC Plugin

This HPC environment can be selected via `--exec-env=hpc.adhoc`.

In this HPC environment, SIERRA will run experiments spread across an ad-hoc network of compute nodes. SIERRA makes the following assumptions about the compute nodes it is allocated each invocation:

- All nodes have a shared filesystem.

The following environmental variables are used in the Adhoc HPC environment:

Environment variable	SIERRA context	Command line override	Notes
SIERRA_NODEFILE	Contains hostnames/IP address of all compute nodes SIERRA can use. Same format as GNU parallel `--sshloginfile`.	`--nodefile`	*SIERRA_NODEFILE* must be defined or `--nodefile` passed. If neither is true, SIERRA will throw an error.

9.2.2 Real Robot Execution Environment Plugins

SIERRA is capable of adapting its runtime infrastructure to a number of different robots. Supported environments that come with SIERRA are listed on this page.

These plugins are tested with the following platforms (they may work with other platforms out of the box too):

- *ROS1+Robot Platform*

Turtlebot3

This real robot plugin can be selected via `--exec-env=robot.turtlebot3`.

In this execution environment, SIERRA will run experiments spread across multiple turtlebots using GNU parallel.

The following environmental variables are used in the turtlebot3 environment:

Environment variable	SIERRA context	Command line override	Notes
SIERRA_NODEFILE	Contains hostnames/IP address of all robots SIERRA can use. Same format as GNU parallel `--sshloginfile`.	`--nodefile`	*SIERRA_NODEFILE* must be defined or `--nodefile` passed. If neither is true, SIERRA will throw an error.

9.3 Storage Plugins

SIERRA is capable of reading *Experimental Run* output data in a number of formats. Supported formats that come with SIERRA are:

- *CSV*

9.3.1 CSV

This storage plugin can be selected via `--storage-medium=storage.csv`.

This is the default storage type which SIERRA will use to read *Experimental Run* outputs.

Important: The CSV files read by this plugin must be semicolon (`;`) delimited, *NOT* comma delimited; this may changed in a future version of SIERRA.

CHAPTER

TEN

SIERRA INSTALLATION REFERENCE

10.1 SIERRA PyPi Package

The SIERRA PyPi package provides the following executables:

- `sierra-cli` - The command line interface to SIERRA.

The SIERRA PyPi package provides the following man pages, which can be viewed via `man <name>`. Man pages are meant as a *reference*, and though I've tried to make them as full featured as possible, there are some aspects of SIERRA which are only documented on the online docs (e.g., the tutorials). Available manpages are:

- `sierra-cli` - The SIERRA command line interface.
- `sierra-usage` - How to use SIERRA (everything BUT the command line interface).
- `sierra-platforms` - The target platforms that SIERRA currently supports (e.g., ARGoS).
- `sierra-examples` - Examples of SIERRA usage via command line invocations demonstrating various features.
- `sierra-glossary` - Glossary of SIERRA terminology to make things easier to understand.
- `sierra-exec-envs` - The execution environments that SIERRA currently supports (e.g., SLURM).

10.1.1 Installing SIERRA

```
pip3 install sierra-research
```

10.2 SIERRA ROSBridge

SIERRA provides a *ROS1* package containing functionality it uses to manage simulations and provide run-time support to *projects* using a *Platform* built on ROS. To use SIERRA with a ROS platform, you need to setup the SIERRA ROSbridge package here (details in README): https://github.com/jharwell/sierra_rosbridge.git.

This package provides the following nodes:

- `sierra_timekeeper` - Tracks time on an *Experimental Run*, and terminates once the amount of time specified in `--exp-setup` has elapsed. Necessary because ROS does not provide a way to say "Run for this long and then terminate". An XML tag including this node is inserted by SIERRA into each `.launch` file.

CHAPTER

ELEVEN

SIERRA DESIGN PHILOSOPHY

This document outlines the main philosophy behind SIERRA's design, how it can be used, and so forth. It really boils down to a few core ideas.

11.1 Single Input, Multiple Output

During stage 1, SIERRA generates multiple experiments via multiple experimental run input files all from a single template input file, which must be specified on the command line. SIERRA does not follow any references/links to other XML files in this template input file, greatly simplifying the generation process and improving reproducability of experiments (i.e., less cryptic/subtle errors because of a ROS `<include>` which is different between the package versions installed on one system and another).

11.2 Assert Often, Fail Early

If a condition arises which SIERRA can't easily handle, abort, either via an uncaught exception or an `assert()`. Don't try to recover by throwing an exception which can be caught at a higher level, etc., just abort. This gives users confidence that *if* SIERRA doesn't crash, then it is probably working properly. As a result of this, any `try-catch` blocks which do exist should always be in the same function; never rely on raised exceptions to be caught at higher levels.

11.3 Never Delete Things

Because SIERRA is intended for academic research, and experimental data can be hard fought, SIERRA tries to guard against accidental deletions/overwrites of said data that users actually want to keep, but forgot to change parameters to direct SIERRA to keep it. Therefore, we force the user to explicitly say that deleting/overwriting is OK when it might cause irreparable damage to experiment results (i.e., only pipeline stages 1 and 2). Overwriting is OK in later pipeline stages since those files are built from stage 1 and 2 files, and can be easily regenerated if overwritten.

11.4 Better Too Much Configuration Than Too Little

SIERRA is designed to be as modular and extensible as possible (just like ARGoS, ROS, Gazebo, etc.), so that it can be adapted for a wide variety of applications. When in doubt, SIERRA exposes relevant settings as configuration (even if the average user will never change them from their default values).

11.5 Swiss Army Pipeline

SIERRAS 5-stage *Pipeline* is meant to be like a Swiss army knife, in that you can run (mostly) arbitrary subsets of the 5 stages and have everything work as you would expect it to, to help with tweaking experimental design, generated graphs, etc., with minimal headaches of "Grrr I have to wait for THAT part to run again before it will get to re-run the part I just changed the configuration for".

CHAPTER

TWELVE

FAQ

1. Q: I'm really confused by all the terminology that SIERRA uses–how can I better understand the documentation?

   A: Look at the *Glossary* for all of the terms which SIERRA defines specific names for.

2. Q: Do I need to re-run my experiments if I want to tweak a particular generated graph ?

   A: No! Experiment execution happens in stage 2, and graph generation happens in stage4. If you just want to add/remove lines from a graph, change the title, formatting, etc., you can just tell SIERRA to re-run stage 4 via `--pipeline 4`. See *SIERRA Pipeline* for more details about pipeline stages.

3. Q: How to I resume an experiment which got killed part of the way through by an HPC scheduler for exceeding its job time limit ?

   A: Run SIERRA just as you did before, but add `--exec-resume`, which will tell SIERRA to pick up where it left off. See *Command Line Interface* for the full cmdline reference.

4. Q: How do I run a non-default set of pipeline stages, such as {3,4}?

   A: `sierra-cli ... --pipeline 3 4`

   Important: If you run something other than `--pipeline 1 2 3 4`, then before stage X will run without crashing, you (probably) need to run stage X-1. This is a logical limitation, because the different pipeline stages build on each other.

   ---

5. Q: How do I prevent SIERRA from stripping out ARGoS XML tags for sensors/actuators?

   A: There are 3 options for this: `--with-robot-leds`, `--with-robot-rab`, and `--with-robot-battery`. More may be added in the future if needed.

6. Q: SIERRA crashed–why?

   A: The most likely cause of the crash is that stage X-1 of the pipeline did not successfully complete before you ran stage X. Pipeline stages build on each other, so previously stages need to complete before later stages can run.

7. Q: SIERRA hangs during stage {3,4}–why?

   A: The most likely cause is that not all runs generated outputs which resulted in CSV files of the same shape when read into SIERRA. E.g., for CSV file outputs, not all CSV files with the same name in the output directory for each run had the same number of rows and columns. SIERRA does not sanitize run outputs before processing them, and relies CSV files of the same shape when processing results for generating statistics. Depending on the nature of the inconsistency, you may see a crash, or you may see it hang indefinitely as it waits for a sub-process which crashed to finish.

8. Q: SIERRA fails to run on my HPC environment?

A: The most likely reason is that you don't have the necessary environment variables set up–see *HPC Execution Environment Plugins* for details on what is required.

9. Q: SIERRA doesn't generate any graphs during stage4/the graph I'm interested is not there.

   A: SIERRA matches the stem of an output CSV file with the stem in a `.yaml` configuration file; if these are not the same, then no graph will be generated. You can run SIERRA with `--log-level=TRACE` to during stage 4 to see which graphs it is generating, and which it is not because the necessary source CSV file does not exist. This is usually enough to identify the culprit.

10. Q: Stage 3 takes a very long time–why?

    A: SIERRA is conservative, and attempts to verify all the results of all runs within each experiment it processes during stage 3, which can be VERY computationally intensive, depending on how many outputs are generated for each runs. During early experimental runs, this can be helpful to identify which runs crashed/did not finish/etc as a result of errors in the project C++ library. For later runs, or runs in which you are generating outputs for rendering, this is unnecessary, and can be disabled with `--df-skip-verify`.

11. Q: SIERRA does not overwrite the input configuration for my experiment/SIERRA won't run my experiments again after they run the first time–why?

    A: This is by design: SIERRA *never ever never* deletes stuff for you in stages {1,2} that can result in lost experimental results in later stages. The files generated in stages {3,4,5} are generated *from* the earlier results, so it is OK to overwrite those. However, if you are sure you want to overwrite stuff, you can pass `--exp-overwrite` to disable this behavior during stages {1,2}. See also *SIERRA Design Philosophy*.

12. Q: I need to apply very precise/nuanced configuration to experiments that is too specific to be easily captured in a *Batch Criteria* or other variable. How can I do this?

    A: You could create one or more controller categories/controllers in `controllers.yaml`. Within each category *AND* controller, you can specify arbitrary changes to the `--template-input-file`: adding tags, removing tags, modifying attributes. This is a good way to apply tricky configuration which doesn't really fit anywhere else, or to try out some "quick and dirty" changes to see if they do what you want before codifying them with a python class (see *Main Configuration* for details on how to do that).

13. Q: SIERRA can't find a module it should be able to find via *SIERRA_PLUGIN_PATH* or *PYTHONPATH*. I know the module path is correct–why can't SIERRA find it?

    A: If you're sure you have the two environment variables set correctly, the reason is likely that you have an import *inside* the module you are trying to load which is not found. Try this:

    ```
    python3 -m full.path.to.module
    ```

    This command will attempt to find and load the problematic module, and will print any import errors. When you load modules dynamically in python, those errors don't get printed, python just says "can't find the module" instead of "found the module but I can't load it because of bad imports".

14. Q: I have multiple projects which all share batch criteria/generators/etc. How can I share this between projects?

    A: You have a couple options, depending on your preferences and the nature of what you want to share:

    - You could create a "common" project containing the reusable classes, and your other projects inherit from these classes as needed. This works if most of the stuff you want to share is class-based and does *not* need to be selectable via `--batch-criteria`.

      Pros: Easy, straightforward.

      Cons: Being able to import stuff from a project which was not passed via `--project` is subject to *SIERRA_PLUGIN_PATH*, which might make sharing classes trickier, because you will have to make sure the right version of a class is found by SIERRA (you can have it tell you via `--log-level=TRACE`).

- You can put common stuff into a separate python module/package/repo, and import it into your SIERRA project via *PYTHONPATH*. This works if most of the stuff you want to share does *not* need to be selectable via `--batch-criteria`.

  Pros: Clearer separation between shared and non-shared code.

  Cons: Debugging is more difficult because you now have multiple environment variables which need to be set in order to be able to run SIERRA.

- You can put shared stuff into a common project, and then "lift" these classes declarations into your projects SIERRA import path as needed. For example, suppose you have a project `laserblast` on *SIERRA_PLUGIN_PATH* as `$HOME/git/sierra-projects/laser_blast` (i.e., *SIERRA_PLUGIN_PATH*=`$HOME/git/sierra-projects`), which relies on some shared code in `$HOME/git/sierra-projects/common`. Specifically a `SimpleBlaster` class in `common/variables/simple_blaster.py` which you want to be selectable via `--batch-criteria=simple_blaster.XX.YY.ZZ` in the `laser_blast` project. You can create the following `__init__.py` file in `laser_blast/variables/__init__.py`:

```
import sys
from common.variables simple_blaster

sys.modules['laser_blast.variables.simple_blaster'] = simple_blaster
```

  Then, when SIERRA asks the python interpreter to find `laser_blast.variables.simple_blaster`, it is as if you had defined this class in the `laser_blast.variables` namespace.

  This works well when the stuff you want to share between projects *does* need to be selectable via `--batch-criteria`.

  Pros: Good code reuse, no danger of selecting the wrong version of a class.

  Cons: Sort of hacky from a python interpreter/language point of view.

CHAPTER

THIRTEEN

CONTRIBUTING

13.1 Types of contributions

All types of contributions are welcome: bugfixes, adding unit/integration tests, etc. If you're only have a little bit of time and/or are new to SIERRA, looking at the issues is a good place to look for places to contribute. If you have more time and/or want to give back to the community in a bigger way, see *Development Roadmap* for some big picture ideas about where things might be going, and help shape the future!

13.2 Mechanics

13.2.1 Writing the code

To contribute to the SIERRA core, in you should follow the general workflow and python development guide outlined in Python Development Guide and ln-libra-dev-workflow. For the static analysis step:

1. Install development packages for SIERRA (from the SIERRA repo root):

   ```
   pip3 install .[devel]
   ```

2. Run `nox` to check most things prior to commiting/pushing your changes. If there are errors *you* have introduced, fix them. Some checkers (such as pylint), will still report errors, as cleaning up the code is always a work in progress:

   ```
   nox
   ```

3. Run the following on any module directories you changed, from the root of SIERRA:

   ```
   mypy --ignore-missing-imports --warn-unreachable sierra
   ```

   Fix relevant errors your changes have introduced (there will probably still be errors in the my output, because cleaning up the code is always a work in progress), and mypy just gives a lot of false positives in general; this is also why it's not part of what runs with `nox`.

13.2.2 SIERRA Source Code Directory Structure

It is helpful to know how SIERRA is layed out, so it is easier to see how things fit together, and where to look for implementation details if (really *when*) SIERRA crashes. So here is the directory structure, as seen from the root of the repository.

- `sierra` - The SIERRA python package
  - `core/` - The parts of SIERRA which independent of the *Project* being run.
    - `experiment/` - Various interfaces and bindings for use by plugins.
    - `generators/` - Generic controller and scenario generators used to modify template XML files to provide the setting/context for running experiments with variables.
    - `graphs/` - Generic code to generate graphs of different types.
    - `models/` - Model interfaces.
    - `pipeline/` - Core pipline code in 5 stages (see *SIERRA Pipeline*).
    - `ros1/` - Common *ROS1* bindings.
    - `variables/` - Generic generators for experimental variables to modify template XML files in order to run experiments with a given controller.
  - `plugins/` - Plugins which provide broad customization of SIERRA, and enables it to adapt to a wide variety of platforms and experiment outputs.
- `docs/` - Contains sphinx source code to generate these shiny docs.

CHAPTER

FOURTEEN

DEVELOPMENT ROADMAP

This page shows a brief overview of some of the ways I think SIERRA could be improved. Big picture stuff, not "add more unit tests".

14.1 Supporting ROS2

This would require adding several new platform plugin (one for each simulator that SIERRA supports that supports ROS1, and a new ROS2 real robot plugin). Since ROS2 still supports XML, this should actually be a fairly self-contained improvement.

14.2 Supporting WeBots

This would require adding a new platform plugin. Should be a fairly self-contained improvement.

14.3 Supporting NetLogo

This would require adding a new platform plugin. I *think* netlogo can take in XML, so this should be a fairly self-contained improvement. netlogo handles parallel experimental runs, so that might require some additional configuration, since none of the currently supported platforms do that.

Adding this would also make SIERRA more appealing/using to researchers outside of robotics.

14.4 Supporting multiple types of experiment input files (not just XML)

This is a fairly involved change, as it affects the SIERRA core. One way I *cannot* do this is to define yet another plain text format for template inputs files and say "to use SIERRA you have to use this format". That makes it easier for me as a developer, but will turn off 90% of people who want a nice plug and play approach to automating their research.

There are a couple of ways of doing this which might work:

14.4.1 Option 1

Finding an AST tool or stitching a few together to make a toolchain which can parse to and from arbitrary plain text formats and replace the XML experimental definition class. All batch criteria and variables would have to be rewritten to express their changes, additions, or removals from the template input file in terms of operations on this AST. Pandoc might be a good starting point, but it does not support XML out of the box. You would have to do XML -> html with xlst, and then do html -> whatever with pandoc. No idea if this would be a viable toolchain which could support pretty much any simulator/platform.

Pros: Makes it easier for SIERRA to support any plain text input format the AST tool supports.

Cons: Expressing things in terms of operations on an AST instead of changes to a textual input file is likely to be more confusing and difficult for users as they create new variables. This change would not be backwards compatible.

14.4.2 Option 2

Keep using XML as the language for the template input files that SIERRA processes. To support other formats needed by a given simulator or platform, define a translation layer/engine which takes in XML and outputs the desired format, or reads it in and transforms it to XML for SIERRA to manipulate.

Pros: Keeps the SIERRA core mostly the same. Reduces the type restrictions on template input files to "plain text" rather than strictly XML. Would be backwards compatible.

Cons: Difficult to translate to/from XML/other plaintext formats. xlst can output plain text from XML, or read plain text and convert it to XML, but you need to define a schema for how to do that, which may be very non-trivial. This would make the process of defining platform plugins for non-XML inputs/outputs much trickier. From SIERRA's perspective, the changes would be relatively minor.

14.4.3 Option 3

Reworking the SIERRA core to support a set of types of experimental definitions which would be implemented as plugins: you would have an XMLExpDef, PythonExpDef, etc. The type of experiment definition and therefore the type of the template input file could be inferred from the contents of `--template-input-file`, or specified on the cmdline.

Pros: SIERRA core would remain relatively easier to understand, and this paradigm is in line with SIERRA's plugin philosophy. Would be backwards compatible.

Cons: You would need multiple versions of a batch criteria or variable if you wanted that particular criteria to work across platforms which did not all support XML. You could get around this with a base class for a given batch criteria/variable which implemented everything but the actual generation of changes/additions/removals to the experimental definition, and just have adapter classes for each type of experimental definition you wanted to be able to use that batch criteria with. However, there will probably still be cases which would result in lots of code duplication.

CHAPTER

FIFTEEN

GLOSSARY

SIERRA uses a number of common research-related terms, perhaps in different ways than you are used to. They are defined here to help demystify *why* SIERRA works/is designed the way it is, and to help you find your way around.

ARGoS A state-of-the-art multi-physics engine robotics simulator which SIERRA supports as a *Platform*. The ARGoS website is at https://www.argos-sim.info/index.php.

Gazebo A state-of-the-art robotics simulator with many features including .urdf robot models, realistic 3D rendering, and more. The Gazebo website is at http://gazebosim.org.

ROS1 You know it. You either love it or hate it, but you can't escape it. The ROS website is at https://wiki.ros.org.

ROS1+Gazebo A *Platform* supported by SIERRA to use *ROS1* with the *Gazebo* simulator.

ROS1+Robot A *Platform* supported by SIERRA using *ROS1* and any robot which supports ROS.

Project The SIERRA plugin allowing you to use your code on the platform of your choice within the SIERRA framework. A project is mainly a set of `.yaml` configuration files, project-specific *Batch Criteria*, and some utility classes which allow SIERRA to translate the `--scenario` argument passed on the cmdline to sets of XML changes as part of *Experiment* definition.

Important: The parent directories of all SIERRA project plugins must be on *SIERRA_PLUGIN_PATH*, or SIERRA won't be able to find them.

Important: You can't share projects across *Platforms*, so if you want to be able to use your project on ROS and ARGoS (for example), you will need to create 2 separate projects with shared python code imported into each as needed.

Specified via `--project` on the cmdline. See *Command Line Interface* for documentation.

Tick A timestep. In SIERRA, physics engines will perform some number of iterations per tick, and after that the controllers for all agents will be run.

Batch Criteria A *variable* you wish to use with SIERRA to measure its effect on system behavior. A batch criteria can has a single axis (such as *Population Size*), in which case it is called *univariate*, or have two axes (such as *Population Size* and another batch criteria such as one defining sensor and actuator noise to apply to the robots), in which case it is called *bivariate*.

Univariate batch criteria have one dimension, and so the graphs produced by them are (usually) linegraphs with a numerical representation of the range for the variable on the X axis, and some other quantity of interest on the Y.

Bivariate batch criteria have two dimensions, and so the graphs produced by them are (usually) heatmaps with the first variable in the criteria on the X axis, the second on the Y, and the quantity of interest on the Z. Bivariate

batch criteria are always built using univariate batch criteria as building blocks.

The axis/axes you select as part of the batch criteria you will use to define for *Batch Experiment* define the *search space* for your experiment: either a 1D or 2D array of *Experiments*.

Batch criteria define a *range* of sets changes for one or more elements in a template `.xml` file (passed to SIERRA with `--template-input-file`). For each element in the range, the changes are applied to the template `.xml` file to define *Experiments*. The set of defined experiments is called a *Batch Experiment*.

The batch criteria you can use depends on:

- The *Project* you have loaded, as each project can define their own batch criteria (see *Create A New Batch Criteria*).
- The *Platform* you have selected, as each platform defines some basic batch criteria that any project/experiment can use.

SIERRA itself does not define any batch criteria.

Batch Experiment A set of *Experiments* each defined by XML changes generated by the selected *Batch Criteria* to a template `.argos` file passed to SIERRA during stage 1 via `--template-input-file`.

For example, for the *Population Size* batch criteria, each experiment is defined by a single XML change to the provided `.argos` file: the number of robots in the swarm. Depending on the specifics you set for the *range* of sizes you are interested in, several experiments will be generated from the template `.argos` file, each differing from the template in the configured swarm size.

Experiment A single datapoint within a *Batch Experiment*; that is a single value of the *thing* that you are interested in varying across some range of experiments to see what happens (or doesn't happen).

Experimental Run Meaning is *Platform* dependent.

For `--platform=platform.argos` it is an *ARGoS* simulation that runs as part of an experiment. For `--platform=platform.ros1gazebo` it is a *Gazebo* simulation that runs as part of an experiment.

The number of simulations which will be run by SIERRA in stage 2 and averaged together by SIERRA in stage 3 is controlled by `--n-runs`.

All runs in within an *Experiment* are identical, with the exception of:

- Different values for the XML changes resulting from the different experiments they are part of, as defined by the batch criteria generating the batch experiment.
- Different random seeds

Output .csv A CSV file generated as an output from a single *Experimental Run*. It will (probably) contain a set of columns of representing outputs of interest, with rows corresponding to values captured throughout the run.

Averaged .csv A CSV file generated as from averaging files from multiple *Experimental Runs*. It will (probably) contain a set of columns of representing outputs of interest, with rows corresponding to values captured throughout the run (i.e., a time series).

Collated .csv A CSV file created by SIERRA during stage 4 (if inter-experiment graph generation is to be run). Collated CSV files contain a set columns, one per *Experiment* in the *Batch Experiment*. Each column is the captured value of a *single* column within an *Output .csv*. This is to capture a specific aspect of the behavior of the swarm within a batch experiment, for use in graph generation.

Summary .csv A CSV file created by SIERRA during stage 4 (if inter-experiment graph generation is to be run). A summary CSV file created from a *Collated .csv* file by taking the last row; this usually corresponds to steady-state behavior, which is what you are after. However, you can also capture transient behaviors by creating *Collated .csv* and *Summary .csv* files from captured *Experimental Run* outputs over short stretches of time–SIERRA does not know the difference.

Inter-Batch .csv A CSV file created by SIERRA during stage 5. An inter-batch CSV is created by "collating" columns from a *Summary .csv* present in multiple *Batch Experiments* into a single CSV. Used during stage 5.

Imagizing The process of turning a text file of some kind (e.g., CSV, `.gml`) into an image.

Platform The *context* in which experiments are run: either via a simulator of some kind, or a run-time framework for deploying code to one or more real robots.

Important: In SIERRA terminology, platform != OS. A given OS such has linux might support multiple platforms like ARGoS, ROS, etc, while a different OS like OSX might support only ARGoS.

Graph Category A semantic label attached to a set of graphs which are similar. For example, if you want to generate graphs about the different ways that robots allocate tasks, you might create a `LN_task_alloc` label, so that you can enable/disable all task allocation related graphs for one or more controllers easily when *configuring* your project.

Controller Category A semantic label attached to a set of controllers which are similar in some way. For example, if you have two controllers which use the same type of memory (say it's a "last N objects seen" memory), you could create a `LastN` category, and then define controllers within it, e.g., `LastN.Ring` and `LastN.DecayRing` for two controllers which have a ringbuffer of remembered objects and a decaying ringbuffer of remembered objects (i.e., an object is forgotten after some period of time even if it is not forced out of the ringbuffer by seeing a new object). See *configuring* your project.

Model A python implementation of a theoretical model of some kind. Can use empirical data from simulations/real robot experiments, or not, as needed. Intended to generate predictions of *something* which can then be plotted against empirical results for comparison.

Plugin A python package/module living in a directory on *`SIERRA_PLUGIN_PATH`* which contains functionality to extend SIERRA without modifying its core (i.e., customization of different parts of the pipeline). Plugins come in several flavors, all of which are handled equivalently by SIERRA:

- Pipeline plugins - Plugins which provide different ways of executing core parts of the SIERRA pipeline (e.g., how to run experiments).
- Platform plugins - Plugins which correspond to different *Platforms*.
- Project plugins - Plugins which correspond to different *Projects*.

CHAPTER

SIXTEEN

API REFERENCE

16.1 Core

16.1.1 sierra.core

sierra.core.cmdline

Core command line parsing and validation classes.

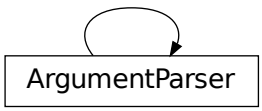
- *ArgumentParser*: SIERRA's argument parser overriding default argparse behavior.
- *BaseCmdline*: The base cmdline definition class for SIERRA for reusability.
- *BootstrapCmdline*: Defines the arguments that are used to bootstrap SIERRA/load plugins.
- *CoreCmdline*: Defines the core command line arguments for SIERRA using `argparse`.

class sierra.core.cmdline.**ArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=<class 'argparse.HelpFormatter'>*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*, *exit_on_error=True*)

SIERRA's argument parser overriding default argparse behavior.

Delivers a custom help message without listing options, as the options which SIERRA accepts differs dramatically depending on loaded plugins.

Inheritance



__doc__ = "SIERRA's argument parser overriding default argparse behavior.\n\n Delivers a custom help message without listing options, as the options which\n SIERRA accepts differs dramatically depending on loaded plugins.\n\n "

__module__ = 'sierra.core.cmdline'

error(*message: string*)

Prints a usage message incorporating the message to stderr and exits.

If you override this in a subclass, it should not return – it should either exit or raise an exception.

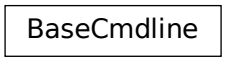
kHelpMsg = "Usage:\n\nsierra-cli [-v | --version] [OPTION]...\n\nWhat command line options SIERRA accepts depends on the loaded\n--project, --platform, --exec-env, and project-specific options.\n'man sierra-cli' will give you the full set of options that\ncomes with SIERRA.\n\n"

print_help(*file=None*)

class sierra.core.cmdline.**BaseCmdline**

The base cmdline definition class for SIERRA for reusability.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.cmdline', '__doc__': '\n The base cmdline definition class for SIERRA for reusability.\n\n ', 'stage_usage_doc': <staticmethod object>, 'bc_applicable_doc': <staticmethod object>, 'graphs_applicable_doc': <staticmethod object>, '__dict__': <attribute '__dict__' of 'BaseCmdline' objects>, '__weakref__': <attribute '__weakref__' of 'BaseCmdline' objects>, '__annotations__': {}})

__doc__ = '\n The base cmdline definition class for SIERRA for reusability.\n\n '

__module__ = 'sierra.core.cmdline'

__weakref__

list of weak references to the object (if defined)

static bc_applicable_doc(*criteria: List[str]*) → str

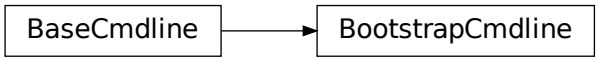
static graphs_applicable_doc(*graphs: List[str]*) → str

static stage_usage_doc(*stages: List[int]*, *omitted: str = 'If omitted: N/A.'*) → str

class sierra.core.cmdline.**BootstrapCmdline**

Defines the arguments that are used to bootstrap SIERRA/load plugins.

Inheritance



__doc__ = '\n Defines the arguments that are used to bootstrap SIERRA/load plugins.\n '

__init__() → None

__module__ = 'sierra.core.cmdline'

class sierra.core.cmdline.**CoreCmdline**(*parents: Optional[List[*sierra.core.cmdline.ArgumentParser*]]*, *stages: List[int]*)

Defines the core command line arguments for SIERRA using `argparse`.

Inheritance

__doc__ = 'Defines the core command line arguments for SIERRA using :class:`argparse`.\n\n '

__init__(*parents: Optional[List[*sierra.core.cmdline.ArgumentParser*]]*, *stages: List[int]*) → None

__module__ = 'sierra.core.cmdline'

init_cli(*stages: List[int]*) → None

Define cmdline arguments for stages 1-5.

init_multistage() → None

Define cmdline arguments which are used in multiple pipeline stages.

init_stage1() → None

Define cmdline arguments for stage 1.

init_stage2() → None

Define cmdline arguments for stage 2.

init_stage3() → None

Define cmdline arguments for stage 3.

init_stage4() → None

Define cmdline arguments for stage 4.

init_stage5() → None
Define cmdline arguments for stage 5.

scaffold_cli(*parents: Optional[List[*sierra.core.cmdline.ArgumentParser*]]*) → None
Scaffold CLI by defining the parser and common argument groups.

sierra.core.config

Contains all SIERRA hard-coded configuration in one place.

sierra.core.experiment

sierra.core.experiment.bindings

- *IParsedCmdlineConfigurer*: Modify arguments as needed for the platform or execution environment.
- *IExpRunShellCmdsGenerator*: Interface for generating the shell cmds to run for an *Experimental Run*.
- *IExpShellCmdsGenerator*: Interface for generating the shell cmds to run for an *Experiment*.
- *IExpConfigurer*: Perform addition configuration after creating experiments.
- *IExecEnvChecker*: Perform sanity checks for stage 2 execution environment.
- *ICmdlineParserGenerator*: Return the argparse object containing ALL options relevant to the platform.

class sierra.core.experiment.bindings.**IParsedCmdlineConfigurer**(*exec_env: str*)
Modify arguments as needed for the platform or execution environment.

Inheritance

__call__(*args: argparse.Namespace*) → None
Call self as a function.

__doc__ = '\n Modify arguments as needed for the platform or execution environment.\n '

__init__(*exec_env: str*) → None

__module__ = 'sierra.core.experiment.bindings'

class sierra.core.experiment.bindings.**IExpRunShellCmdsGenerator**(*cmdopts: Dict[str, Any]*, *criteria:* sierra.core.variables.batch_criteria.BatchCriteria, *n_robots: int*, *exp_num: int*)
Interface for generating the shell cmds to run for an *Experimental Run*.

This includes:

- The cmds to run the prior to the run.

- The cmds to executed the experimental run.
- Any post-run cleanup cmds before the next run is executed.

Depending on your environment, you may want to use *SIERRA_ARCH* (either in this function or your dispatch) to chose a version of your simulator compiled for a given architecture for maximum performance.

Parameters

- **cmdopts** – Dictionary of parsed cmdline options.
- **n_robots** – The configured # of robots for the experimental run.
- **exp_num** – The 0-based index of the experiment in the batch.

Inheritance

__doc__ = 'Interface for generating the shell cmds to run for an :term:`Experimental Run`.\n\n This includes:\n\n - The cmds to run the prior to the run.\n\n - The cmds to executed the experimental run.\n\n - Any post-run cleanup cmds before the next run is executed.\n\n Depending on your environment, you may want to use :envvar:`SIERRA_ARCH`\n (either in this function or your dispatch) to chose a version of your\n simulator compiled for a given architecture for maximum performance.\n\n Arguments:\n\n cmdopts: Dictionary of parsed cmdline options.\n\n n_robots: The configured # of robots for the experimental run.\n\n exp_num: The 0-based index of the experiment in the batch.\n\n '

__init__(*cmdopts: Dict[str, Any]*, *criteria:* sierra.core.variables.batch_criteria.BatchCriteria, *n_robots: int*, *exp_num: int*) → None

__module__ = 'sierra.core.experiment.bindings'

exec_run_cmds(*host: str*, *input_fpath: pathlib.Path*, *run_num: int*) → List[*sierra.core.types.ShellCmdSpec*]

Generate shell commands to execute a single *Experimental Run*.

This is (usually) a command to launch the simulation environment or start the controller code on a real robot, but it doesn't have to be. These commands are run during stage 2 after the *pre_run_cmds()* for each experimental run in the *Experiment*.

Called during stage 1.

Parameters

- **input_fpath** – Absolute path to the input/launch file for the experimental run.
- **run_num** – The 0 based ID of the experimental run.

post_run_cmds(*host: str*) → List[*sierra.core.types.ShellCmdSpec*]

Generate shell commands to run after an experimental run has finished.

These commands are run during stage 2 in the *same* sub-shell as the pre- and exec-run commands. These commands can include things like cleaning up/stopping background processes, visualization daemons, etc.

Called during stage 1.

pre_run_cmds(*host: str*, *input_fpath: pathlib.Path*, *run_num: int*) → List[*sierra.core.types.ShellCmdSpec*]
Generate shell commands to setup the experimental run environment.

These commands are run in stage 2 prior to each experimental run in the *Experiment*; that is prior to the `exec_run_cmds()`. These commands can include setting up things which are unique to/should not be shared between experimental runs within the experiment, such as launching daemons/background processes, setting environment variables, etc.

Called during stage 1.

Parameters

- **input_fpath** – Absolute path to the input/launch file for the experimental run.
- **run_num** – The 0 based ID of the experimental run.

class sierra.core.experiment.bindings.**IExpShellCmdsGenerator**(*cmdopts: Dict[str, Any]*, *exp_num: int*)

Interface for generating the shell cmds to run for an *Experiment*.

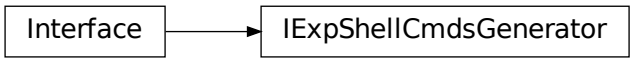
This includes:

- The cmds to run the prior to the experiment (before any *Experimental Runs*).
- The cmds to run the experiment.
- Any post-experiment cleanup cmds before the next *Experiment* is run.

Parameters

- **cmdopts** – Dictionary of parsed cmdline options.
- **exp_num** – The 0-based index of the experiment in the batch.

Inheritance




```
__doc__ = '\n Interface for generating the shell cmds to run for an
:term:`Experiment`.\n\n This includes:\n\n - The cmds to run the prior to the
experiment (before any\n :term:`Experimental Runs <Experimental Run>`).\n\n - The
cmds to run the experiment.\n\n - Any post-experiment cleanup cmds before the next
:term:`Experiment` is\n run.\n\n Arguments:\n\n cmdopts:  Dictionary of parsed
cmdline options.\n\n exp_num:  The 0-based index of the experiment in the batch.\n '
```

__init__(*cmdopts: Dict[str, Any]*, *exp_num: int*) → None

```
__module__ = 'sierra.core.experiment.bindings'
```

exec_exp_cmds(*exec_opts: Dict[str, str]*) → List[*sierra.core.types.ShellCmdSpec*]
Generate shell commands to execute an *Experiment*.

This is (usually) a single GNU parallel command, but it does not have to be. These commands are run in the *same* sub-shell as the pre- and post-exp commands during stage 2.

Called during stage 2.

Parameters `exec_opts` – A dictionary containing:

- `jobroot_path` - The root directory for the batch experiment.
- `exec_resume` - Is this a resume of a previously run experiment?
- `n_jobs` - How many parallel jobs are allowed per node?
- `joblog_path` - The logfile for output for the experiment run cmd (different than the *Project* code).
- `cmdfile_stem_path` - Stem of the file containing the launch cmds to run (one per line), all the way up to but not including the extension.
- `cmdfile_ext` - Extension of files containing the launch cmds to run.
- `nodefile` - Path to file containing compute resources for SIERRA to use to run experiments. See `--nodefile` and *SIERRA_NODEFILE* for details.

`post_exp_cmds()` → List[*sierra.core.types.ShellCmdSpec*]

Generate shell commands to run after an *Experiment* has finished.

Commands are run during stage 2 after all Experimental runs <Experimental Run>` for an *Experiment* have finished, but before the next experiment in the *Batch Experiment* is launched. These commands are run in the *same* sub-shell as the pre- and exec-exp commands.

These commands include things like cleaning up/stopping background processes, visualization daemons, etc.

Called during stage 1.

`pre_exp_cmds()` → List[*sierra.core.types.ShellCmdSpec*]

Generate shell commands to setup the environment for an *Experiment*.

These commands can include setting up things which are common to/should be the same for all experimental runs within the experiment, such as launching daemons/background processes needed by the platform, setting environment variables, etc. These commands are run prior to each experiment in the batch during stage 2, in a *new* sub-shell which SIERRA uses to run all *Experiment Runs* within the experiment.

Called during stage 1.

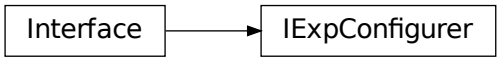
class `sierra.core.experiment.bindings.IExpConfigurer`(*cmdopts: Dict[str, Any]*)

Perform addition configuration after creating experiments.

E.g., creating directories store outputs in if they are not created by the simulator/*Project* code.

Parameters `cmdopts` – Dictionary of parsed cmdline options.

Inheritance



__doc__ = 'Perform addition configuration after creating experiments.\n\n E.g., creating directories store outputs in if they are not created by the\n simulator/:term:`Project` code.\n\n Arguments:\n\n cmdopts: Dictionary of parsed cmdline options.\n\n '

__init__(*cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.core.experiment.bindings'

cmdfile_paradigm() → str

Return the paradigm for the platform, in terms of GNU parallel cmds.

- `per-exp` - A single GNU parallel cmds file per experiment.
- `per-run` - A single GNU parallel cmds file per run.

for_exp(*exp_input_root: pathlib.Path*) → None

Configure an experiment.

Parameters **exp_input_root** – Absolute path to the input directory for the experiment.

for_exp_run(*exp_input_root: pathlib.Path*, *run_output_root: pathlib.Path*) → None

Configure an experimental run.

Parameters

- **exp_input_root** – Absolute path to the input directory for the experiment.
- **run_output_root** – Absolute path to the output directory for the experimental run.

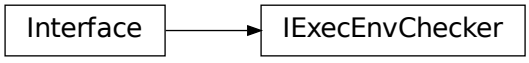
class sierra.core.experiment.bindings.**IExecEnvChecker**(*cmdopts: Dict[str, Any]*)

Perform sanity checks for stage 2 execution environment.

This is needed because stage 2 can run separate from stage 1, and we need to guarantee that the execution environment we verified during stage 1 is still valid.

Parameters **cmdopts** – Dictionary of parsed cmdline options.

Inheritance



__call__() → None
Call self as a function.

__doc__ = 'Perform sanity checks for stage 2 execution environment.\n\n This is needed because stage 2 can run separate from stage 1, and we need to\n guarantee that the execution environment we verified during stage 1 is still\n valid.\n\n Arguments:\n\n cmdopts: Dictionary of parsed cmdline options.\n\n '

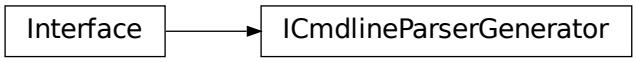
__init__(*cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.core.experiment.bindings'

class sierra.core.experiment.bindings.**ICmdlineParserGenerator**
Return the argparse object containing ALL options relevant to the platform.

This includes the options for whatever `--exec-env` are valid for the platform, making use of the `parents` option for the cmdline.

Inheritance



__call__() → argparse.ArgumentParser
Call self as a function.

__doc__ = 'Return the argparse object containing ALL options relevant to the platform.\n\n This includes the options for whatever ``--exec-env`` are valid for the\n platform, making use of the ``parents`` option for the cmdline.\n\n '

__module__ = 'sierra.core.experiment.bindings'

sierra.core.experiment.definition

Functionality for reading, writing, and manipulating experiment definitions.

Currently, experiments can be specified in:

- XML
- *XMLExpDef*: Read, write, and modify parsed XML files into experiment definitions.

class sierra.core.experiment.definition.**XMLExpDef**(*input_fpath: pathlib.Path*, *write_config: Optional[*sierra.core.experiment.xml.WriterConfig*] = None*)
Read, write, and modify parsed XML files into experiment definitions.

Functionality includes single tag removal/addition, single attribute change/add/remove.

input_filepath
The location of the XML file to process.

writer
The configuration for how the XML data will be written.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.experiment.definition', '__doc__': 'Read, write, and modify parsed XML files into experiment definitions.\n\n Functionality includes single tag removal/addition, single attribute\n change/add/remove.\n\n Attributes:\n\n input_filepath: The location of the XML file to process.\n\n writer: The configuration for how the XML data will be written.\n ', '__init__': <function XMLExpDef.__init__>, 'write_config_set': <function XMLExpDef.write_config_set>, 'write': <function XMLExpDef.write>, 'attr_get': <function XMLExpDef.attr_get>, 'attr_change': <function XMLExpDef.attr_change>, 'attr_add': <function XMLExpDef.attr_add>, 'has_tag': <function XMLExpDef.has_tag>, 'has_attr': <function XMLExpDef.has_attr>, 'tag_change': <function XMLExpDef.tag_change>, 'tag_remove': <function XMLExpDef.tag_remove>, 'tag_remove_all': <function XMLExpDef.tag_remove_all>, 'tag_add': <function XMLExpDef.tag_add>, '__dict__': <attribute '__dict__' of 'XMLExpDef' objects>, '__weakref__': <attribute '__weakref__' of 'XMLExpDef' objects>, '__annotations__': {}})

__doc__ = 'Read, write, and modify parsed XML files into experiment definitions.\n\n Functionality includes single tag removal/addition, single attribute\n change/add/remove.\n\n Attributes:\n\n input_filepath: The location of the XML file to process.\n\n writer: The configuration for how the XML data will be written.\n '

__init__(*input_fpath: pathlib.Path*, *write_config: Optional[*sierra.core.experiment.xml.WriterConfig*] = None*) → None

__module__ = 'sierra.core.experiment.definition'

__weakref__
list of weak references to the object (if defined)

attr_add(*path: str*, *attr: str*, *value: str*, *noprint: bool = False*) → bool
Add the specified attribute to the element matching the specified path.

Only the *FIRST* element matching the specified path searching from the tree root is modified.

Parameters

- **path** – An XPath expression that for the element containing the attribute to add. The element must exist or an error will be raised.
- **attr** – An XPath expression for the name of the attribute to change within the enclosing element.
- **value** – The value to set the attribute to.

attr_change(*path: str*, *attr: str*, *value: str*, *noprint: bool = False*) → bool
Change the specified attribute of the element at the specified path.

Only the attribute of the *FIRST* element matching the specified path is changed.

Parameters

- **path** – An XPath expression that for the element containing the attribute to change. The element must exist or an error will be raised.
- **attr** – An XPath expression for the name of the attribute to change within the enclosing element.
- **value** – The value to set the attribute to.

attr_get(*path: str*, *attr: str*)
Retrieve the specified attribute of the element at the specified path.

If it does not exist, None is returned.

has_attr(*path: str*, *attr: str*) → bool

has_tag(*path: str*) → bool

tag_add(*path: str*, *tag: str*, *attr: Dict[str, str] = {}*, *allow_dup: bool = True*, *noprint: bool = False*) → bool
Add tag name as a child element of enclosing parent.

tag_change(*path: str*, *tag: str*, *value: str*) → bool
Change the specified tag of the element matching the specified path.

Parameters

- **path** – An XPath expression that for the element containing the tag to change. The element must exist or an error will be raised.
- **tag** – An XPath expression of the tag to change within the enclosing element.
- **value** – The value to set the tag to.

tag_remove(*path: str*, *tag: str*, *noprint: bool = False*) → bool
Remove the specified child in the enclosing parent specified by the path.

If more than one tag matches, only one is removed. If the path does not exist, nothing is done.

Parameters

- **path** – An XPath expression that for the element containing the tag to remove. The element must exist or an error will be raised.
- **tag** – An XPath expression of the tag to remove within the enclosing element.

tag_remove_all(*path: str*, *tag: str*, *noprint: bool = False*) → bool
Remove the specified tag(s) in the enclosing parent specified by the path.

If more than one tag matches in the parent, all matching child tags are removed.

Parameters

- **path** – An XPath expression that for the element containing the tag to remove. The element must exist or an error will be raised.
- **tag** – An XPath expression for the tag to remove within the enclosing element.

write(*base_opath: pathlib.Path*) → None
Write the XML stored in the object to the filesystem.

write_config_set(*config:* sierra.core.experiment.xml.WriterConfig) → None
Set the write config for the object.

Provided for cases in which the configuration is dependent on whether or not certain tags are present in the input file.

sierra.core.experiment.spec

- *ExperimentSpec*: The specification for a single experiment with a batch.

class sierra.core.experiment.spec.**ExperimentSpec**(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria, *exp_num: int*, *cmdopts: Dict[str, Any]*)

The specification for a single experiment with a batch.

In the interest of DRY, this class collects the following common components:

- Experiment # within the batch
- Root input directory for all *Experimental Run* input files comprising the *Experiment*
- Pickle file path for the experiment
- Arena dimensions for the experiment
- Full scenario name

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.experiment.spec', '__doc__': '\n The specification for a single experiment with a batch.\n\n In the interest of DRY, this class collects the following common components:\n\n - Experiment # within the batch\n\n - Root input directory for all :term:`Experimental Run` input files\n comprising the :term:`Experiment`\n\n - Pickle file path for the experiment\n\n - Arena dimensions for the experiment\n\n - Full scenario name\n ', '__init__': <function ExperimentSpec.__init__>, '__dict__': <attribute '__dict__' of 'ExperimentSpec' objects>, '__weakref__': <attribute '__weakref__' of 'ExperimentSpec' objects>, '__annotations__': {}})

__doc__ = '\n The specification for a single experiment with a batch.\n\n In the interest of DRY, this class collects the following common components:\n\n - Experiment # within the batch\n\n - Root input directory for all :term:`Experimental Run` input files\n comprising the :term:`Experiment`\n\n - Pickle file path for the experiment\n\n - Arena dimensions for the experiment\n\n - Full scenario name\n '

__init__(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria, *exp_num: int*, *cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.core.experiment.spec'

__weakref__
list of weak references to the object (if defined)

sierra.core.experiment.xml

Helper classes for XML experiment definitions.

Adding/removing tags, modifying attributes, configuration for how to write XML files.

- *AttrChange*: Specification for a change to an existing XML attribute.
- *AttrChangeSet*: Data structure for *AttrChange* objects.
- *TagAdd*: Specification for adding a new XML tag.
- *TagAddList*: Data structure for *TagAdd* objects.
- *TagRm*: Specification for removal of an existing XML tag.
- *TagRmList*: Data structure for *TagRm* objects.
- *WriterConfig*: Config for writing *XMLExpDef*.

class sierra.core.experiment.xml.**AttrChange**(*path: str*, *attr: str*, *value: Union[str, int, float]*)
Specification for a change to an existing XML attribute.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.experiment.xml', '__doc__': '\n Specification for a change to an existing XML attribute.\n ', '__init__': <function AttrChange.__init__>, '__iter__': <function AttrChange.__iter__>, '__repr__': <function AttrChange.__repr__>, '__dict__': <attribute '__dict__' of 'AttrChange' objects>, '__weakref__': <attribute '__weakref__' of 'AttrChange' objects>, '__annotations__': {}})

__doc__ = '\n Specification for a change to an existing XML attribute.\n '

__init__(*path: str*, *attr: str*, *value: Union[str, int, float]*) → None

__iter__()

__module__ = 'sierra.core.experiment.xml'

__repr__() → str
Return repr(self).

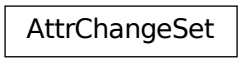
__weakref__
list of weak references to the object (if defined)

class sierra.core.experiment.xml.**AttrChangeSet**(**args:* sierra.core.experiment.xml.AttrChange)

Data structure for *AttrChange* objects.

The order in which attributes are changed doesn't matter from the standpoint of correctness (i.e., different orders won't cause crashes).

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.experiment.xml', '__doc__': "\n Data structure for :class:`AttrChange` objects.\n\n The order in which attributes are changed doesn't matter from the standpoint\n of correctness (i.e., different orders won't cause crashes).\n\n ", 'unpickle': <staticmethod object>, '__init__': <function AttrChangeSet.__init__>, '__len__': <function AttrChangeSet.__len__>, '__iter__': <function AttrChangeSet.__iter__>, '__ior__': <function AttrChangeSet.__ior__>, '__or__': <function AttrChangeSet.__or__>, '__repr__': <function AttrChangeSet.__repr__>, 'add': <function AttrChangeSet.add>, 'pickle': <function AttrChangeSet.pickle>, '__dict__': <attribute '__dict__' of 'AttrChangeSet' objects>, '__weakref__': <attribute '__weakref__' of 'AttrChangeSet' objects>, '__annotations__': {}})

__doc__ = "\n Data structure for :class:`AttrChange` objects.\n\n The order in which attributes are changed doesn't matter from the standpoint\n of correctness (i.e., different orders won't cause crashes).\n\n "

__init__(**args:* sierra.core.experiment.xml.AttrChange) → None

__ior__(*other:* sierra.core.experiment.xml.AttrChangeSet) → *sierra.core.experiment.xml.AttrChangeSet*

__iter__() → Iterator[*sierra.core.experiment.xml.AttrChange*]

__len__() → int

__module__ = 'sierra.core.experiment.xml'

__or__(*other:* sierra.core.experiment.xml.AttrChangeSet) → *sierra.core.experiment.xml.AttrChangeSet*

__repr__() → str

Return repr(self).

__weakref__

list of weak references to the object (if defined)

add(*chg:* sierra.core.experiment.xml.AttrChange) → None

pickle(*fpath:* *pathlib.Path*, *delete:* *bool* = *False*) → None

static unpickle(*fpath:* *pathlib.Path*) → *sierra.core.experiment.xml.AttrChangeSet*

Unpickle XML changes.

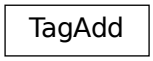
You don't know how many there are, so go until you get an exception.

class sierra.core.experiment.xml.**TagAdd**(*path: str*, *tag: str*, *attr: Dict[str, str]*, *allow_dup: bool*)

Specification for adding a new XML tag.

The tag may be added idempotently, or duplicates can be allowed.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.experiment.xml', '__doc__': '\n Specification for adding a new XML tag.\n\n The tag may be added idempotently, or duplicates can be allowed.\n ', 'as_root': <staticmethod object>, '__init__': <function TagAdd.__init__>, '__iter__': <function TagAdd.__iter__>, '__repr__': <function TagAdd.__repr__>, '__dict__': <attribute '__dict__' of 'TagAdd' objects>, '__weakref__': <attribute '__weakref__' of 'TagAdd' objects>, '__annotations__': {}})

__doc__ = '\n Specification for adding a new XML tag.\n\n The tag may be added idempotently, or duplicates can be allowed.\n '

__init__(*path: str*, *tag: str*, *attr: Dict[str, str]*, *allow_dup: bool*)

Init the object.

Parameters

- **path** – The path to the **parent** tag you want to add a new tag under, in XPath syntax. If None, then the tag will be added as the root XML tag.
- **tag** – The name of the tag to add.
- **attr** – A dictionary of (attribute, value) pairs to also create as children of the new tag when creating the new tag.

__iter__()

__module__ = 'sierra.core.experiment.xml'

__repr__() → str

Return repr(self).

__weakref__

list of weak references to the object (if defined)

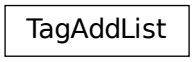
static as_root(*tag: str*, *attr: Dict[str, str]*) → *sierra.core.experiment.xml.TagAdd*

class sierra.core.experiment.xml.**TagAddList**(**args:* sierra.core.experiment.xml.TagAdd)

Data structure for *TagAdd* objects.

The order in which tags are added matters (i.e., if you add dependent tags in the wrong order you will get an exception), hence the list representation.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.experiment.xml', '__doc__': '\n Data structure for :class:`TagAdd` objects.\n\n The order in which tags are added matters (i.e., if you add dependent tags\n in the wrong order you will get an exception), hence the list\n representation.\n ', 'unpickle': <staticmethod object>, '__init__': <function TagAddList.__init__>, '__len__': <function TagAddList.__len__>, '__iter__': <function TagAddList.__iter__>, '__repr__': <function TagAddList.__repr__>, 'extend': <function TagAddList.extend>, 'append': <function TagAddList.append>, 'prepend': <function TagAddList.prepend>, 'pickle': <function TagAddList.pickle>, '__dict__': <attribute '__dict__' of 'TagAddList' objects>, '__weakref__': <attribute '__weakref__' of 'TagAddList' objects>, '__annotations__': {}})

__doc__ = '\n Data structure for :class:`TagAdd` objects.\n\n The order in which tags are added matters (i.e., if you add dependent tags\n in the wrong order you will get an exception), hence the list\n representation.\n '

__init__(**args:* sierra.core.experiment.xml.TagAdd) → None

__iter__() → Iterator[*sierra.core.experiment.xml.TagAdd*]

__len__() → int

__module__ = 'sierra.core.experiment.xml'

__repr__() → str
Return repr(self).

__weakref__
list of weak references to the object (if defined)

append(*other:* sierra.core.experiment.xml.TagAdd) → None

extend(*other:* sierra.core.experiment.xml.TagAddList) → None

pickle(*fpath:* *pathlib.Path*, *delete:* *bool* = *False*) → None

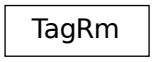
prepend(*other:* sierra.core.experiment.xml.TagAdd) → None

static unpickle(*fpath:* *pathlib.Path*) → Optional[*sierra.core.experiment.xml.TagAddList*]
Unpickle XML modifications.

You don't know how many there are, so go until you get an exception.

class sierra.core.experiment.xml.**TagRm**(*path:* *str*, *tag:* *str*)
Specification for removal of an existing XML tag.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.experiment.xml', '__doc__': '\n Specification for removal of an existing XML tag.\n ', '__init__': <function TagRm.__init__>, '__iter__': <function TagRm.__iter__>, '__repr__': <function TagRm.__repr__>, '__dict__': <attribute '__dict__' of 'TagRm' objects>, '__weakref__': <attribute '__weakref__' of 'TagRm' objects>, '__annotations__': {}})

__doc__ = '\n Specification for removal of an existing XML tag.\n '

__init__(*path: str*, *tag: str*)
Init the object.

Parameters

- **path** – The path to the **parent** of the tag you want to remove, in XPath syntax.
- **tag** – The name of the tag to remove.

__iter__()

__module__ = 'sierra.core.experiment.xml'

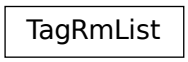
__repr__() → str
Return repr(self).

__weakref__
list of weak references to the object (if defined)

class sierra.core.experiment.xml.**TagRmList**(**args:* sierra.core.experiment.xml.TagRm)
Data structure for *TagRm* objects.

The order in which tags are removed matters (i.e., if you remove dependent tags in the wrong order you will get an exception), hence the list representation.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.experiment.xml', '__doc__': '\n Data structure for :class:`TagRm` objects.\n\n The order in which tags are removed matters (i.e., if you remove dependent\n tags in the wrong order you will get an exception), hence the list\n representation.\n\n ', '__init__': <function TagRmList.__init__>, '__len__': <function TagRmList.__len__>, '__iter__': <function TagRmList.__iter__>, '__repr__': <function TagRmList.__repr__>, 'extend': <function TagRmList.extend>, 'append': <function TagRmList.append>, 'pickle': <function TagRmList.pickle>, '__dict__': <attribute '__dict__' of 'TagRmList' objects>, '__weakref__': <attribute '__weakref__' of 'TagRmList' objects>, '__annotations__': {}})

__doc__ = '\n Data structure for :class:`TagRm` objects.\n\n The order in which tags are removed matters (i.e., if you remove dependent\n tags in the wrong order you will get an exception), hence the list\n representation.\n\n '

__init__(**args:* sierra.core.experiment.xml.TagRm) → None

__iter__() → Iterator[*sierra.core.experiment.xml.TagRm*]

__len__() → int

__module__ = 'sierra.core.experiment.xml'

__repr__() → str
Return repr(self).

__weakref__
list of weak references to the object (if defined)

append(*other:* sierra.core.experiment.xml.TagRm) → None

extend(*other:* sierra.core.experiment.xml.TagRmList) → None

pickle(*fpath:* *pathlib.Path*, *delete:* *bool* = *False*) → None

class sierra.core.experiment.xml.**WriterConfig**(*values: List[dict]*)
Config for writing *XMLExpDef*.

Different parts of the XML tree can be written to multiple XML files.

values
Dict with the following possible key, value pairs:

src_parent - The parent of the root of the XML tree specifying a sub-tree to write out as a child of `dest_root`. This key is required.

src_tag - The name of the tag within `src_parent` to write out. This key is required.

dest_parent - The new name of `src_root` when writing out the partial XML tree to a new file. This key is optional.

create_tags - Additional tags to create under `dest_parent`. Must form a tree with a single root.

opath_leaf - Additional bits added to whatever the opath file stem that is set for the *XMLExpDef* instance. This key is optional. Can be used to add an extension.

child_grafts - Additional bits of the XML tree to add under the new `dest_root/src_tag`, specified as a list of XPath strings. You can't just have multiple src_roots because that makes unambiguous renaming of `src_root` -> `dest_root` impossible. This key is optional.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.experiment.xml', '__doc__': "Config for writing :class:`~sierra.core.experiment.definition.XMLExpDef`.\n\n Different parts of the XML tree can be written to multiple XML files.\n\n Attributes:\n\n values: Dict with the following possible key, value pairs:\n\n ``src_parent`` - The parent of the root of the XML tree\n specifying a sub-tree to write out as a child\n of ``dest_root``. This key is required.\n\n ``src_tag`` - The name of the tag within ``src_parent`` to write\n out. This key is required.\n\n ``dest_parent`` - The new name of ``src_root`` when writing out\n the partial XML tree to a new file. This key\n is optional.\n\n ``create_tags`` - Additional tags to create under\n ``dest_parent``. Must form a tree with a\n single root.\n\n ``opath_leaf`` - Additional bits added to whatever the opath\n file stem that is set for the\n :class:`~sierra.core.experiment.definition.XMLExpDef`\n instance. This key is optional. Can be used to\n add an extension.\n\n ``child_grafts`` - Additional bits of the XML tree to add under\n the new ``dest_root/src_tag``, specified as a\n list of XPath strings. You can't just have\n multiple src_roots because that makes\n unambiguous renaming of ``src_root`` ->\n ``dest_root`` impossible. This key is\n optional.\n\n ", '__init__': <function WriterConfig.__init__>, 'add': <function WriterConfig.add>, '__dict__': <attribute '__dict__' of 'WriterConfig' objects>, '__weakref__': <attribute '__weakref__' of 'WriterConfig' objects>, '__annotations__': {}})

__doc__ = "Config for writing :class:`~sierra.core.experiment.definition.XMLExpDef`.\n\n Different parts of the XML tree can be written to multiple XML files.\n\n Attributes:\n\n values: Dict with the following possible key, value pairs:\n\n ``src_parent`` - The parent of the root of the XML tree\n specifying a sub-tree to write out as a child\n of ``dest_root``. This key is required.\n\n ``src_tag`` - The name of the tag within ``src_parent`` to write\n out. This key is required.\n\n ``dest_parent`` - The new name of ``src_root`` when writing out\n the partial XML tree to a new file. This key\n is optional.\n\n ``create_tags`` - Additional tags to create under\n ``dest_parent``. Must form a tree with a\n single root.\n\n ``opath_leaf`` - Additional bits added to whatever the opath\n file stem that is set for the\n :class:`~sierra.core.experiment.definition.XMLExpDef`\n instance. This key is optional. Can be used to\n add an extension.\n\n ``child_grafts`` - Additional bits of the XML tree to add under\n the new ``dest_root/src_tag``, specified as a\n list of XPath strings. You can't just have\n multiple src_roots because that makes\n unambiguous renaming of ``src_root`` ->\n ``dest_root`` impossible. This key is\n optional.\n\n "

__init__(*values: List[dict]*) → None

__module__ = 'sierra.core.experiment.xml'

__weakref__
list of weak references to the object (if defined)

add(*value: dict*) → None

sierra.core.generators

sierra.core.generators.controller_generator_parser

- *ControllerGeneratorParser*: Parse the controller generator specification from cmdline arguments.

class sierra.core.generators.controller_generator_parser.**ControllerGeneratorParser**

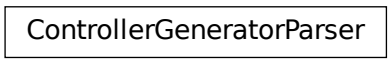
Parse the controller generator specification from cmdline arguments.

Used later to create generator classes to make modifications to template input files.

Format for pair is: <category>.<controller>.

Returns Parsed controller specification, the generator is missing from the command line altogether; this can occur if the user is only running stage [4,5], and is not an error. In that case, None is returned.

Inheritance



__call__(*args: argparse.Namespace*) → Optional[str]

Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.generators.controller_generator_parser', '__doc__': 'Parse the controller generator specification from cmdline arguments.\n\n Used later to create generator classes to make modifications to template\n input files.\n\n Format for pair is: ``<category>.<controller>``.\n\n Return:\n\n Parsed controller specification, the generator is missing from the command\n line altogether; this can occur if the user is only running stage [4,5],\n and is not an error. In that case, None is returned.\n\n ', '__call__': <function ControllerGeneratorParser.__call__>, '__dict__': <attribute '__dict__' of 'ControllerGeneratorParser' objects>, '__weakref__': <attribute '__weakref__' of 'ControllerGeneratorParser' objects>, '__annotations__': {}})

__doc__ = 'Parse the controller generator specification from cmdline arguments.\n\n Used later to create generator classes to make modifications to template\n input files.\n\n Format for pair is: ``<category>.<controller>``.\n\n Return:\n\n Parsed controller specification, the generator is missing from the command\n line altogether; this can occur if the user is only running stage [4,5],\n and is not an error. In that case, None is returned.\n\n '

__module__ = 'sierra.core.generators.controller_generator_parser'

__weakref__

list of weak references to the object (if defined)

sierra.core.generators.exp_creator

Experiment creation classes.

Experiment creation takes an experiment definition *generated* by classes in `exp_generators.py` and writes the experiment to the filesystem.

- *ExpCreator*: Instantiate a generated experiment from an experiment definition.
- *BatchExpCreator*: Instantiate a *Batch Experiment*.

class sierra.core.generators.exp_creator.**ExpCreator**(*cmdopts: Dict[str, Any]*, *criteria:* sierra.core.variables.batch_criteria.BatchCriteria, *template_ipath: pathlib.Path*, *exp_input_root: pathlib.Path*, *exp_output_root: pathlib.Path*, *exp_num: int*)

Instantiate a generated experiment from an experiment definition.

Takes generated *Experiment* definitions and writes them to the filesystem.

template_ipath
Absolute path to the template XML configuration file.

exp_input_root
Absolute path to experiment directory where generated XML input files for this experiment should be written.

exp_output_root
Absolute path to root directory for run outputs for this experiment.

cmdopts
Dictionary containing parsed cmdline options.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.generators.exp_creator', '__doc__': 'Instantiate a generated experiment from an experiment definition.\n\n Takes generated :term:`Experiment` definitions and writes them to the\n filesystem.\n\n Attributes:\n\n template_ipath: Absolute path to the template XML configuration file.\n\n exp_input_root: Absolute path to experiment directory where generated\n XML input files for this experiment should be written.\n\n exp_output_root: Absolute path to root directory for run outputs\n for this experiment.\n\n cmdopts: Dictionary containing parsed cmdline options.\n\n ', '__init__': <function ExpCreator.__init__>, 'from_def': <function ExpCreator.from_def>, '_create_exp_run': <function ExpCreator._create_exp_run>, '_get_launch_file_stempath': <function ExpCreator._get_launch_file_stempath>, '_update_cmds_file': <function ExpCreator._update_cmds_file>, '__dict__': <attribute '__dict__' of 'ExpCreator' objects>, '__weakref__': <attribute '__weakref__' of 'ExpCreator' objects>, '__annotations__': {}})

__doc__ = 'Instantiate a generated experiment from an experiment definition.\n\n Takes generated :term:`Experiment` definitions and writes them to the\n filesystem.\n\n Attributes:\n\n template_ipath: Absolute path to the template XML configuration file.\n\n exp_input_root: Absolute path to experiment directory where generated\n XML input files for this experiment should be written.\n\n exp_output_root: Absolute path to root directory for run outputs\n for this experiment.\n\n cmdopts: Dictionary containing parsed cmdline options.\n\n '

__init__(*cmdopts: Dict[str, Any]*, *criteria:* sierra.core.variables.batch_criteria.BatchCriteria, *template_ipath: pathlib.Path*, *exp_input_root: pathlib.Path*, *exp_output_root: pathlib.Path*, *exp_num: int*) → None

__module__ = 'sierra.core.generators.exp_creator'

__weakref__
: list of weak references to the object (if defined)

_create_exp_run(*run_exp_def:* sierra.core.experiment.definition.XMLExpDef, *cmds_generator*, *run_num: int*) → None

_get_launch_file_stempath(*run_num: int*) → pathlib.Path
: File is named as <template input file stem>_run<run_num>.

_update_cmds_file(*cmds_file*, *cmds_generator:* sierra.core.experiment.bindings.IExpRunShellCmdsGenerator, *paradigm: str*, *run_num: int*, *launch_stem_path: pathlib.Path*, *for_host: str*) → None
: Add command to launch a given experimental run to the command file.

from_def(*exp_def:* sierra.core.experiment.definition.XMLExpDef)
: Create all experimental runs by writing input files to filesystem.

  The passed *XMLExpDef* object contains all changes that should be made to all runs in the experiment. Additional changes to create a set of unique runs from which distributions of system behavior can be meaningfully computed post-hoc are added.

class sierra.core.generators.exp_creator.**BatchExpCreator**(*criteria:* sierra.core.variables.batch_criteria.BatchCriteria, *cmdopts: Dict[str, Any]*)

Instantiate a *Batch Experiment*.

Calls *ExpCreator* on each experimental definition in the batch

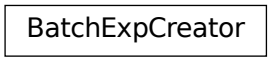
batch_config_template
: Absolute path to the root template XML configuration file.

batch_input_root
Root directory for all generated XML input files all experiments should be stored (relative to current dir or absolute). Each experiment will get a directory within this root to store the xml input files for the experimental runs comprising an experiment; directory name determined by the batch criteria used.

batch_output_root
Root directory for all experiment outputs. Each experiment will get a directory 'exp<n>' in this directory for its outputs.

criteria
BatchCriteria derived object instance created from cmdline definition.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.generators.exp_creator', '__doc__': "Instantiate a :term:`Batch Experiment`.\n\n Calls :class:`~sierra.core.generators.exp_creator.ExpCreator` on each\n experimental definition in the batch\n\n Attributes:\n\n batch_config_template: Absolute path to the root template XML\n configuration file.\n\n batch_input_root: Root directory for all generated XML input files all\n experiments should be stored (relative to current dir\n or absolute). Each experiment will get a directory\n within this root to store the xml input files for the\n experimental runs comprising an experiment; directory\n name determined by the batch criteria used.\n\n batch_output_root: Root directory for all experiment outputs. Each\n experiment will get a directory 'exp<n>' in this\n directory for its outputs.\n\n criteria: :class:`~sierra.core.variables.batch_criteria.BatchCriteria`\n derived object instance created from cmdline definition.\n\n ", '__init__': <function BatchExpCreator.__init__>, 'create': <function BatchExpCreator.create>, '__dict__': <attribute '__dict__' of 'BatchExpCreator' objects>, '__weakref__': <attribute '__weakref__' of 'BatchExpCreator' objects>, '__annotations__': {}})

__doc__ = "Instantiate a :term:`Batch Experiment`.\n\n Calls :class:`~sierra.core.generators.exp_creator.ExpCreator` on each\n experimental definition in the batch\n\n Attributes:\n\n batch_config_template: Absolute path to the root template XML\n configuration file.\n\n batch_input_root: Root directory for all generated XML input files all\n experiments should be stored (relative to current dir\n or absolute). Each experiment will get a directory\n within this root to store the xml input files for the\n experimental runs comprising an experiment; directory\n name determined by the batch criteria used.\n\n batch_output_root: Root directory for all experiment outputs. Each\n experiment will get a directory 'exp<n>' in this\n directory for its outputs.\n\n criteria: :class:`~sierra.core.variables.batch_criteria.BatchCriteria`\n derived object instance created from cmdline definition.\n\n "

__init__(*criteria:* sierra.core.variables.batch_criteria.BatchCriteria, *cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.core.generators.exp_creator'

__weakref__
list of weak references to the object (if defined)

create(*generator:* sierra.core.generators.exp_generators.BatchExpDefGenerator) → None

sierra.core.generators.exp_generators

Experiment generation classes.

Experiment generation modifies the *XMLExpDef* object built from the specified batch criteria as follows:

- Platform-specific modifications common to all batch experiments
- Project-specific modifications common to all batch experiments
- Modifications generated by the selected controller+scenario

NOTE:: Generated definitions from batch criteria are not handled here; they are already generated to scaffold the batch experiment when experiment generation is run.

- *BatchExpDefGenerator*: Generate experiment definitions for a *Batch Experiment*.

class sierra.core.generators.exp_generators.**BatchExpDefGenerator**(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatch, *controller_name: str*, *scenario_basename: str*, *cmdopts: Dict[str, Any]*)

Generate experiment definitions for a *Batch Experiment*.

Does not create the batch experiment after generation.

batch_config_template
Absolute path to the root template XML configuration file.

batch_input_root
Root directory for all generated XML input files all experiments should be stored (relative to current dir or absolute). Each experiment will get a directory within this root to store the xml input files for the set of *Experimental Runs* comprising an *Experiment*; directory name determined by the batch criteria used.

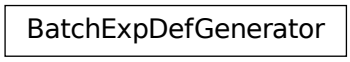
batch_output_root
Root directory for all experiment outputs (relative to current dir or absolute). Each experiment will get a directory 'exp<n>' in this directory for its outputs.

criteria
BatchCriteria derived object instance created from cmdline definition.

controller_name
Name of controller generator to use.

scenario_basename
Name of scenario generator to use.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.generators.exp_generators', '__doc__': "Generate experiment definitions for a :term:`Batch Experiment`.\n\n Does not create the batch experiment after generation.\n\n Attributes:\n\n batch_config_template: Absolute path to the root template XML\n configuration file.\n\n batch_input_root: Root directory for all generated XML input files all\n experiments should be stored (relative to current\n dir or absolute). Each experiment will get a\n directory within this root to store the xml input\n files for the set of :term:`Experimental Runs\n <Experimental Run>` comprising an\n :term:`Experiment`; directory name determined by\n the batch criteria used.\n\n batch_output_root: Root directory for all experiment outputs (relative\n to current dir or absolute). Each experiment will get\n a directory 'exp<n>' in this directory for its\n outputs.\n\n criteria: :class:`~sierra.core.variables.batch_criteria.BatchCriteria`\n derived object instance created from cmdline definition.\n\n controller_name: Name of controller generator to use.\n\n scenario_basename: Name of scenario generator to use.\n\n ", '__init__': <function BatchExpDefGenerator.__init__>, 'generate_defs': <function BatchExpDefGenerator.generate_defs>, '_create_exp_generator': <function BatchExpDefGenerator._create_exp_generator>, '__dict__': <attribute '__dict__' of 'BatchExpDefGenerator' objects>, '__weakref__': <attribute '__weakref__' of 'BatchExpDefGenerator' objects>, '__annotations__': {}})

__doc__ = "Generate experiment definitions for a :term:`Batch Experiment`.\n\n Does not create the batch experiment after generation.\n\n Attributes:\n\n batch_config_template: Absolute path to the root template XML\n configuration file.\n\n batch_input_root: Root directory for all generated XML input files all\n experiments should be stored (relative to current\n dir or absolute). Each experiment will get a\n directory within this root to store the xml input\n files for the set of :term:`Experimental Runs\n <Experimental Run>` comprising an\n :term:`Experiment`; directory name determined by\n the batch criteria used.\n\n batch_output_root: Root directory for all experiment outputs (relative\n to current dir or absolute). Each experiment will get\n a directory 'exp<n>' in this directory for its\n outputs.\n\n criteria: :class:`~sierra.core.variables.batch_criteria.BatchCriteria`\n derived object instance created from cmdline definition.\n\n controller_name: Name of controller generator to use.\n\n scenario_basename: Name of scenario generator to use.\n\n "

__init__(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria, *controller_name: str*, *scenario_basename: str*, *cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.core.generators.exp_generators'

__weakref__
list of weak references to the object (if defined)

_create_exp_generator(*exp_num: int*)
Create the joint scenario+controller generator from command line definitions.

Parameters **exp_num** – Experiment number in the batch

generate_defs() → List[*sierra.core.experiment.definition.XMLExpDef*]
Generate and return the batch experiment definition.

Returns A list of experiment definitions (one for each experiment in the batch).

sierra.core.generators.generator_factory

Factory for combining controller+scenario XML modification generators.

By combining them together, the result can be easily used to apply modifications to the template XML file to scaffold the batch experiment.

- *joint_generator_create()*: Combine controller and scenario XML modification generators together.
- *scenario_generator_create()*: Create a scenario generator using the plugin search path.
- *controller_generator_create()*: Create a controller generator from the cmdline specification.

sierra.core.generators.generator_factory.**joint_generator_create**(*controller*, *scenario*)
Combine controller and scenario XML modification generators together.

sierra.core.generators.generator_factory.**scenario_generator_create**(*exp_spec:* sierra.core.experiment.spec.ExperimentSpec, *controller*, ***kwargs*)
Create a scenario generator using the plugin search path.

sierra.core.generators.generator_factory.**controller_generator_create**(*controller: str*, *config_root: pathlib.Path*, *cmdopts: Dict[str, Any]*, *exp_spec:* sierra.core.experiment.spec.ExperimentSpec)
Create a controller generator from the cmdline specification.

- *ControllerGenerator*: Generate XML changes for a selected `--controller`.

class sierra.core.generators.generator_factory.**ControllerGenerator**(*controller: str*, *config_root: pathlib.Path*, *cmdopts: Dict[str, Any]*, *exp_spec:* sierra.core.experiment.spec.ExperimentSpec)
Generate XML changes for a selected `--controller`.

If the specified controller is not found in `controllers.yaml` for the loaded *Project*, an assert will be triggered.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.generators.generator_factory', '__doc__': 'Generate XML changes for a selected ``--controller``.\n\n If the specified controller is not found in ``controllers.yaml`` for the\n loaded :term:`Project`, an assert will be triggered.\n ', '__init__': <function ControllerGenerator.__init__>, 'generate': <function ControllerGenerator.generate>, '_pp_for_tag_add': <function ControllerGenerator._pp_for_tag_add>, '_do_tag_add': <function ControllerGenerator._do_tag_add>, '_generate_controller_support': <function ControllerGenerator._generate_controller_support>, '_generate_controller': <function ControllerGenerator._generate_controller>, '__dict__': <attribute '__dict__' of 'ControllerGenerator' objects>, '__weakref__': <attribute '__weakref__' of 'ControllerGenerator' objects>, '__annotations__': {}})

__doc__ = 'Generate XML changes for a selected ``--controller``.\n\n If the specified controller is not found in ``controllers.yaml`` for the\n loaded :term:`Project`, an assert will be triggered.\n '

__init__(*controller: str*, *config_root: pathlib.Path*, *cmdopts: Dict[str, Any]*, *exp_spec:* sierra.core.experiment.spec.ExperimentSpec) → None

__module__ = 'sierra.core.generators.generator_factory'

__weakref__
list of weak references to the object (if defined)

_do_tag_add(*exp_def:* sierra.core.experiment.definition.XMLExpDef, *add:* sierra.core.experiment.xml.TagAdd) → None

_generate_controller(*exp_def:* sierra.core.experiment.definition.XMLExpDef) → None

_generate_controller_support(*exp_def:* sierra.core.experiment.definition.XMLExpDef) → None

_pp_for_tag_add(*add:* sierra.core.experiment.xml.TagAdd, *robot_id: Optional[int] = None*) → *sierra.core.experiment.xml.TagAdd*

generate(*exp_def:* sierra.core.experiment.definition.XMLExpDef) → *sierra.core.experiment.definition.XMLExpDef*
Generate all modifications to the experiment definition from the controller.

Does not save.

sierra.core.graphs

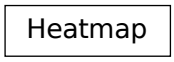
sierra.core.graphs.heatmap

- *Heatmap*: Generates a X vs. Y vs. Z heatmap plot of a `.mean` file.
- *DualHeatmap*: Generates a side-by-side plot of two heataps from two CSV files.
- *HeatmapSet*: Generates a *Heatmap* plot for each of the specified I/O path pairs.

class sierra.core.graphs.heatmap.**Heatmap**(*input_fpath: pathlib.Path*, *output_fpath: pathlib.Path*, *title: str*, *xlabel: str*, *ylabel: str*, *zlabel: Optional[str] = None*, *large_text: bool = False*, *xtick_labels: Optional[List[str]] = None*, *ytick_labels: Optional[List[str]] = None*, *transpose: bool = False*, *interpolation: Optional[str] = None*)
Generates a X vs. Y vs. Z heatmap plot of a `.mean` file.

If the necessary .mean file does not exist, the graph is not generated.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.graphs.heatmap', '__doc__': '\n Generates a X vs. Y vs. Z heatmap plot of a ``.mean`` file.\n\n If the necessary .mean file does not exist, the graph is not generated.\n\n ', 'set_graph_size': <staticmethod object>, '__init__': <function Heatmap.__init__>, 'generate': <function Heatmap.generate>, '_plot_df': <function Heatmap._plot_df>, '_plot_colorbar': <function Heatmap._plot_colorbar>, '_plot_ticks': <function Heatmap._plot_ticks>, '__dict__': <attribute '__dict__' of 'Heatmap' objects>, '__weakref__': <attribute '__weakref__' of 'Heatmap' objects>, '__annotations__': {}})

__doc__ = '\n Generates a X vs. Y vs. Z heatmap plot of a ``.mean`` file.\n\n If the necessary .mean file does not exist, the graph is not generated.\n\n '

__init__(*input_fpath: pathlib.Path*, *output_fpath: pathlib.Path*, *title: str*, *xlabel: str*, *ylabel: str*, *zlabel: Optional[str] = None*, *large_text: bool = False*, *xtick_labels: Optional[List[str]] = None*, *ytick_labels: Optional[List[str]] = None*, *transpose: bool = False*, *interpolation: Optional[str] = None*) → None

__module__ = 'sierra.core.graphs.heatmap'

__weakref__
: list of weak references to the object (if defined)

_plot_colorbar(*ax*) → None
: Put the Z-axis colorbar on the plot.

_plot_df(*df: pandas.core.frame.DataFrame*, *opath: pathlib.Path*) → None
: Given a dataframe read from a file, plot it as a heatmap.

_plot_ticks(*ax*) → None
: Plot X,Y ticks and their corresponding labels.

generate() → None

static set_graph_size(*df: pandas.core.frame.DataFrame*, *fig*) → None
: Set graph X,Y size based on dataframe dimensions.

class sierra.core.graphs.heatmap.**DualHeatmap**(*ipaths: List[pathlib.Path]*, *output_fpath: pathlib.Path*, *title: str*, *xlabel: Optional[str] = None*, *ylabel: Optional[str] = None*, *zlabel: Optional[str] = None*, *large_text: bool = False*, *xtick_labels: Optional[List[str]] = None*, *ytick_labels: Optional[List[str]] = None*, *legend: Optional[List[str]] = None*)

Generates a side-by-side plot of two heataps from two CSV files.

`.mean` files must be named as `<input_stem_fpath>_X.mean`, where *X* is non-negative integer. Input `.mean` files must be 2D grids of the same cardinality.

This graph does not plot standard deviation.

If there are not exactly two file paths passed, the graph is not generated.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.graphs.heatmap', '__doc__': 'Generates a side-by-side plot of two heataps from two CSV files.\n\n ``.mean`` files must be named as ``<input_stem_fpath>_X.mean``, where `X` is\n non-negative integer. Input ``.mean`` files must be 2D grids of the same\n cardinality.\n\n This graph does not plot standard deviation.\n\n If there are not exactly two file paths passed, the graph is not generated.\n\n ', 'kCardinality': 2, '__init__': <function DualHeatmap.__init__>, 'generate': <function DualHeatmap.generate>, '_plot_colorbar': <function DualHeatmap._plot_colorbar>, '_plot_ticks': <function DualHeatmap._plot_ticks>, '_plot_labels': <function DualHeatmap._plot_labels>, '__dict__': <attribute '__dict__' of 'DualHeatmap' objects>, '__weakref__': <attribute '__weakref__' of 'DualHeatmap' objects>, '__annotations__': {}})

__doc__ = 'Generates a side-by-side plot of two heataps from two CSV files.\n\n ``.mean`` files must be named as ``<input_stem_fpath>_X.mean``, where `X` is\n non-negative integer. Input ``.mean`` files must be 2D grids of the same\n cardinality.\n\n This graph does not plot standard deviation.\n\n If there are not exactly two file paths passed, the graph is not generated.\n\n '

__init__(*ipaths: List[pathlib.Path]*, *output_fpath: pathlib.Path*, *title: str*, *xlabel: Optional[str] = None*, *ylabel: Optional[str] = None*, *zlabel: Optional[str] = None*, *large_text: bool = False*, *xtick_labels: Optional[List[str]] = None*, *ytick_labels: Optional[List[str]] = None*, *legend: Optional[List[str]] = None*) → None

__module__ = 'sierra.core.graphs.heatmap'

__weakref__
list of weak references to the object (if defined)

_plot_colorbar(*fig*, *im*, *ax*, *remove: bool*) → None
Plot the Z-axis color bar on the dual heatmap.

_plot_labels(*ax*, *xlabel: bool*, *ylabel: bool*) → None
Plot X,Y axis labels.

_plot_ticks(*ax*, *xvals*, *yvals*, *xlabels: bool*, *ylabels: bool*) → None
Plot ticks and tick labels.

If the labels are numerical and the numbers are too large, force scientific notation (the `rcParam` way of doing this does not seem to work...)

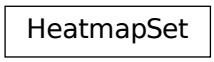
generate() → None

kCardinality = 2

class sierra.core.graphs.heatmap.**HeatmapSet**(*ipaths: List[pathlib.Path]*, *opaths: List[pathlib.Path]*, *titles: List[str]*, ***kwargs*)

Generates a *Heatmap* plot for each of the specified I/O path pairs.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.graphs.heatmap', '__doc__': '\n Generates a :class:`Heatmap` plot for each of the specified I/O path pairs.\n ', '__init__': <function HeatmapSet.__init__>, 'generate': <function HeatmapSet.generate>, '__dict__': <attribute '__dict__' of 'HeatmapSet' objects>, '__weakref__': <attribute '__weakref__' of 'HeatmapSet' objects>, '__annotations__': {}})

__doc__ = '\n Generates a :class:`Heatmap` plot for each of the specified I/O path pairs.\n '

__init__(*ipaths: List[pathlib.Path]*, *opaths: List[pathlib.Path]*, *titles: List[str]*, ***kwargs*) → None

__module__ = 'sierra.core.graphs.heatmap'

__weakref__
list of weak references to the object (if defined)

generate() → None

sierra.core.graphs.scatterplot2D

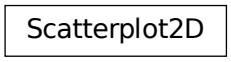
- *Scatterplot2D*: Generates a 2D scatterplot of rows vs. colums (X vs. Y) from a CSV.

class sierra.core.graphs.scatterplot2D.**Scatterplot2D**(*input_fpath: pathlib.Path*, *output_fpath: pathlib.Path*, *title: str*, *xlabel: str*, *ylabel: str*, *xcol: str*, *ycol: str*, *large_text: bool = False*, *regression: bool = False*)

Generates a 2D scatterplot of rows vs. colums (X vs. Y) from a CSV.

If the necessary CSV file does not exist, the graph is not generated.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.graphs.scatterplot2D', '__doc__': 'Generates a 2D scatterplot of rows vs. colums (X vs. Y) from a CSV.\n\n If the necessary CSV file does not exist, the graph is not generated.\n\n ', '__init__': <function Scatterplot2D.__init__>, 'generate': <function Scatterplot2D.generate>, '_plot_regression': <function Scatterplot2D._plot_regression>, '__dict__': <attribute '__dict__' of 'Scatterplot2D' objects>, '__weakref__': <attribute '__weakref__' of 'Scatterplot2D' objects>, '__annotations__': {}})

__doc__ = 'Generates a 2D scatterplot of rows vs. colums (X vs. Y) from a CSV.\n\n If the necessary CSV file does not exist, the graph is not generated.\n\n '

__init__(*input_fpath: pathlib.Path*, *output_fpath: pathlib.Path*, *title: str*, *xlabel: str*, *ylabel: str*, *xcol: str*, *ycol: str*, *large_text: bool = False*, *regression: bool = False*) → None

__module__ = 'sierra.core.graphs.scatterplot2D'

__weakref__
list of weak references to the object (if defined)

_plot_regression(*df*)

generate() → None

sierra.core.graphs.stacked_line_graph

- *StackedLineGraph*: Generates a line graph from a set of columns in a CSV file.

class sierra.core.graphs.stacked_line_graph.**StackedLineGraph**(*stats_root: pathlib.Path*, *input_stem: str*, *output_fpath: pathlib.Path*, *title: str*, *xlabel: Optional[str] = None*, *ylabel: Optional[str] = None*, *large_text: bool = False*, *legend: Optional[List[str]] = None*, *cols: Optional[List[str]] = None*, *linestyles: Optional[List[str]] = None*, *dashstyles: Optional[List[str]] = None*, *logyscale: bool = False*, *stddev_fpath: Optional[pathlib.Path] = None*, *stats: str = 'none'*, *model_root: Optional[pathlib.Path] = None*)

Generates a line graph from a set of columns in a CSV file.

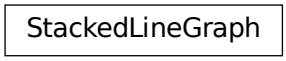
If the necessary data file does not exist, the graph is not generated.

If the .stddev file that goes with the .mean does not exist, then no error bars are plotted.

If the .model file that goes with the .mean does not exist, then no model predictions are plotted.

Ideally, model predictions/stddev calculations would be in derived classes, but I can't figure out a good way to easily pull that stuff out of here.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.graphs.stacked_line_graph', '__doc__': "Generates a line graph from a set of columns in a CSV file.\n\n If the necessary data file does not exist, the graph is not generated.\n\n If the .stddev file that goes with the .mean does not exist, then no error\n bars are plotted.\n\n If the .model file that goes with the .mean does not exist, then no model\n predictions are plotted.\n\n Ideally, model predictions/stddev calculations would be in derived classes,\n but I can't figure out a good way to easily pull that stuff out of here.\n\n ", '__init__': <function StackedLineGraph.__init__>, 'generate': <function StackedLineGraph.generate>, '_plot_ticks': <function StackedLineGraph._plot_ticks>, '_plot_selected_cols': <function StackedLineGraph._plot_selected_cols>, '_plot_col_errorbars': <function StackedLineGraph._plot_col_errorbars>, '_plot_legend': <function StackedLineGraph._plot_legend>, '_read_stats': <function StackedLineGraph._read_stats>, '_read_models': <function StackedLineGraph._read_models>, '__dict__': <attribute '__dict__' of 'StackedLineGraph' objects>, '__weakref__': <attribute '__weakref__' of 'StackedLineGraph' objects>, '__annotations__': {}})

__doc__ = "Generates a line graph from a set of columns in a CSV file.\n\n If the necessary data file does not exist, the graph is not generated.\n\n If the .stddev file that goes with the .mean does not exist, then no error\n bars are plotted.\n\n If the .model file that goes with the .mean does not exist, then no model\n predictions are plotted.\n\n Ideally, model predictions/stddev calculations would be in derived classes,\n but I can't figure out a good way to easily pull that stuff out of here.\n\n "

__init__(*stats_root: pathlib.Path*, *input_stem: str*, *output_fpath: pathlib.Path*, *title: str*, *xlabel: Optional[str] = None*, *ylabel: Optional[str] = None*, *large_text: bool = False*, *legend: Optional[List[str]] = None*, *cols: Optional[List[str]] = None*, *linestyles: Optional[List[str]] = None*, *dashstyles: Optional[List[str]] = None*, *logyscale: bool = False*, *stddev_fpath: Optional[pathlib.Path] = None*, *stats: str = 'none'*, *model_root: Optional[pathlib.Path] = None*) → None

__module__ = 'sierra.core.graphs.stacked_line_graph'

__weakref__
list of weak references to the object (if defined)

_plot_col_errorbars(*data_df: pandas.core.frame.DataFrame*, *stddev_df: pandas.core.frame.DataFrame*, *col: str*) → None

Plot the errorbars for a specific column in a dataframe.

_plot_legend(*ax*, *model_legend: List[str]*, *ncols: int*) → None

_plot_selected_cols(*data_df: pandas.core.frame.DataFrame*, *stat_dfs: Dict[str, pandas.core.frame.DataFrame]*, *cols: List[str]*, *model: Tuple[pandas.core.frame.DataFrame, List[str]]*)

Plot the selected columns in a dataframe.

This may include:

- Errorbars
- Models
- Custom line styles
- Custom dash styles

_plot_ticks(*ax*) → None

_read_models() → Tuple[pandas.core.frame.DataFrame, List[str]]

_read_stats() → Dict[str, pandas.core.frame.DataFrame]

generate() → None

sierra.core.graphs.stacked_surface_graph

- *StackedSurfaceGraph*: Generates a plot of a set of 3D surface graphs from a set of CSVs.

class sierra.core.graphs.stacked_surface_graph.**StackedSurfaceGraph**(*ipaths: List[pathlib.Path]*, *output_fpath: pathlib.Path*, *title: str*, *legend: List[str]*, *xlabel: str*, *ylabel: str*, *zlabel: str*, *xtick_labels: List[str]*, *ytick_labels: List[str]*, *comp_type: str*, *large_text: bool = False*)

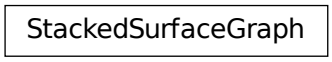
Generates a plot of a set of 3D surface graphs from a set of CSVs.

`.mean` files must be named as``<input_stem_fpath>_X.mean``, where *X* is non-negative integer. Input CSV files must be 2D grids of the same cardinality.

This graph does not plot standard deviation.

If no `.mean` files matching the pattern are found, the graph is not generated.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.graphs.stacked_surface_graph', '__doc__': 'Generates a plot of a set of 3D surface graphs from a set of CSVs.\n\n ``.mean`` files must be named as``<input_stem_fpath>_X.mean``, where `X` is\n non-negative integer. Input CSV files must be 2D grids of the same\n cardinality.\n\n This graph does not plot standard deviation.\n\n If no ``.mean`` files matching the pattern are found, the graph is not\n generated.\n\n ', 'kMaxSurfaces': 4, '__init__': <function StackedSurfaceGraph.__init__>, 'generate': <function StackedSurfaceGraph.generate>, '_plot_surfaces': <function StackedSurfaceGraph._plot_surfaces>, '_plot_ticks': <function StackedSurfaceGraph._plot_ticks>, '_plot_legend': <function StackedSurfaceGraph._plot_legend>, '_plot_labels': <function StackedSurfaceGraph._plot_labels>, '_save_figs': <function StackedSurfaceGraph._save_figs>, '__dict__': <attribute '__dict__' of 'StackedSurfaceGraph' objects>, '__weakref__': <attribute '__weakref__' of 'StackedSurfaceGraph' objects>, '__annotations__': {}})

__doc__ = 'Generates a plot of a set of 3D surface graphs from a set of CSVs.\n\n ``.mean`` files must be named as``<input_stem_fpath>_X.mean``, where `X` is\n non-negative integer. Input CSV files must be 2D grids of the same\n cardinality.\n\n This graph does not plot standard deviation.\n\n If no ``.mean`` files matching the pattern are found, the graph is not\n generated.\n\n '

__init__(*ipaths: List[pathlib.Path]*, *output_fpath: pathlib.Path*, *title: str*, *legend: List[str]*, *xlabel: str*, *ylabel: str*, *zlabel: str*, *xtick_labels: List[str]*, *ytick_labels: List[str]*, *comp_type: str*, *large_text: bool = False*) → None

__module__ = 'sierra.core.graphs.stacked_surface_graph'

__weakref__
list of weak references to the object (if defined)

_plot_labels(*ax*)

_plot_legend(*ax*, *cmap_handles*, *handler_map*)

_plot_surfaces(*X*, *Y*, *ax*, *colors*, *dfs*)

_plot_ticks(*ax*, *xvals*, *yvals*)
Plot ticks and tick labels.

If the labels are numerical and the numbers are too large, force scientific notation (the `rcParam` way of doing this does not seem to work...)

_save_figs(*fig*, *ax*)
Save multiple rotated copies of the same figure.

Necessary for automation of 3D figure generation, because you can't really know a priori what views are going to give the best results. MPL doesn't have a true 3D plot generator yet.

generate() → None

kMaxSurfaces = 4

sierra.core.graphs.summary_line_graph

Linegraph for summarizing the results of a batch experiment in different ways.

- *SummaryLineGraph*: Generates a linegraph from a *Summary .csv*.

class sierra.core.graphs.summary_line_graph.**SummaryLineGraph**(*stats_root: pathlib.Path*, *input_stem: str*, *output_fpath: pathlib.Path*, *title: str*, *xlabel: str*, *ylabel: str*, *xticks: List[float]*, *xtick_labels: Optional[List[str]] = None*, *large_text: bool = False*, *legend: List[str] = ['Empirical Data']*, *logyscale: bool = False*, *stats: str = 'none'*, *model_root: Optional[pathlib.Path] = None*)

Generates a linegraph from a *Summary .csv*.

Possibly shows the 95% confidence interval or box and whisker plots, according to configuration.

stats_root
The absolute path to the `statistics/` directory for the batch experiment.

input_stem
Stem of the *Summary .csv* file to generate a graph from.

output_fpath
The absolute path to the output image file to save generated graph to.

title
Graph title.

xlabel
X-label for graph.

ylabel
Y-label for graph.

xticks
The xticks for the graph.

xtick_labels
The xtick labels for the graph (can be different than the xticks; e.g., if the xticxs are 1-10 for categorical data, then then labels would be the categories).

large_text
Should the labels, ticks, and titles be large, or regular size?

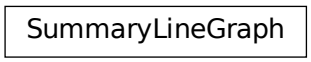
legend
Legend for graph.

logyscale
Should the Y axis be in the log2 domain ?

stats
The type of statistics to include on the graph (from `--dist-stats`).

model_root
The absolute path to the `models/` directory for the batch experiment.

Inheritance




```
__dict__ = mappingproxy({'__module__': 'sierra.core.graphs.summary_line_graph',
'__doc__': 'Generates a linegraph from a :term:`Summary .csv`.\n\n Possibly shows
the 95% confidence interval or box and whisker plots,\n according to
configuration.\n\n Attributes:\n\n stats_root: The absolute path to the
``statistics/`` directory for the\n batch experiment.\n\n input_stem: Stem of the
:term:`Summary .csv` file to generate a graph\n from.\n\n output_fpath: The
absolute path to the output image file to save\n generated graph to.\n\n title:
Graph title.\n\n xlabel: X-label for graph.\n\n ylabel: Y-label for graph.\n\n
xticks: The xticks for the graph.\n\n xtick_labels: The xtick labels for the graph
(can be different than the\n xticks; e.g., if the xticxs are 1-10 for categorical
data,\n then then labels would be the categories).\n\n large_text: Should the
labels, ticks, and titles be large, or regular\n size?\n\n legend: Legend for
graph.\n\n logyscale: Should the Y axis be in the log2 domain ?\n\n stats: The
type of statistics to include on the graph (from\n ``--dist-stats``).\n\n
model_root: The absolute path to the ``models/`` directory for the batch\n
experiment.\n\n ', 'kLineStyles': ['-', '--', '.-', ':', '-', '--', '.-', ':'],
'kMarkStyles': ['o', '^', 's', 'x', 'o', '^', 's', 'x'], '__init__': <function
SummaryLineGraph.__init__>, 'generate': <function SummaryLineGraph.generate>,
'_plot_lines': <function SummaryLineGraph._plot_lines>, '_plot_stats': <function
SummaryLineGraph._plot_stats>, '_plot_conf95_stats': <function
SummaryLineGraph._plot_conf95_stats>, '_plot_bw_stats': <function
SummaryLineGraph._plot_bw_stats>, '_plot_ticks': <function
SummaryLineGraph._plot_ticks>, '_plot_legend': <function
SummaryLineGraph._plot_legend>, '_read_stats': <function
SummaryLineGraph._read_stats>, '_read_conf95_stats': <function
SummaryLineGraph._read_conf95_stats>, '_read_bw_stats': <function
SummaryLineGraph._read_bw_stats>, '_read_models': <function
SummaryLineGraph._read_models>, '__dict__': <attribute '__dict__' of
'SummaryLineGraph' objects>, '__weakref__': <attribute '__weakref__' of
'SummaryLineGraph' objects>, '__annotations__': {}})
```

__doc__ = 'Generates a linegraph from a :term:`Summary .csv`.\n\n Possibly shows the 95% confidence interval or box and whisker plots,\n according to configuration.\n\n Attributes:\n\n stats_root: The absolute path to the ``statistics/`` directory for the\n batch experiment.\n\n input_stem: Stem of the :term:`Summary .csv` file to generate a graph\n from.\n\n output_fpath: The absolute path to the output image file to save\n generated graph to.\n\n title: Graph title.\n\n xlabel: X-label for graph.\n\n ylabel: Y-label for graph.\n\n xticks: The xticks for the graph.\n\n xtick_labels: The xtick labels for the graph (can be different than the\n xticks; e.g., if the xticxs are 1-10 for categorical data,\n then then labels would be the categories).\n\n large_text: Should the labels, ticks, and titles be large, or regular\n size?\n\n legend: Legend for graph.\n\n logyscale: Should the Y axis be in the log2 domain ?\n\n stats: The type of statistics to include on the graph (from\n ``--dist-stats``).\n\n model_root: The absolute path to the ``models/`` directory for the batch\n experiment.\n\n '

__init__(*stats_root: pathlib.Path*, *input_stem: str*, *output_fpath: pathlib.Path*, *title: str*, *xlabel: str*, *ylabel: str*, *xticks: List[float]*, *xtick_labels: Optional[List[str]] = None*, *large_text: bool = False*, *legend: List[str] = ['Empirical Data']*, *logyscale: bool = False*, *stats: str = 'none'*, *model_root: Optional[pathlib.Path] = None*) → None

__module__ = 'sierra.core.graphs.summary_line_graph'

__weakref__
list of weak references to the object (if defined)

_plot_bw_stats(*ax*, *xticks*, *data_dfy: pandas.core.frame.DataFrame*, *stat_dfs: Dict[str, pandas.core.frame.DataFrame]*) → None

_plot_conf95_stats(*xticks*, *data_dfy: pandas.core.frame.DataFrame*, *stat_dfs: Dict[str, pandas.core.frame.DataFrame]*) → None

_plot_legend(*model: Tuple[pandas.core.frame.DataFrame, List[str]]*) → None

_plot_lines(*data_dfy: pandas.core.frame.DataFrame*, *model: Tuple[pandas.core.frame.DataFrame, List[str]]*) → None

_plot_stats(*ax*, *xticks*, *data_dfy: pandas.core.frame.DataFrame*, *stat_dfs: Dict[str, pandas.core.frame.DataFrame]*) → None
Plot statistics for all lines on the graph.

_plot_ticks(*ax*) → None

_read_bw_stats() → Dict[str, List[pandas.core.frame.DataFrame]]

_read_conf95_stats() → Dict[str, List[pandas.core.frame.DataFrame]]

_read_models() → Tuple[pandas.core.frame.DataFrame, List[str]]

_read_stats() → Dict[str, List[pandas.core.frame.DataFrame]]

generate() → None

kLineStyles = ['-', '--', '.-', ':', '-', '--', '.-', ':']

kMarkStyles = ['o', '^', 's', 'x', 'o', '^', 's', 'x']

sierra.core.hpc

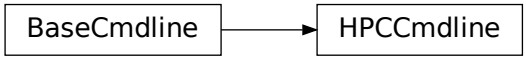
sierra.core.hpc.cmdline

Common cmdline classes for the various HPC plugins.

- *HPCCmdline*: The base cmdline definition class for SIERRA for reusability.

class sierra.core.hpc.cmdline.**HPCCmdline**(*stages: List[int]*)

Inheritance



__doc__ = None

__init__(*stages: List[int]*) → None

__module__ = 'sierra.core.hpc.cmdline'

static cmdopts_update(*cli_args: argparse.Namespace*, *cmdopts: Dict[str, Any]*) → None
Update cmdopts dictionary with the HPC-specific cmdline options.

init_cli(*stages: List[int]*) → None

init_stage2() → None
Add HPC cmdline options.

Options may be interpreted differently between *Platforms*, or ignored, depending. These include:

- `--exec-jobs-per-node`
- `--exec-no-devnull`
- `--exec-resume`
- `--exec-strict`

scaffold_cli() → None

sierra.core.logging

Extensions to the standard python `logging` module for SIERRA.

These include:

- Supporting the additional `TRACE` level. No idea why this is not supported by default...
- Adding nice colored logging which is helpful but not angry fruit salad.

sierra.core.models

sierra.core.models.graphs

Graphs which can always be generated, irrespective of model specifics.

For example, you can always compare the model value to the empirical value, and plot the difference as error.

sierra.core.models.interface

Interface classes for the mathematical models framework in SIERRA.

Models can be run and added to any configured graph during stage 4.

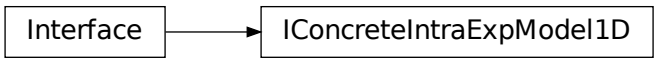
- *IConcreteIntraExpModel1D*: Interface for one-dimensional models.
- *IConcreteIntraExpModel2D*: Interface for two-dimensional models.
- *IConcreteInterExpModel1D*: Interface for one-dimensional models.

class sierra.core.models.interface.**IConcreteIntraExpModel1D**

Interface for one-dimensional models.

1D models are those that generate a single time series from zero or more experimental outputs within a *single Experiment*. Models will be rendered as lines on a *StackedLineGraph*. Models "target" one or more CSV files which are already configured to be generated, and will show up as additional lines on the generated graph.

Inheritance



__doc__ = 'Interface for one-dimensional models.\n\n 1D models are those that generate a single time series from zero or more\n experimental outputs within a *single* :term:`Experiment`. Models will be\n rendered as lines on a\n :class:`~sierra.core.graphs.stacked_line_graph.StackedLineGraph`. Models\n "target" one or more CSV files which are already configured to be generated,\n and will show up as additional lines on the generated graph.\n\n '

__module__ = 'sierra.core.models.interface'

__repr__() → str

Return the UUID string of the model name.

legend_names() → List[str]

Compute names for the model predictions for the target graph legend.

Applicable to:

- *StackedLineGraph*

run(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria, *exp_num: int*, *cmdopts: Dict[str, Any]*) → List[pandas.core.frame.DataFrame]
Run the model and generate a list of dataframes.

Each dataframe can (potentially) target different graphs. All dataframes should contain a single column named `model`, with each row of the dataframe containing the model prediction at the *Experimental Run* interval corresponding to the row (e.g., row 7 contains the model prediction for interval 7).

run_for_exp(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria, *cmdopts: Dict[str, Any]*, *exp_num: int*) → bool
Determine if the model should be run for the specified experiment.

Some models may only be valid/make sense to run for a subset of experiments within a batch, so models can be selectively executed with this function.

target_csv_stems() → List[str]
Return a list of CSV file stems that the model is targeting.

File stem = path sans directory path and extension.

class sierra.core.models.interface.**IConcreteIntraExpModel2D**
Interface for two-dimensional models.

2D models are those that generate a list of 2D matrices, forming a 2D time series. Can be built from zero or more experimental outputs from a *single Experiment*. Models "target" one or more CSV files which are already configured to be generated, and will show up as additional lines on the generated graph.

Inheritance

__doc__ = 'Interface for two-dimensional models.\n\n 2D models are those that generate a list of 2D matrices, forming a 2D time\n series. Can be built from zero or more experimental outputs from a *single*\n :term:`Experiment`. Models "target" one or more CSV files which are already\n configured to be generated, and will show up as additional lines on the\n generated graph.\n\n '

__module__ = 'sierra.core.models.interface'

__repr__() → str
Return the UUID string of the model name.

run(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria, *exp_num: int*, *cmdopts: Dict[str, Any]*) → List[pandas.core.frame.DataFrame]
Run the model and generate a list of dataframes.

Each dataframe can (potentially) target a different graph. Each dataframe should be a NxM grid (with N not necessarily equal to M). All dataframes do not have to be the same dimensions. The index of a given dataframe in a list should correspond to the model value for interval/timestep.

run_for_exp(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria, *cmdopts: Dict[str, Any]*, *exp_num: int*) → bool
Determine if a model should be run for the specified experiment.

Some models may only be valid/make sense to run for a subset of experiments within a batch, so models can be selectively executed with this function.

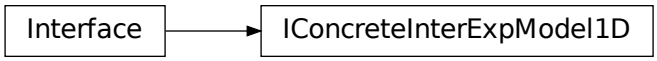
target_csv_stems() → List[str]
Return a list of CSV file stems that the model is targeting.

File stem = path sans directory path and extension.

class sierra.core.models.interface.**IConcreteInterExpModel1D**
Interface for one-dimensional models.

1D models are those that generate a single time series from any number of experimental outputs across *all* experiments in a batch(or from another source). Models will be rendered as lines on a *SummaryLineGraph*. Models "target" one or more CSV files which are already configured to be generated, and will show up as additional lines on the generated graph.

Inheritance



__doc__ = 'Interface for one-dimensional models.\n\n 1D models are those that generate a single time series from any number of\n experimental outputs across *all* experiments in a batch(or from another\n source). Models will be rendered as lines on a\n :class:`~sierra.core.graphs.summary_line_graph.SummaryLineGraph`. Models\n "target" one or more CSV files which are already configured to be generated,\n and will show up as additional lines on the generated graph.\n\n '

__module__ = 'sierra.core.models.interface'

__repr__() → str
Return the UUID string of the model name.

legend_names() → List[str]
Compute names for the model predictions for the target graph legend.

Applicable to:

- ~sierra.core.graphs.summary_line_graph.SummaryLineGraph`

run(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria, *cmdopts: Dict[str, Any]*) → List[pandas.core.frame.DataFrame]
Run the model and generate list of dataframes.

Each dataframe can (potentially) target a different graph. Each dataframe should contain a single row, with one column for the predicted value of the model for each experiment in the batch.

run_for_batch(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria, *cmdopts: Dict[str, Any]*) → bool
Determine if the model should be run for the specified batch criteria.

Some models may only be valid/make sense to run for some batch criteria, so models can be selectively executed with this function.

target_csv_stems() → List[str]
Return a list of CSV file stems that the model is targeting.

File stem = path sans directory path and extension.

sierra.core.pipeline

sierra.core.pipeline.pipeline

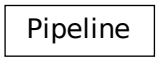
The 5 pipeline stages implemented by SIERRA.

See *SIERRA Pipeline* for high-level documentation.

- *Pipeline*: Implements SIERRA's 5 stage pipeline.

class sierra.core.pipeline.pipeline.**Pipeline**(*args: argparse.Namespace*, *controller: Optional[str]*, *rdg_opts: Dict[str, Any]*)
Implements SIERRA's 5 stage pipeline.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.pipeline', '__doc__': "Implements SIERRA's 5 stage pipeline.", '__init__': <function Pipeline.__init__>, 'run': <function Pipeline.run>, '_load_config': <function Pipeline._load_config>, '__dict__': <attribute '__dict__' of 'Pipeline' objects>, '__weakref__': <attribute '__weakref__' of 'Pipeline' objects>, '__annotations__': {}})

__doc__ = "Implements SIERRA's 5 stage pipeline."

__init__(*args: argparse.Namespace*, *controller: Optional[str]*, *rdg_opts: Dict[str, Any]*) → None

__module__ = 'sierra.core.pipeline.pipeline'

__weakref__
list of weak references to the object (if defined)

_load_config() → None

run() → None
Run pipeline stages 1-5 as configured.

sierra.core.pipeline.stage1

sierra.core.pipeline.stage1.pipeline_stage1

Stage 1 of the experimental pipeline: generating experimental inputs.

- *PipelineStage1*: Generates a *Batch Experiment* for running during stage 2.

class sierra.core.pipeline.stage1.pipeline_stage1.**PipelineStage1**(*cmdopts: Dict[str, Any]*, *controller: str*, *criteria:* sierra.core.variables.batch_criteria.IConcreteBatch

Generates a *Batch Experiment* for running during stage 2.

Generated experimental input files are written to the filesystem, and can be be used in stage 2 to launch simulations/real robot controller. This stage is idempotent with default settings; this can be overridden with `--no-preserve-seeds`, in which case this stage is no longer idempotent.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage1.pipeline_stage1', '__doc__': 'Generates a :term:`Batch Experiment` for running during stage 2.\n\n Generated experimental input files are written to the filesystem, and can be\n be used in stage 2 to launch simulations/real robot controller. This stage\n is idempotent with default settings; this can be overridden with\n ``--no-preserve-seeds``, in which case this stage is no longer idempotent.\n\n ', '__init__': <function PipelineStage1.__init__>, 'run': <function PipelineStage1.run>, '__dict__': <attribute '__dict__' of 'PipelineStage1' objects>, '__weakref__': <attribute '__weakref__' of 'PipelineStage1' objects>, '__annotations__': {}})

__doc__ = 'Generates a :term:`Batch Experiment` for running during stage 2.\n\n Generated experimental input files are written to the filesystem, and can be\n be used in stage 2 to launch simulations/real robot controller. This stage\n is idempotent with default settings; this can be overridden with\n ``--no-preserve-seeds``, in which case this stage is no longer idempotent.\n\n '

__init__(*cmdopts: Dict[str, Any]*, *controller: str*, *criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None

__module__ = 'sierra.core.pipeline.stage1.pipeline_stage1'

__weakref__
list of weak references to the object (if defined)

run() → None
Create the *Batch Experiment* in the filesystem.

The creation process can be summarized as:

1. Scaffold the batch experiment by applying sets of changes from the batch criteria to the XML template input file for each experiment in the batch. These changes apply to all *Experiments* in the batch and all *Experimental Runs* in the experiment.
2. Generate changes to be applied to to the newly written per-experiment "template" XML file for each experiment which are non-*Batch Criteria* related (e.g., *Platform* changes). These changes apply to all experiment runs in an experiment, but may differ across experimental runs (e.g., # cores used for a simulator platform).
3. Generate per-experimental run changes for each experimental run in the experiment, according to platform and *Project* configuration.
4. Write the input files for all experimental runs in all experiments to the filesystem after generation.

sierra.core.pipeline.stage2

sierra.core.pipeline.stage2.exp_runner

Classes for executing experiments via the specified `--exec-env`.

- `BatchExpRunner`: Runs each *Experiment* in *Batch Experiment* in sequence.
- `ExpRunner`: Execute each *Experimental Run* in an *Experiment*.
- `ExpShell`: Launch a shell which persists across experimental runs.

class sierra.core.pipeline.stage2.exp_runner.**BatchExpRunner**(*cmdopts: Dict[str, Any]*, *criteria:* sierra.core.variables.batch_criteria.BatchCriteria)

Runs each *Experiment* in *Batch Experiment* in sequence.

batch_exp_root
Absolute path to the root directory for the batch experiment inputs (i.e. experiment directories are placed in here).

batch_stat_root
Absolute path to the root directory for statistics which are computed in stage {3,4} (i.e. experiment directories are placed in here).

batch_stat_exec_root
Absolute path to the root directory for statistics which are generated as experiments run during stage 2 (e.g., how long each experiment took).

cmdopts
Dictionary of parsed cmdline options.

criteria
Batch criteria for the experiment.

exec_exp_range
The subset of experiments in the batch to run (can be None to run all experiments in the batch).

Inheritance

__call__() → None
Execute experiments in the batch according to configuration.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage2.exp_runner', '__doc__': 'Runs each :term:`Experiment` in :term:`Batch Experiment` in sequence.\n\n Attributes:\n\n batch_exp_root: Absolute path to the root directory for the batch\n experiment inputs (i.e. experiment directories are\n placed in here).\n\n batch_stat_root: Absolute path to the root directory for statistics\n which are computed in stage {3,4} (i.e. experiment\n directories are placed in here).\n\n batch_stat_exec_root: Absolute path to the root directory for statistics\n which are generated as experiments run during\n stage 2 (e.g., how long each experiment took).\n\n cmdopts: Dictionary of parsed cmdline options.\n\n criteria: Batch criteria for the experiment.\n\n exec_exp_range: The subset of experiments in the batch to run (can be\n None to run all experiments in the batch).\n\n ', '__init__': <function BatchExpRunner.__init__>, '__call__': <function BatchExpRunner.__call__>, '__dict__': <attribute '__dict__' of 'BatchExpRunner' objects>, '__weakref__': <attribute '__weakref__' of 'BatchExpRunner' objects>, '__annotations__': {}})

__doc__ = 'Runs each :term:`Experiment` in :term:`Batch Experiment` in sequence.\n\n Attributes:\n\n batch_exp_root: Absolute path to the root directory for the batch\n experiment inputs (i.e. experiment directories are\n placed in here).\n\n batch_stat_root: Absolute path to the root directory for statistics\n which are computed in stage {3,4} (i.e. experiment\n directories are placed in here).\n\n batch_stat_exec_root: Absolute path to the root directory for statistics\n which are generated as experiments run during\n stage 2 (e.g., how long each experiment took).\n\n cmdopts: Dictionary of parsed cmdline options.\n\n criteria: Batch criteria for the experiment.\n\n exec_exp_range: The subset of experiments in the batch to run (can be\n None to run all experiments in the batch).\n\n '

__init__(*cmdopts: Dict[str, Any]*, *criteria:* sierra.core.variables.batch_criteria.BatchCriteria) → None

__module__ = 'sierra.core.pipeline.stage2.exp_runner'

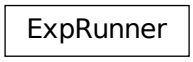
__weakref__
list of weak references to the object (if defined)

class sierra.core.pipeline.stage2.exp_runner.**ExpRunner**(*cmdopts: Dict[str, Any]*, *exec_times_fpath: pathlib.Path*, *generator:* sierra.core.platform.ExpShellCmdsGenerator, *shell:* sierra.core.pipeline.stage2.exp_runner.ExpShell)

Execute each *Experimental Run* in an *Experiment*.

In parallel if the selected execution environment supports it, otherwise sequentially.

Inheritance



__call__(*exp_input_root: pathlib.Path*, *exp_num: int*) → None
Execute experimental runs for a single experiment.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage2.exp_runner', '__doc__': '\n Execute each :term:`Experimental Run` in an :term:`Experiment`.\n\n In parallel if the selected execution environment supports it, otherwise\n sequentially.\n\n ', '__init__': <function ExpRunner.__init__>, '__call__': <function ExpRunner.__call__>, '__dict__': <attribute '__dict__' of 'ExpRunner' objects>, '__weakref__': <attribute '__weakref__' of 'ExpRunner' objects>, '__annotations__': {}})

__doc__ = '\n Execute each :term:`Experimental Run` in an :term:`Experiment`.\n\n In parallel if the selected execution environment supports it, otherwise\n sequentially.\n\n '

__init__(*cmdopts: Dict[str, Any]*, *exec_times_fpath: pathlib.Path*, *generator:* sierra.core.platform.ExpShellCmdsGenerator, *shell:* sierra.core.pipeline.stage2.exp_runner.ExpShell) → None

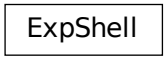
__module__ = 'sierra.core.pipeline.stage2.exp_runner'

__weakref__
list of weak references to the object (if defined)

class sierra.core.pipeline.stage2.exp_runner.**ExpShell**(*exec_strict: bool*)
Launch a shell which persists across experimental runs.

Having a persistent shell is necessary so that running pre- and post-run shell commands have an effect on the actual commands to execute the run. If you set an environment variable before the simulator launches (for example), and then the shell containing that change exits, and the simulator launches in a new shell, then the configuration has no effect. Thus, a persistent shell.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage2.exp_runner', '__doc__': 'Launch a shell which persists across experimental runs.\n\n Having a persistent shell is necessary so that running pre- and post-run\n shell commands have an effect on the actual commands to execute the run. If\n you set an environment variable before the simulator launches (for example),\n and then the shell containing that change exits, and the simulator launches\n in a new shell, then the configuration has no effect. Thus, a persistent\n shell.\n\n ', '__init__': <function ExpShell.__init__>, 'run_from_spec': <function ExpShell.run_from_spec>, '_update_env': <function ExpShell._update_env>, '__dict__': <attribute '__dict__' of 'ExpShell' objects>, '__weakref__': <attribute '__weakref__' of 'ExpShell' objects>, '__annotations__': {'procs': 'tp.List[subprocess.Popen]'}})

__doc__ = 'Launch a shell which persists across experimental runs.\n\n Having a persistent shell is necessary so that running pre- and post-run\n shell commands have an effect on the actual commands to execute the run. If\n you set an environment variable before the simulator launches (for example),\n and then the shell containing that change exits, and the simulator launches\n in a new shell, then the configuration has no effect. Thus, a persistent\n shell.\n\n '

__init__(*exec_strict: bool*) → None

__module__ = 'sierra.core.pipeline.stage2.exp_runner'

__weakref__
: list of weak references to the object (if defined)

_update_env(*stdout*) → None

run_from_spec(*spec:* sierra.core.types.ShellCmdSpec) → bool

sierra.core.pipeline.stage2.pipeline_stage2

Stage 2 of the experimental pipeline: running experiments.

- *PipelineStage2*: Runs *Experiments* in a *Batch Experiment*.

class sierra.core.pipeline.stage2.pipeline_stage2.**PipelineStage2**(*cmdopts: Dict[str, Any]*)
: Runs *Experiments* in a *Batch Experiment*.

  GNUParallel is used as the execution engine, and a provided/generated set of hosts is used to actually execute experiments in the selected execution environment.

  This stage is *NOT* idempotent, for obvious reasons.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage2.pipeline_stage2', '__doc__': 'Runs :term:`Experiments <Experiment>` in a :term:`Batch Experiment`.\n\n GNUParallel is used as the execution engine, and a provided/generated set of\n hosts is used to actually execute experiments in the selected execution\n environment.\n\n This stage is *NOT* idempotent, for obvious reasons.\n\n ', '__init__': <function PipelineStage2.__init__>, 'run': <function PipelineStage2.run>, '__dict__': <attribute '__dict__' of 'PipelineStage2' objects>, '__weakref__': <attribute '__weakref__' of 'PipelineStage2' objects>, '__annotations__': {}})

__doc__ = 'Runs :term:`Experiments <Experiment>` in a :term:`Batch Experiment`.\n\n GNUParallel is used as the execution engine, and a provided/generated set of\n hosts is used to actually execute experiments in the selected execution\n environment.\n\n This stage is *NOT* idempotent, for obvious reasons.\n\n '

__init__(*cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.core.pipeline.stage2.pipeline_stage2'

__weakref__
list of weak references to the object (if defined)

run(*criteria:* sierra.core.variables.batch_criteria.BatchCriteria) → None

sierra.core.pipeline.stage3

sierra.core.pipeline.stage3.imagizer

Classes for creating image files from `.mean` files for experiments.

See *Rendering* for usage documentation.

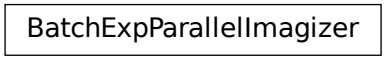
- *BatchExpParallelImagizer*: Generate images for each *Experiment* in the *Batch Experiment*.
- *ExpImagizer*: Create images from the averaged `.mean` files from an experiment.

class sierra.core.pipeline.stage3.imagizer.**BatchExpParallelImagizer**(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*)

Generate images for each *Experiment* in the *Batch Experiment*.

Ideally this is done in parallel across experiments, but this can be changed to serial if memory on the SIERRA host machine is limited via `--processing-serial`.

Inheritance



__call__(*HM_config: Dict[str, Any]*, *criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage3.imagizer', '__doc__': 'Generate images for each :term:`Experiment` in the :term:`Batch Experiment`.\n\n Ideally this is done in parallel across experiments, but this can be changed\n to serial if memory on the SIERRA host machine is limited via\n ``--processing-serial``.\n\n ', '__init__': <function BatchExpParallelImagizer.__init__>, '__call__': <function BatchExpParallelImagizer.__call__>, '_enqueue_for_exp': <function BatchExpParallelImagizer._enqueue_for_exp>, '_thread_worker': <staticmethod object>, '__dict__': <attribute '__dict__' of 'BatchExpParallelImagizer' objects>, '__weakref__': <attribute '__weakref__' of 'BatchExpParallelImagizer' objects>, '__annotations__': {}})

__doc__ = 'Generate images for each :term:`Experiment` in the :term:`Batch Experiment`.\n\n Ideally this is done in parallel across experiments, but this can be changed\n to serial if memory on the SIERRA host machine is limited via\n ``--processing-serial``.\n\n '

__init__(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.core.pipeline.stage3.imagizer'

__weakref__
list of weak references to the object (if defined)

_enqueue_for_exp(*exp_stat_root: pathlib.Path*, *exp_imagize_root: pathlib.Path*, *q: multiprocessing.context.BaseContext.JoinableQueue*) → None

static _thread_worker(*q: multiprocessing.context.BaseContext.Queue*, *HM_config: Dict[str, Any]*) → None

class sierra.core.pipeline.stage3.imagizer.**ExpImagizer**
Create images from the averaged `.mean` files from an experiment.

If no `.mean` files suitable for averaging are found, nothing is done. See *Rendering* for per-platform descriptions of what "suitable" means.

Parameters

- **HM_config** – Parsed YAML configuration for heatmaps.
- **imagize_opts** – Dictionary of imagizing options.

Inheritance

__call__(*HM_config: Dict[str, Any]*, *imagize_opts: dict*) → None
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage3.imagizer', '__doc__': 'Create images from the averaged ``.mean`` files from an experiment.\n\n If no ``.mean`` files suitable for averaging are found, nothing is done. See\n :ref:`ln-sierra-usage-rendering` for per-platform descriptions of what\n "suitable" means.\n\n Arguments:\n\n HM_config: Parsed YAML configuration for heatmaps.\n\n imagize_opts: Dictionary of imagizing options.\n\n ', '__init__': <function ExpImagizer.__init__>, '__call__': <function ExpImagizer.__call__>, '__dict__': <attribute '__dict__' of 'ExpImagizer' objects>, '__weakref__': <attribute '__weakref__' of 'ExpImagizer' objects>, '__annotations__': {}})

__doc__ = 'Create images from the averaged ``.mean`` files from an experiment.\n\n If no ``.mean`` files suitable for averaging are found, nothing is done. See\n :ref:`ln-sierra-usage-rendering` for per-platform descriptions of what\n "suitable" means.\n\n Arguments:\n\n HM_config: Parsed YAML configuration for heatmaps.\n\n imagize_opts: Dictionary of imagizing options.\n\n '

__init__() → None

__module__ = 'sierra.core.pipeline.stage3.imagizer'

__weakref__
list of weak references to the object (if defined)

sierra.core.pipeline.stage3.pipeline_stage3

Stage 3 of the experimental pipeline: processing experimental results.

- *PipelineStage3*: Processes the results of running a *Batch Experiment*.

class sierra.core.pipeline.stage3.pipeline_stage3.**PipelineStage3**(*main_config: dict*, *cmdopts: Dict[str, Any]*)

Processes the results of running a *Batch Experiment*.

Currently this includes:

- Generating statistics from results for generating per-experiment graphs during stage 4. This can generate *Averaged .csv* files, among other statistics.
- Collating results across experiments for generating inter-experiment graphs during stage 4.
- Generating image files from project metric collection for later use in video rendering in stage 4.

This stage is idempotent.

Inheritance

__dict__ = mappingproxy({'__module__':
'sierra.core.pipeline.stage3.pipeline_stage3', '__doc__': 'Processes the results of
running a :term:`Batch Experiment`.\n\n Currently this includes:\n\n - Generating
statistics from results for generating per-experiment graphs\n during stage 4. This
can generate :term:`Averaged .csv` files, among\n other statistics.\n\n - Collating
results across experiments for generating inter-experiment\n graphs during stage
4.\n\n - Generating image files from project metric collection for later use in\n
video rendering in stage 4.\n\n This stage is idempotent.\n\n ', '__init__':
<function PipelineStage3.__init__>, 'run': <function PipelineStage3.run>,
'_run_statistics': <function PipelineStage3._run_statistics>, '_run_run_collation':
<function PipelineStage3._run_run_collation>, '_run_imagizing': <function
PipelineStage3._run_imagizing>, '__dict__': <attribute '__dict__' of
'PipelineStage3' objects>, '__weakref__': <attribute '__weakref__' of
'PipelineStage3' objects>, '__annotations__': {}})

__doc__ = 'Processes the results of running a :term:`Batch Experiment`.\n\n
Currently this includes:\n\n - Generating statistics from results for generating
per-experiment graphs\n during stage 4. This can generate :term:`Averaged .csv`
files, among\n other statistics.\n\n - Collating results across experiments for
generating inter-experiment\n graphs during stage 4.\n\n - Generating image files
from project metric collection for later use in\n video rendering in stage 4.\n\n
This stage is idempotent.\n\n '

__init__(*main_config: dict*, *cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.core.pipeline.stage3.pipeline_stage3'

__weakref__
list of weak references to the object (if defined)

_run_imagizing(*main_config: dict*, *intra_HM_config: dict*, *cmdopts: Dict[str, Any]*, *criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria)

_run_run_collation(*main_config: dict*, *cmdopts: Dict[str, Any]*, *criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria)

_run_statistics(*main_config: dict*, *cmdopts: Dict[str, Any]*, *criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria)

run(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None

sierra.core.pipeline.stage3.run_collator

Classes for collating data within a *Batch Experiment*.

Collation is the process of "lifting" data from *Experimental Runs* across all *Experiment* for all experiments in a *Batch Experiment* into a single CSV (a reduce operation). This is needed to correctly calculate summary statistics for performance measures in stage 4: you can't just run the calculated stddev through the calculations for flexibility (for example) because comparing curves of stddev is not meaningful. Stage 4 needs access to raw-(er) run data to construct a *distribution* of performance measure values to then calculate the summary statistics (such as stddev) over.

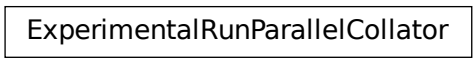
- *ExperimentalRunParallelCollator*: Generates *Collated .csv* files for each *Experiment*.
- *ExperimentalRunCSVGatherer*: Gather *Output .csv* files across all runs within an experiment.
- *ExperimentalRunCollator*: Collate gathered *Output .csv* files together (reduce operation).

class sierra.core.pipeline.stage3.run_collator.**ExperimentalRunParallelCollator**(*main_config: dict*, *cmdopts: Dict[str, Any]*)

Generates *Collated .csv* files for each *Experiment*.

***Collated .csv* files generated from *Output .csv* files across** *Experimental Runs*. Gathered in parallel for each experiment for speed, unless disabled with `--processing-serial`.

Inheritance



__call__(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage3.run_collator', '__doc__': 'Generates :term:`Collated .csv` files for each :term:`Experiment`.\n\n :term:`Collated .csv` files generated from :term:`Output .csv` files across\n :term:`Experimental Runs <Experimental Run>`. Gathered in parallel for\n each experiment for speed, unless disabled with ``--processing-serial``.\n\n ', '__init__': <function ExperimentalRunParallelCollator.__init__>, '__call__': <function ExperimentalRunParallelCollator.__call__>, '_gather_worker': <staticmethod object>, '_process_worker': <staticmethod object>, '__dict__': <attribute '__dict__' of 'ExperimentalRunParallelCollator' objects>, '__weakref__': <attribute '__weakref__' of 'ExperimentalRunParallelCollator' objects>, '__annotations__': {}})

__doc__ = 'Generates :term:`Collated .csv` files for each :term:`Experiment`.\n\n :term:`Collated .csv` files generated from :term:`Output .csv` files across\n :term:`Experimental Runs <Experimental Run>`. Gathered in parallel for\n each experiment for speed, unless disabled with ``--processing-serial``.\n\n '

__init__(*main_config: dict*, *cmdopts: Dict[str, Any]*)

__module__ = 'sierra.core.pipeline.stage3.run_collator'

__weakref__
list of weak references to the object (if defined)

static _gather_worker(*gatherq: multiprocessing.context.BaseContext.Queue*, *processq: multiprocessing.context.BaseContext.Queue*, *main_config: Dict[str, Any]*, *project: str*, *storage_medium: str*) → None

static _process_worker(*processq: multiprocessing.context.BaseContext.Queue*, *main_config: Dict[str, Any]*, *batch_stat_collate_root: pathlib.Path*, *storage_medium: str*, *df_homogenize: str*) → None

class sierra.core.pipeline.stage3.run_collator.**ExperimentalRunCSVGatherer**(*main_config: Dict[str, Any]*, *storage_medium: str*, *processq: multiprocessing.context.BaseContext.Queue*)

Gather *Output .csv* files across all runs within an experiment.

This class can be extended/overriden using a *Project* hook. See *SIERRA Hooks* for details.

processq
The multiprocessing-safe producer-consumer queue that the data gathered from experimental runs will be placed in for processing.

storage_medium
The name of the storage medium plugin to use to extract dataframes from when reading run data.

main_config
Parsed dictionary of main YAML configuration.

logger
The handle to the logger for this class. If you extend this class, you should save/restore this variable in tandem with overriding it in order to get logging messages have unique logger names between this class and your derived class, in order to reduce confusion.

Inheritance

__call__(*batch_output_root: pathlib.Path*, *exp_leaf: str*)
Gather CSV data from all experimental runs in an experiment.

Gathered data is put in a queue for processing.

Parameters **exp_leaf** – The name of the experiment directory within the `batch_output_root`.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage3.run_collator', '__doc__': 'Gather :term:`Output .csv` files across all runs within an experiment.\n\n This class can be extended/overriden using a :term:`Project` hook. See\n :ref:`ln-sierra-tutorials-project-hooks` for details.\n\n Attributes:\n\n processq: The multiprocessing-safe producer-consumer queue that the data\n gathered from experimental runs will be placed in for\n processing.\n\n storage_medium: The name of the storage medium plugin to use to extract\n dataframes from when reading run data.\n\n main_config: Parsed dictionary of main YAML configuration.\n\n logger: The handle to the logger for this class. If you extend this\n class, you should save/restore this variable in tandem with\n overriding it in order to get logging messages have unique\n logger names between this class and your derived class, in order\n to reduce confusion.\n\n ', '__init__': <function ExperimentalRunCSVGatherer.__init__>, '__call__': <function ExperimentalRunCSVGatherer.__call__>, 'gather_csvs_from_run': <function ExperimentalRunCSVGatherer.gather_csvs_from_run>, '__dict__': <attribute '__dict__' of 'ExperimentalRunCSVGatherer' objects>, '__weakref__': <attribute '__weakref__' of 'ExperimentalRunCSVGatherer' objects>, '__annotations__': {}})

__doc__ = 'Gather :term:`Output .csv` files across all runs within an experiment.\n\n This class can be extended/overriden using a :term:`Project` hook. See\n :ref:`ln-sierra-tutorials-project-hooks` for details.\n\n Attributes:\n\n processq: The multiprocessing-safe producer-consumer queue that the data\n gathered from experimental runs will be placed in for\n processing.\n\n storage_medium: The name of the storage medium plugin to use to extract\n dataframes from when reading run data.\n\n main_config: Parsed dictionary of main YAML configuration.\n\n logger: The handle to the logger for this class. If you extend this\n class, you should save/restore this variable in tandem with\n overriding it in order to get logging messages have unique\n logger names between this class and your derived class, in order\n to reduce confusion.\n\n '

__init__(*main_config: Dict[str, Any]*, *storage_medium: str*, *processq: multiprocessing.context.BaseContext.Queue*) → None

__module__ = 'sierra.core.pipeline.stage3.run_collator'

__weakref__
list of weak references to the object (if defined)

gather_csvs_from_run(*run_output_root: pathlib.Path*) → Dict[Tuple[str, str], pandas.core.frame.DataFrame]
Gather all data from a single run within an experiment.

Returns

A dictionary of <(CSV file name, CSV performance column), dataframe> key-value pairs. The CSV file name is the leaf part of the path with the extension included.

Return type dict

class sierra.core.pipeline.stage3.run_collator.**ExperimentalRunCollator**(*main_config: Dict[str, Any]*, *batch_stat_collate_root: pathlib.Path*, *storage_medium: str*, *df_homogenize: str*)

Collate gathered *Output .csv* files together (reduce operation).

Output .csv`s gathered from N :term: `Experimental Runs are combined together into a single *Summary .csv* per *Experiment* with 1 column per run.

Inheritance

__call__(*gathered_runs: List[str]*, *gathered_dfs: List[Dict[Tuple[str, str], pandas.core.frame.DataFrame]]*, *exp_leaf: str*) → None
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage3.run_collator', '__doc__': 'Collate gathered :term:`Output .csv` files together (reduce operation).\n\n :term:`Output .csv`s gathered from N :term:`Experimental Runs <Experimental\n Run>` are combined together into a single :term:`Summary .csv` per\n :term:`Experiment` with 1 column per run.\n\n ', '__init__': <function ExperimentalRunCollator.__init__>, '__call__': <function ExperimentalRunCollator.__call__>, '__dict__': <attribute '__dict__' of 'ExperimentalRunCollator' objects>, '__weakref__': <attribute '__weakref__' of 'ExperimentalRunCollator' objects>, '__annotations__': {}})

__doc__ = 'Collate gathered :term:`Output .csv` files together (reduce operation).\n\n :term:`Output .csv`s gathered from N :term:`Experimental Runs <Experimental\n Run>` are combined together into a single :term:`Summary .csv` per\n :term:`Experiment` with 1 column per run.\n\n '

__init__(*main_config: Dict[str, Any]*, *batch_stat_collate_root: pathlib.Path*, *storage_medium: str*, *df_homogenize: str*) → None

__module__ = 'sierra.core.pipeline.stage3.run_collator'

__weakref__
list of weak references to the object (if defined)

sierra.core.pipeline.stage3.statistics_calculator

Classes for generating statistics within and across experiments in a batch.

- *GatherSpec*: Data class for specifying .csv files to gather from an *Experiment*.
- *BatchExpParallelCalculator*: Process *Output .csv* files for each experiment in the batch.
- *ExpCSVGatherer*: Gather all *Output .csv* files from all runs within an experiment.
- *ExpStatisticsCalculator*: Generate statistics from output files for all runs within an experiment.

class sierra.core.pipeline.stage3.statistics_calculator.**GatherSpec**(*exp_name: str*, *item_stem: str*, *imagize_csv_stem: Optional[str]*)
Data class for specifying .csv files to gather from an *Experiment*.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage3.statistics_calculator', '__doc__': '\n Data class for specifying .csv files to gather from an :term:`Experiment`.\n ', '__init__': <function GatherSpec.__init__>, 'for_imagizing': <function GatherSpec.for_imagizing>, '__dict__': <attribute '__dict__' of 'GatherSpec' objects>, '__weakref__': <attribute '__weakref__' of 'GatherSpec' objects>, '__annotations__': {}})

__doc__ = '\n Data class for specifying .csv files to gather from an :term:`Experiment`.\n '

__init__(*exp_name: str*, *item_stem: str*, *imagize_csv_stem: Optional[str]*)

__module__ = 'sierra.core.pipeline.stage3.statistics_calculator'

__weakref__
list of weak references to the object (if defined)

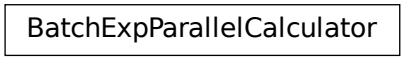
for_imagizing()

class sierra.core.pipeline.stage3.statistics_calculator.**BatchExpParallelCalculator**(*main_config: dict*, *cmdopts: Dict[str, Any]*)

Process *Output .csv* files for each experiment in the batch.

In parallel for speed.

Inheritance



__call__(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage3.statistics_calculator', '__doc__': 'Process :term:`Output .csv` files for each experiment in the batch.\n\n In parallel for speed.\n ', '__init__': <function BatchExpParallelCalculator.__init__>, '__call__': <function BatchExpParallelCalculator.__call__>, '_execute': <function BatchExpParallelCalculator._execute>, '_gather_worker': <staticmethod object>, '_process_worker': <staticmethod object>, '__dict__': <attribute '__dict__' of 'BatchExpParallelCalculator' objects>, '__weakref__': <attribute '__weakref__' of 'BatchExpParallelCalculator' objects>, '__annotations__': {}})

__doc__ = 'Process :term:`Output .csv` files for each experiment in the batch.\n\n In parallel for speed.\n '

__init__(*main_config: dict*, *cmdopts: Dict[str, Any]*)

__module__ = 'sierra.core.pipeline.stage3.statistics_calculator'

__weakref__
list of weak references to the object (if defined)

_execute(*exp_to_avg: List[pathlib.Path]*, *avg_opts: Dict[str, Union[str, int]]*, *n_gatherers: int*, *n_processors: int*, *pool*) → None

static _gather_worker(*gatherq: multiprocessing.context.BaseContext.Queue*, *processq: multiprocessing.context.BaseContext.Queue*, *main_config: Dict[str, Any]*, *avg_opts: Dict[str, str]*) → None

static _process_worker(*processq: multiprocessing.context.BaseContext.Queue*, *main_config: Dict[str, Any]*, *batch_stat_root: pathlib.Path*, *avg_opts: Dict[str, str]*) → None

class sierra.core.pipeline.stage3.statistics_calculator.**ExpCSVGatherer**(*main_config: Dict[str, Any]*, *gather_opts: dict*, *processq: multiprocessing.context.BaseContext.Queue*)

Gather all *Output .csv* files from all runs within an experiment.

"Gathering" in this context means creating a dictionary mapping which .csv came from where, so that statistics can be generated both across and with experiments in the batch.

Inheritance

__call__(*exp_output_root: pathlib.Path*) → None
Process the CSV files found in the output save path.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage3.statistics_calculator', '__doc__': 'Gather all :term:`Output .csv` files from all runs within an experiment.\n\n "Gathering" in this context means creating a dictionary mapping which .csv\n came from where, so that statistics can be generated both across and with\n experiments in the batch.\n ', '__init__': <function ExpCSVGatherer.__init__>, '__call__': <function ExpCSVGatherer.__call__>, '_calc_gather_items': <function ExpCSVGatherer._calc_gather_items>, '_gather_item_from_sims': <function ExpCSVGatherer._gather_item_from_sims>, '_wait_for_memory': <function ExpCSVGatherer._wait_for_memory>, '_verify_exp_outputs': <function ExpCSVGatherer._verify_exp_outputs>, '_verify_exp_outputs_pairwise': <function ExpCSVGatherer._verify_exp_outputs_pairwise>, '__dict__': <attribute '__dict__' of 'ExpCSVGatherer' objects>, '__weakref__': <attribute '__weakref__' of 'ExpCSVGatherer' objects>, '__annotations__': {}})

__doc__ = 'Gather all :term:`Output .csv` files from all runs within an experiment.\n\n "Gathering" in this context means creating a dictionary mapping which .csv\n came from where, so that statistics can be generated both across and with\n experiments in the batch.\n '

__init__(*main_config: Dict[str, Any]*, *gather_opts: dict*, *processq: multiprocessing.context.BaseContext.Queue*) → None

__module__ = 'sierra.core.pipeline.stage3.statistics_calculator'

__weakref__
list of weak references to the object (if defined)

_calc_gather_items(*run_output_root: pathlib.Path*, *exp_name: str*) → List[*sierra.core.pipeline.stage3.statistics_calculator.GatherSpec*]

_gather_item_from_sims(*exp_output_root: pathlib.Path*, *item:* sierra.core.pipeline.stage3.statistics_calculator.GatherSpec, *runs: List[pathlib.Path]*) → Dict[*sierra.core.pipeline.stage3.statistics_calculator.GatherSpec*, List[pandas.core.frame.DataFrame]]

_verify_exp_outputs(*exp_output_root: pathlib.Path*) → None
Verify the integrity of all runs in an experiment.

Specifically:

- All runs produced all CSV files.
- All runs CSV files with the same name have the same # rows and columns.
- No CSV files contain NaNs.

_verify_exp_outputs_pairwise(*csv_root1: pathlib.Path*, *csv_root2: pathlib.Path*) → None

_wait_for_memory() → None

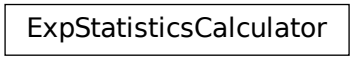
class sierra.core.pipeline.stage3.statistics_calculator.**ExpStatisticsCalculator**(*main_config: Dict[str, Any]*, *avg_opts: dict*, *batch_stat_root: path-lib.Path*)

Generate statistics from output files for all runs within an experiment.

Important: You *CANNOT* use logging ANYWHERE during processing .csv files. Why ? I *think* because of a bug in the logging module itself. If you get unlucky enough to spawn the process which enters the __call__() method in this class while another logging statement is in progress (and is therefore holding an internal logging module lock), then the underlying fork() call will copy the lock in the acquired state. Then, when this class goes to try to log something, it deadlocks with itself.

You also can't just create loggers with unique names, as this seems to be something like the GIL, but for the logging module. Sometimes python sucks.

Inheritance



__call__(*gather_spec:* sierra.core.pipeline.stage3.statistics_calculator.GatherSpec, *gathered_dfs:* *List[pandas.core.frame.DataFrame]*) → None

Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage3.statistics_calculator', '__doc__': "Generate statistics from output files for all runs within an experiment.\n\n .. IMPORTANT:: You *CANNOT* use logging ANYWHERE during processing .csv\n files. Why ? I *think* because of a bug in the logging module itself. If\n you get unlucky enough to spawn the process which enters the __call__()\n method in this class while another logging statement is in progress (and\n is therefore holding an internal logging module lock), then the\n underlying fork() call will copy the lock in the acquired state. Then,\n when this class goes to try to log something, it deadlocks with itself.\n\n You also can't just create loggers with unique names, as this seems to be\n something like the GIL, but for the logging module. Sometimes python\n sucks.\n ", '__init__': <function ExpStatisticsCalculator.__init__>, '__call__': <function ExpStatisticsCalculator.__call__>, '__dict__': <attribute '__dict__' of 'ExpStatisticsCalculator' objects>, '__weakref__': <attribute '__weakref__' of 'ExpStatisticsCalculator' objects>, '__annotations__': {}})

__doc__ = "Generate statistics from output files for all runs within an experiment.\n\n .. IMPORTANT:: You *CANNOT* use logging ANYWHERE during processing .csv\n files. Why ? I *think* because of a bug in the logging module itself. If\n you get unlucky enough to spawn the process which enters the __call__()\n method in this class while another logging statement is in progress (and\n is therefore holding an internal logging module lock), then the\n underlying fork() call will copy the lock in the acquired state. Then,\n when this class goes to try to log something, it deadlocks with itself.\n\n You also can't just create loggers with unique names, as this seems to be\n something like the GIL, but for the logging module. Sometimes python\n sucks.\n "

__init__(*main_config: Dict[str, Any]*, *avg_opts: dict*, *batch_stat_root: pathlib.Path*) → None

__module__ = 'sierra.core.pipeline.stage3.statistics_calculator'

__weakref__
list of weak references to the object (if defined)

sierra.core.pipeline.stage4

sierra.core.pipeline.stage4.graph_collator

- *UnivarGraphCollator*: For a single graph gather needed data from experiments in a batch.
- *BivarGraphCollator*: For a single graph gather needed data from experiments in a batch.
- *UnivarGraphCollationInfo*: Data class of the *Collated .csv* files for a particular graph.
- *BivarGraphCollationInfo*: Data class of the *Collated .csv* files for a particular graph.

class sierra.core.pipeline.stage4.graph_collator.**UnivarGraphCollator**(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*)

For a single graph gather needed data from experiments in a batch.

Results are put into a single *Collated .csv* file.

Inheritance

__call__(*criteria*, *target: dict*, *stat_collate_root: pathlib.Path*) → None
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage4.graph_collator', '__doc__': 'For a single graph gather needed data from experiments in a batch.\n\n Results are put into a single :term:`Collated .csv` file.\n ', '__init__': <function UnivarGraphCollator.__init__>, '__call__': <function UnivarGraphCollator.__call__>, '_collate_exp': <function UnivarGraphCollator._collate_exp>, '__dict__': <attribute '__dict__' of 'UnivarGraphCollator' objects>, '__weakref__': <attribute '__weakref__' of 'UnivarGraphCollator' objects>, '__annotations__': {}})

__doc__ = 'For a single graph gather needed data from experiments in a batch.\n\n Results are put into a single :term:`Collated .csv` file.\n '

__init__(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.core.pipeline.stage4.graph_collator'

__weakref__
list of weak references to the object (if defined)

_collate_exp(*target: dict*, *exp_dir: str*, *stats: List[*sierra.core.pipeline.stage4.graph_collator.UnivarGraphCollationInfo*]*) → None

class sierra.core.pipeline.stage4.graph_collator.**BivarGraphCollator**(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*)

For a single graph gather needed data from experiments in a batch.

Results are put into a single *Collated .csv* file.

Inheritance

__call__(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria, *target: dict*, *stat_collate_root: pathlib.Path*) → None

Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage4.graph_collator', '__doc__': 'For a single graph gather needed data from experiments in a batch.\n\n Results are put into a single :term:`Collated .csv` file.\n\n ', '__init__': <function BivarGraphCollator.__init__>, '__call__': <function BivarGraphCollator.__call__>, '_collate_exp': <function BivarGraphCollator._collate_exp>, '__dict__': <attribute '__dict__' of 'BivarGraphCollator' objects>, '__weakref__': <attribute '__weakref__' of 'BivarGraphCollator' objects>, '__annotations__': {}})

__doc__ = 'For a single graph gather needed data from experiments in a batch.\n\n Results are put into a single :term:`Collated .csv` file.\n\n '

__init__(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.core.pipeline.stage4.graph_collator'

__weakref__

list of weak references to the object (if defined)

_collate_exp(*target: dict*, *exp_dir: str*, *stats: List[*sierra.core.pipeline.stage4.graph_collator.BivarGraphCollationInfo*]*) → None

class sierra.core.pipeline.stage4.graph_collator.**UnivarGraphCollationInfo**(*df_ext: str*, *ylabels: List[str]*)

Data class of the *Collated .csv* files for a particular graph.

Inheritance

UnivarGraphCollationInfo

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage4.graph_collator', '__doc__': 'Data class of the :term:`Collated .csv` files for a particular graph.\n\n ', '__init__': <function UnivarGraphCollationInfo.__init__>, '__dict__': <attribute '__dict__' of 'UnivarGraphCollationInfo' objects>, '__weakref__': <attribute '__weakref__' of 'UnivarGraphCollationInfo' objects>, '__annotations__': {}})

__doc__ = 'Data class of the :term:`Collated .csv` files for a particular graph.\n\n '

__init__(*df_ext: str*, *ylabels: List[str]*) → None

__module__ = 'sierra.core.pipeline.stage4.graph_collator'

__weakref__
list of weak references to the object (if defined)

class sierra.core.pipeline.stage4.graph_collator.**BivarGraphCollationInfo**(*df_ext: str*, *xlabels: List[str]*, *ylabels: List[str]*)

Data class of the *Collated .csv* files for a particular graph.

Inheritance

BivarGraphCollationInfo

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage4.graph_collator', '__doc__': 'Data class of the :term:`Collated .csv` files for a particular graph.\n\n ', '__init__': <function BivarGraphCollationInfo.__init__>, '__dict__': <attribute '__dict__' of 'BivarGraphCollationInfo' objects>, '__weakref__': <attribute '__weakref__' of 'BivarGraphCollationInfo' objects>, '__annotations__': {'df_seq': 'tp.Dict[int, pd.DataFrame]'}})

__doc__ = 'Data class of the :term:`Collated .csv` files for a particular graph.\n\n '

__init__(*df_ext: str*, *xlabels: List[str]*, *ylabels: List[str]*) → None

__module__ = 'sierra.core.pipeline.stage4.graph_collator'

__weakref__
list of weak references to the object (if defined)

sierra.core.pipeline.stage4.inter_exp_graph_generator

Classes for generating graphs across experiments in a batch.

- *InterExpGraphGenerator*: Generates graphs from *Collated .csv* files.
- *LineGraphsGenerator*: Generates linegraphs from *Collated .csv* files.
- *HeatmapsGenerator*: Generates heatmaps from *Collated .csv* files.

class sierra.core.pipeline.stage4.inter_exp_graph_generator.**InterExpGraphGenerator**(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*, *LN_targets: List[Dict[str, Any]]*, *HM_targets: List[Dict[str, Any]]*)

Generates graphs from *Collated .csv* files.

Which graphs are generated can be controlled by YAML configuration files parsed in *PipelineStage4*.

This class can be extended/overriden using a *Project* hook. See *SIERRA Hooks* for details.

cmdopts
Dictionary of parsed cmdline attributes.

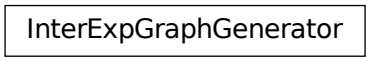
main_config
Parsed dictionary of main YAML configuration

LN_targets
A list of dictionaries, where each dictionary defines an inter-experiment linegraph to generate.

HM_targets
A list of dictionaries, where each dictionary defines an inter-experiment heatmap to generate.

logger
The handle to the logger for this class. If you extend this class, you should save/restore this variable in tandem with overriding it in order to get logging messages have unique logger names between this class and your derived class, in order to reduce confusion.

Inheritance



__call__(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None
Generate graphs.

Performs the following steps:

#. *LineGraphsGenerator* to generate linegraphs (univariate batch criteria only).

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage4.inter_exp_graph_generator', '__doc__': 'Generates graphs from :term:`Collated .csv` files.\n\n Which graphs are generated can be controlled by YAML configuration\n files parsed in\n :class:`~sierra.core.pipeline.stage4.pipeline_stage4.PipelineStage4`.\n\n\n This class can be extended/overriden using a :term:`Project` hook. See\n :ref:`ln-sierra-tutorials-project-hooks` for details.\n\n Attributes:\n\n cmdopts: Dictionary of parsed cmdline attributes.\n\n main_config: Parsed dictionary of main YAML configuration\n\n LN_targets: A list of dictionaries, where each dictionary defines an\n inter-experiment linegraph to generate.\n\n HM_targets: A list of dictionaries, where each dictionary defines an\n inter-experiment heatmap to generate.\n\n logger: The handle to the logger for this class. If you extend this\n class, you should save/restore this variable in tandem with\n overriding it in order to get logging messages have unique logger\n names between this class and your derived class, in order to\n reduce confusion.\n\n ', '__init__': <function InterExpGraphGenerator.__init__>, '__call__': <function InterExpGraphGenerator.__call__>, '__dict__': <attribute '__dict__' of 'InterExpGraphGenerator' objects>, '__weakref__': <attribute '__weakref__' of 'InterExpGraphGenerator' objects>, '__annotations__': {}})

__doc__ = 'Generates graphs from :term:`Collated .csv` files.\n\n Which graphs are generated can be controlled by YAML configuration\n files parsed in\n :class:`~sierra.core.pipeline.stage4.pipeline_stage4.PipelineStage4`.\n\n\n This class can be extended/overriden using a :term:`Project` hook. See\n :ref:`ln-sierra-tutorials-project-hooks` for details.\n\n Attributes:\n\n cmdopts: Dictionary of parsed cmdline attributes.\n\n main_config: Parsed dictionary of main YAML configuration\n\n LN_targets: A list of dictionaries, where each dictionary defines an\n inter-experiment linegraph to generate.\n\n HM_targets: A list of dictionaries, where each dictionary defines an\n inter-experiment heatmap to generate.\n\n logger: The handle to the logger for this class. If you extend this\n class, you should save/restore this variable in tandem with\n overriding it in order to get logging messages have unique logger\n names between this class and your derived class, in order to\n reduce confusion.\n\n '

__init__(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*, *LN_targets: List[Dict[str, Any]]*, *HM_targets: List[Dict[str, Any]]*) → None

__module__ = 'sierra.core.pipeline.stage4.inter_exp_graph_generator'

__weakref__
list of weak references to the object (if defined)

class sierra.core.pipeline.stage4.inter_exp_graph_generator.**LineGraphsGenerator**(*cmdopts: Dict[str, Any]*, *targets: List[Dict[str, Any]]*)

Generates linegraphs from *Collated .csv* files.

The graphs generated by this class respect the `--exp-range` cmdline option.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage4.inter_exp_graph_generator', '__doc__': 'Generates linegraphs from :term:`Collated .csv` files.\n\n The graphs generated by this class respect the ``--exp-range`` cmdline\n option.\n ', '__init__': <function LineGraphsGenerator.__init__>, 'generate': <function LineGraphsGenerator.generate>, '_gen_summary_linegraph': <function LineGraphsGenerator._gen_summary_linegraph>, '_gen_stacked_linegraph': <function LineGraphsGenerator._gen_stacked_linegraph>, '__dict__': <attribute '__dict__' of 'LineGraphsGenerator' objects>, '__weakref__': <attribute '__weakref__' of 'LineGraphsGenerator' objects>, '__annotations__': {}})

__doc__ = 'Generates linegraphs from :term:`Collated .csv` files.\n\n The graphs generated by this class respect the ``--exp-range`` cmdline\n option.\n '

__init__(*cmdopts: Dict[str, Any]*, *targets: List[Dict[str, Any]]*) → None

__module__ = 'sierra.core.pipeline.stage4.inter_exp_graph_generator'

__weakref__
list of weak references to the object (if defined)

_gen_stacked_linegraph(*graph: Dict[str, Any]*, *criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria, *graph_root: pathlib.Path*) → None

_gen_summary_linegraph(*graph: Dict[str, Any]*, *criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria, *graph_root: pathlib.Path*) → None

generate(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None

class sierra.core.pipeline.stage4.inter_exp_graph_generator.**HeatmapsGenerator**(*cmdopts: Dict[str, Any]*, *targets: List[Dict[str, Any]]*)

Generates heatmaps from *Collated .csv* files.

The graphs generated by this class respect the `--exp-range` cmdline option.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage4.inter_exp_graph_generator', '__doc__': 'Generates heatmaps from :term:`Collated .csv` files.\n\n The graphs generated by this class respect the ``--exp-range`` cmdline\n option.\n ', '__init__': <function HeatmapsGenerator.__init__>, 'generate': <function HeatmapsGenerator.generate>, '_generate_hm': <function HeatmapsGenerator._generate_hm>, '__dict__': <attribute '__dict__' of 'HeatmapsGenerator' objects>, '__weakref__': <attribute '__weakref__' of 'HeatmapsGenerator' objects>, '__annotations__': {}})

__doc__ = 'Generates heatmaps from :term:`Collated .csv` files.\n\n The graphs generated by this class respect the ``--exp-range`` cmdline\n option.\n '

__init__(*cmdopts: Dict[str, Any]*, *targets: List[Dict[str, Any]]*) → None

__module__ = 'sierra.core.pipeline.stage4.inter_exp_graph_generator'

__weakref__
list of weak references to the object (if defined)

_generate_hm(*graph: Dict[str, Any]*, *criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria, *interval: int*, *ipath: pathlib.Path*) → None

generate(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None

sierra.core.pipeline.stage4.intra_exp_graph_generator

Classes for generating graphs within a single *Experiment*.

- *BatchIntraExpGraphGenerator*: Undocumented.
- *IntraExpGraphGenerator*: Generates graphs from *Averaged .csv* files for a single experiment.
- *LinegraphsGenerator*: Generates linegraphs from *Averaged .csv* files within an experiment.
- *HeatmapsGenerator*: Generates heatmaps from *Averaged .csv* files for a single experiment.

class sierra.core.pipeline.stage4.intra_exp_graph_generator.**BatchIntraExpGraphGenerator**(*cmdopts: Dict[str, Any]*)

Inheritance

__call__(*main_config: Dict[str, Any]*, *controller_config: Dict[str, Any]*, *LN_config: Dict[str, Any]*, *HM_config: Dict[str, Any]*, *criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None

Generate all intra-experiment graphs for a *Batch Experiment*.

Parameters

- **main_config** – Parsed dictionary of main YAML configuration
- **controller_config** – Parsed dictionary of controller YAML configuration.
- **LN_config** – Parsed dictionary of intra-experiment linegraph configuration.
- **HM_config** – Parsed dictionary of intra-experiment heatmap configuration.
- **criteria** – The *Batch Criteria* used for the batch experiment.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage4.intra_exp_graph_generator', '__init__': <function BatchIntraExpGraphGenerator.__init__>, '__call__': <function BatchIntraExpGraphGenerator.__call__>, '__dict__': <attribute '__dict__' of 'BatchIntraExpGraphGenerator' objects>, '__weakref__': <attribute '__weakref__' of 'BatchIntraExpGraphGenerator' objects>, '__doc__': None, '__annotations__': {}})

__doc__ = None

__init__(*cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.core.pipeline.stage4.intra_exp_graph_generator'

__weakref__

list of weak references to the object (if defined)

class sierra.core.pipeline.stage4.intra_exp_graph_generator.**IntraExpGraphGenerator**(*main_config: Dict[str, Any]*, *controller_config: Dict[str, Any]*, *LN_config: Dict[str, Any]*, *HM_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*)

Generates graphs from *Averaged .csv* files for a single experiment.

Which graphs are generated is controlled by YAML configuration files parsed in `PipelineStage4`.

This class can be extended/overriden using a *Project* hook. See *SIERRA Hooks* for details.

cmdopts
Dictionary of parsed cmdline attributes.

main_config
Parsed dictionary of main YAML configuration

controller_config
Parsed dictionary of controller YAML configuration.

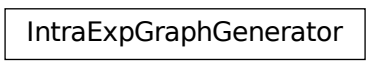
LN_config
Parsed dictionary of intra-experiment linegraph configuration.

HM_config
Parsed dictionary of intra-experiment heatmap configuration.

criteria
The *Batch Criteria* used for the batch experiment.

logger
The handle to the logger for this class. If you extend this class, you should save/restore this variable in tandem with overriding it in order to get logging messages have unique logger names between this class and your derived class, in order to reduce confusion.

Inheritance



__call__(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None
Generate graphs.

Performs the following steps:

#. *LinegraphsGenerator* to generate linegraphs for each experiment in the batch.

#. *HeatmapsGenerator* to generate heatmaps for each experiment in the batch.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage4.intra_exp_graph_generator', '__doc__': 'Generates graphs from :term:`Averaged .csv` files for a single experiment.\n\n Which graphs are generated is controlled by YAML configuration files parsed\n in :class:`~sierra.core.pipeline.stage4.pipeline_stage4.PipelineStage4`.\n\n This class can be extended/overriden using a :term:`Project` hook. See\n :ref:`ln-sierra-tutorials-project-hooks` for details.\n\n Attributes:\n\n cmdopts: Dictionary of parsed cmdline attributes.\n\n main_config: Parsed dictionary of main YAML configuration\n\n controller_config: Parsed dictionary of controller YAML\n configuration.\n\n LN_config: Parsed dictionary of intra-experiment linegraph\n configuration.\n\n HM_config: Parsed dictionary of intra-experiment heatmap\n configuration.\n\n criteria: The :term:`Batch Criteria` used for the batch\n experiment.\n\n logger: The handle to the logger for this class. If you extend this\n class, you should save/restore this variable in tandem with\n overriding it in order to get logging messages have unique logger\n names between this class and your derived class, in order to\n reduce confusion.\n\n ', '__init__': <function IntraExpGraphGenerator.__init__>, '__call__': <function IntraExpGraphGenerator.__call__>, 'generate': <function IntraExpGraphGenerator.generate>, 'calc_targets': <function IntraExpGraphGenerator.calc_targets>, '__dict__': <attribute '__dict__' of 'IntraExpGraphGenerator' objects>, '__weakref__': <attribute '__weakref__' of 'IntraExpGraphGenerator' objects>, '__annotations__': {}})

__doc__ = 'Generates graphs from :term:`Averaged .csv` files for a single experiment.\n\n Which graphs are generated is controlled by YAML configuration files parsed\n in :class:`~sierra.core.pipeline.stage4.pipeline_stage4.PipelineStage4`.\n\n This class can be extended/overriden using a :term:`Project` hook. See\n :ref:`ln-sierra-tutorials-project-hooks` for details.\n\n Attributes:\n\n cmdopts: Dictionary of parsed cmdline attributes.\n\n main_config: Parsed dictionary of main YAML configuration\n\n controller_config: Parsed dictionary of controller YAML\n configuration.\n\n LN_config: Parsed dictionary of intra-experiment linegraph\n configuration.\n\n HM_config: Parsed dictionary of intra-experiment heatmap\n configuration.\n\n criteria: The :term:`Batch Criteria` used for the batch\n experiment.\n\n logger: The handle to the logger for this class. If you extend this\n class, you should save/restore this variable in tandem with\n overriding it in order to get logging messages have unique logger\n names between this class and your derived class, in order to\n reduce confusion.\n\n '

__init__(*main_config: Dict[str, Any]*, *controller_config: Dict[str, Any]*, *LN_config: Dict[str, Any]*, *HM_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.core.pipeline.stage4.intra_exp_graph_generator'

__weakref__
list of weak references to the object (if defined)

calc_targets() → Tuple[List[Dict[str, Any]], List[Dict[str, Any]]]
Calculate what intra-experiment graphs should be generated.

Uses YAML configuration for controller and intra-experiment graphs. Returns a tuple of dictionaries: (intra-experiment linegraphs, intra-experiment heatmaps) defined what graphs to generate. The enabled graphs exist in their YAML respective YAML configuration *and* are enabled by the YAML configuration for the selected controller.

generate(*LN_targets: List[Dict[str, Any]]*, *HM_targets: List[Dict[str, Any]]*)

class sierra.core.pipeline.stage4.intra_exp_graph_generator.**LinegraphsGenerator**(*cmdopts: Dict[str, Any]*, *targets: List[Dict[str, Any]]*)

Generates linegraphs from *Averaged .csv* files within an experiment.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage4.intra_exp_graph_generator', '__doc__': '\n Generates linegraphs from :term:`Averaged .csv` files within an experiment.\n ', '__init__': <function LinegraphsGenerator.__init__>, 'generate': <function LinegraphsGenerator.generate>, '__dict__': <attribute '__dict__' of 'LinegraphsGenerator' objects>, '__weakref__': <attribute '__weakref__' of 'LinegraphsGenerator' objects>, '__annotations__': {}})

__doc__ = '\n Generates linegraphs from :term:`Averaged .csv` files within an experiment.\n '

__init__(*cmdopts: Dict[str, Any]*, *targets: List[Dict[str, Any]]*) → None

__module__ = 'sierra.core.pipeline.stage4.intra_exp_graph_generator'

__weakref__
list of weak references to the object (if defined)

generate() → None

class sierra.core.pipeline.stage4.intra_exp_graph_generator.**HeatmapsGenerator**(*cmdopts: Dict[str, Any]*, *targets: List[Dict[str, Any]]*)

Generates heatmaps from *Averaged .csv* files for a single experiment.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage4.intra_exp_graph_generator', '__doc__': '\n Generates heatmaps from :term:`Averaged .csv` files for a single experiment.\n ', '__init__': <function HeatmapsGenerator.__init__>, 'generate': <function HeatmapsGenerator.generate>, '__dict__': <attribute '__dict__' of 'HeatmapsGenerator' objects>, '__weakref__': <attribute '__weakref__' of 'HeatmapsGenerator' objects>, '__annotations__': {}})

__doc__ = '\n Generates heatmaps from :term:`Averaged .csv` files for a single experiment.\n '

__init__(*cmdopts: Dict[str, Any]*, *targets: List[Dict[str, Any]]*) → None

__module__ = 'sierra.core.pipeline.stage4.intra_exp_graph_generator'

__weakref__
list of weak references to the object (if defined)

generate() → None

sierra.core.pipeline.stage4.model_runner

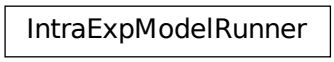
Classes for running project-specific *Models*.

- *IntraExpModelRunner*: Runs all enabled intra-experiment models for all experiments in a batch.
- *InterExpModelRunner*: Runs all enabled inter-experiment models in a batch.

class sierra.core.pipeline.stage4.model_runner.**IntraExpModelRunner**(*cmdopts: Dict[str, Any]*, *to_run: List[Union[*sierra.core.models.interface.IConcr sierra.core.models.interface.IConcreteIntraExpM

Runs all enabled intra-experiment models for all experiments in a batch.

Inheritance



__call__(*main_config: Dict[str, Any]*, *criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage4.model_runner', '__doc__': '\n Runs all enabled intra-experiment models for all experiments in a batch.\n ', '__init__': <function IntraExpModelRunner.__init__>, '__call__': <function IntraExpModelRunner.__call__>, '_run_models_in_exp': <function IntraExpModelRunner._run_models_in_exp>, '_run_model_in_exp': <function IntraExpModelRunner._run_model_in_exp>, '__dict__': <attribute '__dict__' of 'IntraExpModelRunner' objects>, '__weakref__': <attribute '__weakref__' of 'IntraExpModelRunner' objects>, '__annotations__': {}})

__doc__ = '\n Runs all enabled intra-experiment models for all experiments in a batch.\n '

__init__(*cmdopts: Dict[str, Any]*, *to_run: List[Union[*sierra.core.models.interface.IConcreteIntraExpModel1D, sierra.core.models.interface.IConcreteIntraExpModel2D*]]*) → None

__module__ = 'sierra.core.pipeline.stage4.model_runner'

__weakref__
list of weak references to the object (if defined)

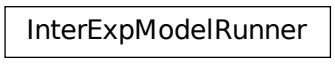
_run_model_in_exp(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria, *cmdopts: Dict[str, Any]*, *exp_index: int*, *model: Union[*sierra.core.models.interface.IConcreteIntraExpModel1D, sierra.core.models.interface.IConcreteIntraExpModel2D*]*) → None

_run_models_in_exp(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria, *exp_dirnames: List[pathlib.Path]*, *exp: pathlib.Path*) → None

class sierra.core.pipeline.stage4.model_runner.**InterExpModelRunner**(*cmdopts: Dict[str, Any]*, *to_run: List[*sierra.core.models.interface.IConcreteInter

Runs all enabled inter-experiment models in a batch.

Inheritance



__call__(*main_config: Dict[str, Any]*, *criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage4.model_runner', '__doc__': '\n Runs all enabled inter-experiment models in a batch.\n ', '__init__': <function InterExpModelRunner.__init__>, '__call__': <function InterExpModelRunner.__call__>, '__dict__': <attribute '__dict__' of 'InterExpModelRunner' objects>, '__weakref__': <attribute '__weakref__' of 'InterExpModelRunner' objects>, '__annotations__': {}})

__doc__ = '\n Runs all enabled inter-experiment models in a batch.\n '

__init__(*cmdopts: Dict[str, Any]*, *to_run: List[*sierra.core.models.interface.IConcreteInterExpModel1D*]*) → None

__module__ = 'sierra.core.pipeline.stage4.model_runner'

__weakref__
list of weak references to the object (if defined)

sierra.core.pipeline.stage4.pipeline_stage4

Stage 4 of the experimental pipeline: generating deliverables.

- *PipelineStage4*: Generates end-result experimental deliverables.

class sierra.core.pipeline.stage4.pipeline_stage4.**PipelineStage4**(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*)

Generates end-result experimental deliverables.

Delvirables can be within a single experiment (intra-experiment) and across experiments in a batch (inter-experiment). Currently this includes:

- Graph generation controlled via YAML config files.
- Video rendering controlled via YAML config files.

This stage is idempotent.

cmdopts
Dictionary of parsed cmdline options.

controller_config
YAML configuration file found in <project_config_root>/controllers.yaml. Contains configuration for what categories of graphs should be generated for what controllers, for all categories of graphs in both inter- and intra-experiment graph generation.

inter_LN_config
YAML configuration file found in <project_config_root>/inter-graphs-line.yaml Contains configuration for categories of linegraphs that can potentially be generated for all controllers *across* experiments in a batch. Which linegraphs are actually generated for a given controller is controlled by <project_config_root>/controllers.yaml.

intra_LN_config
YAML configuration file found in <project_config_root>/intra-graphs-line.yaml Contains configuration for categories of linegraphs that can potentially be generated for all controllers *within* each experiment in a batch. Which linegraphs are actually generated for a given controller in each experiment is controlled by <project_config_root>/controllers.yaml.

intra_HM_config
YAML configuration file found in <project_config_root>/intra-graphs-hm.yaml Contains configuration for categories of heatmaps that can potentially be generated for all controllers *within* each ex-

periment in a batch. Which heatmaps are actually generated for a given controller in each experiment is controlled by <project_config_root>/controllers.yaml.

inter_HM_config
YAML configuration file found in <project_config_root>/inter-graphs-hm.yaml Contains configuration for categories of heatmaps that can potentially be generated for all controllers *across* each experiment in a batch. Which heatmaps are actually generated for a given controller in each experiment is controlled by <project_config_root>/controllers.yaml.

Inheritance

__dict__ = mappingproxy({'__module__':
'sierra.core.pipeline.stage4.pipeline_stage4', '__doc__': 'Generates end-result
experimental deliverables.\n\n Delvirables can be within a single experiment
(intra-experiment) and across\n experiments in a batch (inter-experiment). Currently
this includes:\n\n - Graph generation controlled via YAML config files.\n\n - Video
rendering controlled via YAML config files.\n\n This stage is idempotent.\n\n
Attributes:\n\n cmdopts: Dictionary of parsed cmdline options.\n\n
controller_config: YAML configuration file found in\n
``<project_config_root>/controllers.yaml``. Contains\n configuration for what
categories of graphs should be\n generated for what controllers, for all categories
of\n graphs in both inter- and intra-experiment graph\n generation.\n\n
inter_LN_config: YAML configuration file found in\n
``<project_config_root>/inter-graphs-line.yaml``\n Contains configuration for
categories of linegraphs\n that can potentially be generated for all controllers\n
`across` experiments in a batch. Which linegraphs are\n actually generated for a
given controller is controlled\n by ``<project_config_root>/controllers.yaml``.\n\n
intra_LN_config: YAML configuration file found in\n
``<project_config_root>/intra-graphs-line.yaml``\n Contains configuration for
categories of linegraphs\n that can potentially be generated for all controllers\n
`within` each experiment in a batch. Which linegraphs\n are actually generated for a
given controller in each\n experiment is controlled by\n
``<project_config_root>/controllers.yaml``.\n\n intra_HM_config: YAML configuration
file found in\n ``<project_config_root>/intra-graphs-hm.yaml`` Contains\n
configuration for categories of heatmaps that can\n potentially be generated for all
controllers `within`\n each experiment in a batch. Which heatmaps are actually\n
generated for a given controller in each experiment is\n controlled by\n
``<project_config_root>/controllers.yaml``.\n\n inter_HM_config: YAML configuration
file found in\n ``<project_config_root>/inter-graphs-hm.yaml`` Contains\n
configuration for categories of heatmaps that can\n potentially be generated for all
controllers `across`\n each experiment in a batch. Which heatmaps are actually\n
generated for a given controller in each experiment is\n controlled by\n
``<project_config_root>/controllers.yaml``.\n\n ', '__init__': <function
PipelineStage4.__init__>, 'run': <function PipelineStage4.run>, '_load_models':
<function PipelineStage4._load_models>, '_calc_inter_targets': <function
PipelineStage4._calc_inter_targets>, '_run_rendering': <function
PipelineStage4._run_rendering>, '_run_intra_models': <function
PipelineStage4._run_intra_models>, '_run_inter_models': <function
PipelineStage4._run_inter_models>, '_run_intra_graph_generation': <function
PipelineStage4._run_intra_graph_generation>, '_run_collation': <function
PipelineStage4._run_collation>, '_run_inter_graph_generation': <function
PipelineStage4._run_inter_graph_generation>, '__dict__': <attribute '__dict__' of
'PipelineStage4' objects>, '__weakref__': <attribute '__weakref__' of
'PipelineStage4' objects>, '__annotations__': {}})

__doc__ = 'Generates end-result experimental deliverables.\n\n Delvirables can be within a single experiment (intra-experiment) and across\n experiments in a batch (inter-experiment). Currently this includes:\n\n - Graph generation controlled via YAML config files.\n\n - Video rendering controlled via YAML config files.\n\n This stage is idempotent.\n\n Attributes:\n\n cmdopts: Dictionary of parsed cmdline options.\n\n controller_config: YAML configuration file found in\n ``<project_config_root>/controllers.yaml``. Contains\n configuration for what categories of graphs should be\n generated for what controllers, for all categories of\n graphs in both inter- and intra-experiment graph\n generation.\n\n inter_LN_config: YAML configuration file found in\n ``<project_config_root>/inter-graphs-line.yaml``\n Contains configuration for categories of linegraphs\n that can potentially be generated for all controllers\n `across` experiments in a batch. Which linegraphs are\n actually generated for a given controller is controlled\n by ``<project_config_root>/controllers.yaml``.\n\n intra_LN_config: YAML configuration file found in\n ``<project_config_root>/intra-graphs-line.yaml``\n Contains configuration for categories of linegraphs\n that can potentially be generated for all controllers\n `within` each experiment in a batch. Which linegraphs\n are actually generated for a given controller in each\n experiment is controlled by\n ``<project_config_root>/controllers.yaml``.\n\n intra_HM_config: YAML configuration file found in\n ``<project_config_root>/intra-graphs-hm.yaml`` Contains\n configuration for categories of heatmaps that can\n potentially be generated for all controllers `within`\n each experiment in a batch. Which heatmaps are actually\n generated for a given controller in each experiment is\n controlled by\n ``<project_config_root>/controllers.yaml``.\n\n inter_HM_config: YAML configuration file found in\n ``<project_config_root>/inter-graphs-hm.yaml`` Contains\n configuration for categories of heatmaps that can\n potentially be generated for all controllers `across`\n each experiment in a batch. Which heatmaps are actually\n generated for a given controller in each experiment is\n controlled by\n ``<project_config_root>/controllers.yaml``.\n\n '

__init__(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.core.pipeline.stage4.pipeline_stage4'

__weakref__
list of weak references to the object (if defined)

_calc_inter_targets(*name: str*, *category_prefix: str*, *loaded_graphs: Dict[str, Any]*) → List[Dict[str, Any]]
Calculate what inter-experiment graphs to generate.

This also defines what CSV files need to be collated, as one graph is always generated from one CSV file. Uses YAML configuration for controllers and inter-experiment graphs.

_load_models() → None

_run_collation(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None

_run_inter_graph_generation(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None
Generate inter-experiment graphs (duh).

_run_inter_models(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None

_run_intra_graph_generation(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None
Generate intra-experiment graphs (duh).

_run_intra_models(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None

_run_rendering(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None
Render captured frames and/or imagized frames into videos.

run(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None
Run the pipeline stage.

Intra-experiment graph generation: if intra-experiment graphs should be generated, according to cmdline configuration, the following is run:

1. Model generation for each enabled and loaded model.
2. *BatchIntraExpGraphGenerator* to generate graphs for each experiment in the batch, or a subset.

Inter-experiment graph generation: if inter-experiment graphs should be generated according to cmdline configuration, the following is run:

1. *UnivarGraphCollator* or *BivarGraphCollator* as appropriate (depending on which type of *BatchCriteria* was specified on the cmdline).
2. Model generation for each enabled and loaded model.
3. *InterExpGraphGenerator* to perform graph generation from collated CSV files.

Video generation: The following is run:

1. *PlatformFramesRenderer*, if `--platform-vc` was passed
2. *ProjectFramesRenderer*, if `--project-imagizing` was passed previously to generate frames, and `--project-rendering` is passed.
3. *BivarHeatmapRenderer*, if the batch criteria was bivariate and `--HM-rendering` was passed.

sierra.core.pipeline.stage4.rendering

Classes for rendering frames (images) into videos.

Frames can be:

- Captured by by the `--platform` during stage 2.
- Generated during stage 3 of SIERRA via imagizing.
- Generated inter-experiment heatmaps from bivariate experiments.

- *ParallelRenderer*: Base class for performing the requested rendering in parallel.
- *PlatformFramesRenderer*: Renders frames (images) captured in each experimental run by a platform.
- *ProjectFramesRenderer*: Render the video for each experimental run in each experiment.
- *BivarHeatmapRenderer*: Render videos from generated inter-experiment heatmaps.
- *ExpRenderer*: Render all images in the input directory to a video via **ffmpeg**.

class sierra.core.pipeline.stage4.rendering.**ParallelRenderer**(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*)

Base class for performing the requested rendering in parallel.

Unless disabled with `--proccessing-serial`, then it is done serially.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage4.rendering', '__doc__': 'Base class for performing the requested rendering in parallel.\n\n Unless disabled with ``--proccessing-serial``, then it is done serially.\n\n ', '__init__': <function ParallelRenderer.__init__>, 'do_rendering': <function ParallelRenderer.do_rendering>, '_thread_worker': <staticmethod object>, '__dict__': <attribute '__dict__' of 'ParallelRenderer' objects>, '__weakref__': <attribute '__weakref__' of 'ParallelRenderer' objects>, '__annotations__': {}})

__doc__ = 'Base class for performing the requested rendering in parallel.\n\n Unless disabled with ``--proccessing-serial``, then it is done serially.\n\n '

__init__(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.core.pipeline.stage4.rendering'

__weakref__
list of weak references to the object (if defined)

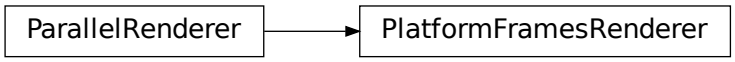
static _thread_worker(*q: multiprocessing.context.BaseContext.Queue*, *main_config: Dict[str, Any]*) → None

do_rendering(*inputs: List[Dict[str, Union[str, int]]]*) → None
Do the rendering.

class sierra.core.pipeline.stage4.rendering.**PlatformFramesRenderer**(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*)

Renders frames (images) captured in each experimental run by a platform.

Inheritance



__call__(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None
Call self as a function.

__doc__ = 'Renders frames (images) captured in each experimental run by a platform.\n\n '

__init__(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*) → None

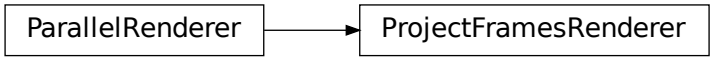
__module__ = 'sierra.core.pipeline.stage4.rendering'

_calc_rendering_inputs(*exp: pathlib.Path*) → List[Dict[str, Union[str, int]]]

class sierra.core.pipeline.stage4.rendering.**ProjectFramesRenderer**(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*)

Render the video for each experimental run in each experiment.

Inheritance



__call__(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None

Call self as a function.

__doc__ = 'Render the video for each experimental run in each experiment.\n '

__init__(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.core.pipeline.stage4.rendering'

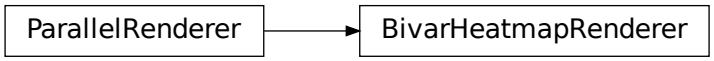
_calc_rendering_inputs(*exp: pathlib.Path*) → List[Dict[str, Union[str, int]]]

class sierra.core.pipeline.stage4.rendering.**BivarHeatmapRenderer**(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*)

Render videos from generated inter-experiment heatmaps.

versionadded:: 1.2.20

Inheritance



__call__(*criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None

Call self as a function.

__doc__ = 'Render videos from generated inter-experiment heatmaps.\n\n versionadded:: 1.2.20\n '

__init__(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.core.pipeline.stage4.rendering'

_calc_rendering_inputs() → List[Dict[str, Union[str, int]]]

class sierra.core.pipeline.stage4.rendering.**ExpRenderer**
Render all images in the input directory to a video via **ffmpeg**.

Inheritance

__call__(*main_config: Dict[str, Any]*, *render_opts: Dict[str, str]*) → None
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage4.rendering', '__doc__': 'Render all images in the input directory to a video via :program:`ffmpeg`.\n\n ', '__init__': <function ExpRenderer.__init__>, '__call__': <function ExpRenderer.__call__>, '__dict__': <attribute '__dict__' of 'ExpRenderer' objects>, '__weakref__': <attribute '__weakref__' of 'ExpRenderer' objects>, '__annotations__': {}})

__doc__ = 'Render all images in the input directory to a video via :program:`ffmpeg`.\n\n '

__init__() → None

__module__ = 'sierra.core.pipeline.stage4.rendering'

__weakref__
list of weak references to the object (if defined)

sierra.core.pipeline.stage4.yaml_config_loader

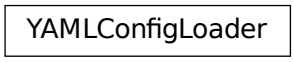
- *YAMLConfigLoader*: Load YAML configuration for *Project* graphs to be generated.

class sierra.core.pipeline.stage4.yaml_config_loader.**YAMLConfigLoader**
Load YAML configuration for *Project* graphs to be generated.

This class can be extended/overriden using a *Project* hook. See *SIERRA Hooks* for details.

logger
The handle to the logger for this class. If you extend this class, you should save/restore this variable in tandem with overriding it in order to get loggingmessages have unique logger names between this class and your derived class, in order to reduce confusion.

Inheritance



__call__(*cmdopts: Dict[str, Any]*) → Dict[str, Dict[str, Any]]

Load YAML configuratoin for graphs.

This includes:

- intra-experiment linegraphs
- inter-experiment linegraphs
- intra-experiment heatmaps
- inter-experiment heatmaps (bivariate batch criteria only)

Returns Dictionary of loaded configuration with keys for `intra_LN`, `inter_LN`, `intra_HM`, `inter_HM`.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage4.yaml_config_loader', '__doc__': 'Load YAML configuration for :term:`Project` graphs to be generated.\n\n This class can be extended/overriden using a :term:`Project` hook. See\n :ref:`ln-sierra-tutorials-project-hooks` for details.\n\n Attributes:\n logger: The handle to the logger for this class. If you extend this\n class, you should save/restore this variable in tandem with\n overriding it in order to get loggingmessages have unique\n logger names between this class and your derived class, in order\n to reduce confusion.\n\n ', '__init__': <function YAMLConfigLoader.__init__>, '__call__': <function YAMLConfigLoader.__call__>, '__dict__': <attribute '__dict__' of 'YAMLConfigLoader' objects>, '__weakref__': <attribute '__weakref__' of 'YAMLConfigLoader' objects>, '__annotations__': {}})

__doc__ = 'Load YAML configuration for :term:`Project` graphs to be generated.\n\n This class can be extended/overriden using a :term:`Project` hook. See\n :ref:`ln-sierra-tutorials-project-hooks` for details.\n\n Attributes:\n logger: The handle to the logger for this class. If you extend this\n class, you should save/restore this variable in tandem with\n overriding it in order to get loggingmessages have unique\n logger names between this class and your derived class, in order\n to reduce confusion.\n\n '

__init__() → None

__module__ = 'sierra.core.pipeline.stage4.yaml_config_loader'

__weakref__

list of weak references to the object (if defined)

sierra.core.pipeline.stage5

sierra.core.pipeline.stage5.inter_scenario_comparator

Classes for comparing deliverables across a set of scenarios.

Univariate batch criteria only. The same controller must be used for all scenarios.

- *UnivarInterScenarioComparator*: Compares a single controller across a set of scenarios.

class sierra.core.pipeline.stage5.inter_scenario_comparator.**UnivarInterScenarioComparator**(*controller: str*, *scenarios: List[str]*, *roots: Dict[str, pathlib.Path]*, *cmdopts: Dict[str, Any]*, *cli_args: argparse.Namespace*, *main_config: Dict[str, Any]*)

Compares a single controller across a set of scenarios.

Graph generation is controlled via a config file parsed in *PipelineStage5*.

Univariate batch criteria only.

controller
Controller to use.

scenarios
List of scenario names to compare `controller` across.

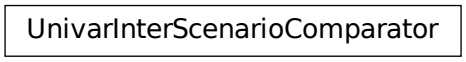
sc_csv_root
Absolute directory path to the location scenario CSV files should be output to.

sc_graph_root
Absolute directory path to the location the generated graphs should be output to.

cmdopts
Dictionary of parsed cmdline parameters.

cli_args
`argparse` object containing the cmdline parameters. Needed for *BatchCriteria* generation for each scenario controllers are compared within, as batch criteria is dependent on controller+scenario definition, and needs to be re-generated for each scenario in order to get graph labels/axis ticks to come out right in all cases.

Inheritance



__call__(*graphs: List[Dict[str, Any]]*, *legend: List[str]*) → None
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage5.inter_scenario_comparator', '__doc__': 'Compares a single controller across a set of scenarios.\n\n Graph generation is controlled via a config file parsed in\n :class:`~sierra.core.pipeline.stage5.pipeline_stage5.PipelineStage5`.\n\n Univariate batch criteria only.\n\n Attributes:\n\n controller: Controller to use.\n\n scenarios: List of scenario names to compare ``controller`` across.\n\n sc_csv_root: Absolute directory path to the location scenario CSV\n files should be output to.\n\n sc_graph_root: Absolute directory path to the location the generated\n graphs should be output to.\n\n cmdopts: Dictionary of parsed cmdline parameters.\n\n cli_args: :class:`argparse` object containing the cmdline\n parameters. Needed for\n :class:`~sierra.core.variables.batch_criteria.BatchCriteria`\n generation for each scenario controllers are compared within,\n as batch criteria is dependent on controller+scenario\n definition, and needs to be re-generated for each scenario in\n order to get graph labels/axis ticks to come out right in all\n cases.\n\n ', '__init__': <function UnivarInterScenarioComparator.__init__>, '__call__': <function UnivarInterScenarioComparator.__call__>, '_leaf_select': <function UnivarInterScenarioComparator._leaf_select>, '_compare_across_scenarios': <function UnivarInterScenarioComparator._compare_across_scenarios>, '_gen_graph': <function UnivarInterScenarioComparator._gen_graph>, '_gen_csvs': <function UnivarInterScenarioComparator._gen_csvs>, '_accum_df': <function UnivarInterScenarioComparator._accum_df>, '__dict__': <attribute '__dict__' of 'UnivarInterScenarioComparator' objects>, '__weakref__': <attribute '__weakref__' of 'UnivarInterScenarioComparator' objects>, '__annotations__': {}})

__doc__ = 'Compares a single controller across a set of scenarios.\n\n Graph generation is controlled via a config file parsed in\n :class:`~sierra.core.pipeline.stage5.pipeline_stage5.PipelineStage5`.\n\n Univariate batch criteria only.\n\n Attributes:\n\n controller: Controller to use.\n\n scenarios: List of scenario names to compare ``controller`` across.\n\n sc_csv_root: Absolute directory path to the location scenario CSV\n files should be output to.\n\n sc_graph_root: Absolute directory path to the location the generated\n graphs should be output to.\n\n cmdopts: Dictionary of parsed cmdline parameters.\n\n cli_args: :class:`argparse` object containing the cmdline\n parameters. Needed for\n :class:`~sierra.core.variables.batch_criteria.BatchCriteria`\n generation for each scenario controllers are compared within,\n as batch criteria is dependent on controller+scenario\n definition, and needs to be re-generated for each scenario in\n order to get graph labels/axis ticks to come out right in all\n cases.\n\n '

__init__(*controller: str*, *scenarios: List[str]*, *roots: Dict[str, pathlib.Path]*, *cmdopts: Dict[str, Any]*, *cli_args: argparse.Namespace*, *main_config: Dict[str, Any]*) → None

__module__ = 'sierra.core.pipeline.stage5.inter_scenario_comparator'

__weakref__
list of weak references to the object (if defined)

_accum_df(*ipath: pathlib.Path*, *opath: pathlib.Path*, *src_stem: str*) → pandas.core.frame.DataFrame

_compare_across_scenarios(*cmdopts: Dict[str, Any]*, *graph: Dict[str, Any]*, *batch_leaf: str*, *legend: List[str]*) → None

_gen_csvs(*cmdopts: Dict[str, Any]*, *batch_leaf: str*, *src_stem: str*, *dest_stem: str*) → None
Generate a set of CSV files for use in inter-scenario graph generation.

Generates:

- **.mean CSV file containing results for each scenario the controller** is being compared across, 1 per line.
- Stastics CSV files containing various statistics for the `.mean` CSV file, 1 per line.
- `.model` file containing model predictions for controller behavior during each scenario, 1 per line (not generated if models were not run the performance measures we are generating graphs for).
- `.legend` file containing legend values for models to plot (not generated if models were not run for the performance measures we are generating graphs for).

_gen_graph(*criteria: sierra.core.variables.batch_criteria.IConcreteBatchCriteria*, *cmdopts: Dict[str, Any]*, *dest_stem: str*, *inc_exps: Optional[str]*, *title: str*, *label: str*, *legend: List[str]*) → None
Generate graph comparing the specified controller across scenarios.

_leaf_select(*candidate: str*) → bool
Figure out if a batch experiment root should be included in the comparison.

Inclusion determined by if a scenario that the selected controller has been run on in the past is part of the set passed that the controller should be compared across (i.e., the controller is not compared across all scenarios it has ever been run on).

sierra.core.pipeline.stage5.intra_scenario_comparator

Classes for comparing deliverables within the same scenario.

Univariate and bivariate batch criteria.

- *UnivarIntraScenarioComparator*: Compares a set of controllers within each of a set of scenarios.
- *BivarIntraScenarioComparator*: Compares a set of controllers within each of a set of scenarios.

class sierra.core.pipeline.stage5.intra_scenario_comparator.**UnivarIntraScenarioComparator**(*controllers: List[str]*, *cc_csv_root: pathlib.Path*, *cc_graph_root: pathlib.Path*, *cmdopts: Dict[str, Any]*, *cli_args*, *main_config: Dict[str, Any]*)

Compares a set of controllers within each of a set of scenarios.

Graph generation is controlled via a config file parsed in *PipelineStage5*.

Univariate batch criteria only.

controllers
List of controller names to compare.

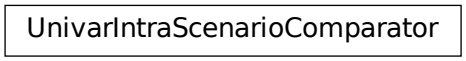
cc_csv_root
Absolute directory path to the location controller CSV files should be output to.

cc_graph_root
Absolute directory path to the location the generated graphs should be output to.

cmdopts
Dictionary of parsed cmdline parameters.

cli_args
`argparse` object containing the cmdline parameters. Needed for *BatchCriteria* generation for each scenario controllers are compared within, as batch criteria is dependent on controller+scenario definition, and needs to be re-generated for each scenario in order to get graph labels/axis ticks to come out right in all cases.

Inheritance



__call__(*graphs: List[Dict[str, Any]]*, *legend: List[str]*, *comp_type: str*) → None
Call self as a function.

__dict__ = mappingproxy({'__module__':
'sierra.core.pipeline.stage5.intra_scenario_comparator', '__doc__': 'Compares a set
of controllers within each of a set of scenarios.\n\n Graph generation\n is
controlled via a config file parsed in\n
:class:`~sierra.core.pipeline.stage5.pipeline_stage5.PipelineStage5`.\n\n Univariate
batch criteria only.\n\n Attributes:\n\n controllers: List of controller names to
compare.\n\n cc_csv_root: Absolute directory path to the location controller CSV\n
files should be output to.\n\n cc_graph_root: Absolute directory path to the
location the generated\n graphs should be output to.\n\n cmdopts: Dictionary of
parsed cmdline parameters.\n\n cli_args: :class:`argparse` object containing the
cmdline\n parameters. Needed for\n
:class:`~sierra.core.variables.batch_criteria.BatchCriteria`\n generation for each
scenario controllers are compared within,\n as batch criteria is dependent on
controller+scenario\n definition, and needs to be re-generated for each scenario
in\n order to get graph labels/axis ticks to come out right in all\n cases.\n\n ',
'__init__': <function UnivarIntraScenarioComparator.__init__>, '__call__':
<function UnivarIntraScenarioComparator.__call__>, '_leaf_select': <function
UnivarIntraScenarioComparator._leaf_select>, '_compare_in_scenario': <function
UnivarIntraScenarioComparator._compare_in_scenario>, '_gen_csv': <function
UnivarIntraScenarioComparator._gen_csv>, '_gen_graph': <function
UnivarIntraScenarioComparator._gen_graph>, '__dict__': <attribute '__dict__' of
'UnivarIntraScenarioComparator' objects>, '__weakref__': <attribute '__weakref__'
of 'UnivarIntraScenarioComparator' objects>, '__annotations__': {}})

__doc__ = 'Compares a set of controllers within each of a set of scenarios.\n\n
Graph generation\n is controlled via a config file parsed in\n
:class:`~sierra.core.pipeline.stage5.pipeline_stage5.PipelineStage5`.\n\n Univariate
batch criteria only.\n\n Attributes:\n\n controllers: List of controller names to
compare.\n\n cc_csv_root: Absolute directory path to the location controller CSV\n
files should be output to.\n\n cc_graph_root: Absolute directory path to the
location the generated\n graphs should be output to.\n\n cmdopts: Dictionary of
parsed cmdline parameters.\n\n cli_args: :class:`argparse` object containing the
cmdline\n parameters. Needed for\n
:class:`~sierra.core.variables.batch_criteria.BatchCriteria`\n generation for each
scenario controllers are compared within,\n as batch criteria is dependent on
controller+scenario\n definition, and needs to be re-generated for each scenario
in\n order to get graph labels/axis ticks to come out right in all\n cases.\n\n '

__init__(*controllers: List[str]*, *cc_csv_root: pathlib.Path*, *cc_graph_root: pathlib.Path*, *cmdopts: Dict[str, Any]*, *cli_args*, *main_config: Dict[str, Any]*) → None

__module__ = 'sierra.core.pipeline.stage5.intra_scenario_comparator'

__weakref__
list of weak references to the object (if defined)

_compare_in_scenario(*cmdopts: Dict[str, Any]*, *graph: Dict[str, Any]*, *batch_leaf: str*, *legend: List[str]*) → None

_gen_csv(*batch_leaf: str*, *criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria, *cmdopts: Dict[str, Any]*, *controller: str*, *src_stem: str*, *dest_stem: str*, *inc_exps: Optional[str]*) → None
Generate a set of CSV files for use in intra-scenario graph generation.

1 CSV per controller.

_gen_graph(*batch_leaf: str*, *criteria:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria, *cmdopts: Dict[str, Any]*, *dest_stem: str*, *title: str*, *label: str*, *inc_exps: Optional[str]*, *legend: List[str]*) → None

Generate a graph comparing the specified controllers within a scenario.

_leaf_select(*candidate: str*) → bool

Determine if a controller can be included in the comparison for a scenario.

You can only compare controllers within the scenario directly generated from the value of `--batch-criteria`; other scenarios will (probably) cause file not found errors.

class sierra.core.pipeline.stage5.intra_scenario_comparator.**BivarIntraScenarioComparator**(*controllers: List[str]*, *cc_csv_root: pathlib.Path*, *cc_graph_root: pathlib.Path*, *cmdopts: Dict[str, Any]*, *cli_args: argparse.Namespace*, *main_config: Dict[str, Any]*)

Compares a set of controllers within each of a set of scenarios.

Graph generation is controlled via a config file parsed in *PipelineStage5*.

Bivariate batch criteria only.

controllers

List of controller names to compare.

cc_csv_root

Absolute directory path to the location controller CSV files should be output to.

cc_graph_root

Absolute directory path to the location the generated graphs should be output to.

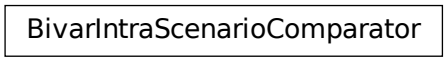
cmdopts

Dictionary of parsed cmdline parameters.

cli_args

`argparse` object containing the cmdline parameters. Needed for *BatchCriteria* generation for each scenario controllers are compared within, as batch criteria is dependent on controller+scenario definition, and needs to be re-generated for each scenario in order to get graph labels/axis ticks to come out right in all cases.

Inheritance



__call__(*graphs: List[Dict[str, Any]]*, *legend: List[str]*, *comp_type: str*) → None
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage5.intra_scenario_comparator', '__doc__': 'Compares a set of controllers within each of a set of scenarios.\n\n Graph generation is controlled via a config file\n parsed in\n :class:`~sierra.core.pipeline.stage5.pipeline_stage5.PipelineStage5`.\n\n Bivariate batch criteria only.\n\n Attributes:\n\n controllers: List of controller names to compare.\n\n cc_csv_root: Absolute directory path to the location controller CSV\n files should be output to.\n\n cc_graph_root: Absolute directory path to the location the generated\n graphs should be output to.\n\n cmdopts: Dictionary of parsed cmdline parameters.\n\n cli_args: :class:`argparse` object containing the cmdline\n parameters. Needed for\n :class:`~sierra.core.variables.batch_criteria.BatchCriteria`\n generation for each scenario controllers are compared within,\n as batch criteria is dependent on controller+scenario\n definition, and needs to be re-generated for each scenario in\n order to get graph labels/axis ticks to come out right in all\n cases.\n\n ', '__init__': <function BivarIntraScenarioComparator.__init__>, '__call__': <function BivarIntraScenarioComparator.__call__>, '_leaf_select': <function BivarIntraScenarioComparator._leaf_select>, '_compare_in_scenario': <function BivarIntraScenarioComparator._compare_in_scenario>, '_gen_csvs_for_2D_or_3D': <function BivarIntraScenarioComparator._gen_csvs_for_2D_or_3D>, '_gen_csvs_for_1D': <function BivarIntraScenarioComparator._gen_csvs_for_1D>, '_gen_graphs1D': <function BivarIntraScenarioComparator._gen_graphs1D>, '_gen_graphs2D': <function BivarIntraScenarioComparator._gen_graphs2D>, '_gen_paired_heatmaps': <function BivarIntraScenarioComparator._gen_paired_heatmaps>, '_gen_dual_heatmaps': <function BivarIntraScenarioComparator._gen_dual_heatmaps>, '_gen_graph3D': <function BivarIntraScenarioComparator._gen_graph3D>, '_gen_zaxis_label': <function BivarIntraScenarioComparator._gen_zaxis_label>, '__dict__': <attribute '__dict__' of 'BivarIntraScenarioComparator' objects>, '__weakref__': <attribute '__weakref__' of 'BivarIntraScenarioComparator' objects>, '__annotations__': {}})

__doc__ = 'Compares a set of controllers within each of a set of scenarios.\n\n Graph generation is controlled via a config file\n parsed in\n :class:`~sierra.core.pipeline.stage5.pipeline_stage5.PipelineStage5`.\n\n Bivariate batch criteria only.\n\n Attributes:\n\n controllers: List of controller names to compare.\n\n cc_csv_root: Absolute directory path to the location controller CSV\n files should be output to.\n\n cc_graph_root: Absolute directory path to the location the generated\n graphs should be output to.\n\n cmdopts: Dictionary of parsed cmdline parameters.\n\n cli_args: :class:`argparse` object containing the cmdline\n parameters. Needed for\n :class:`~sierra.core.variables.batch_criteria.BatchCriteria`\n generation for each scenario controllers are compared within,\n as batch criteria is dependent on controller+scenario\n definition, and needs to be re-generated for each scenario in\n order to get graph labels/axis ticks to come out right in all\n cases.\n\n '

__init__(*controllers: List[str]*, *cc_csv_root: pathlib.Path*, *cc_graph_root: pathlib.Path*, *cmdopts: Dict[str, Any]*, *cli_args: argparse.Namespace*, *main_config: Dict[str, Any]*) → None

__module__ = 'sierra.core.pipeline.stage5.intra_scenario_comparator'

__weakref__
: list of weak references to the object (if defined)

_compare_in_scenario(*cmdopts: Dict[str, Any]*, *graph: Dict[str, Any]*, *batch_leaf: str*, *legend: List[str]*, *comp_type: str*) → None
: Compare all controllers within the specified scenario.

  Generates CSV files and graphs according to configuration.

_gen_csvs_for_1D(*cmdopts: Dict[str, Any]*, *criteria: sierra.core.variables.batch_criteria.IConcreteBatchCriteria*, *batch_leaf: str*, *controller: str*, *src_stem: str*, *dest_stem: str*, *primary_axis: int*, *inc_exps: Optional[str]*) → None
: Generate a set of CSV files for use in intra-scenario graph generation.

  Because we are targeting linegraphs, we draw the the i-th row/col (as configured) from the performance results of each controller .csv, and concatenate them into a new .csv file which can be given to *SummaryLineGraph*.

_gen_csvs_for_2D_or_3D(*cmdopts: Dict[str, Any]*, *batch_leaf: str*, *controller: str*, *src_stem: str*, *dest_stem: str*) → None
: Generate a set of CSV files for use in intra-scenario graph generation.

  1 CSV per controller, for 2D/3D comparison types only. Because each CSV file corresponding to performance measures are 2D arrays, we actually just copy and rename the performance measure CSV files for each controllers into *cc_csv_root*.

  StackedSurfaceGraph expects an `_[0-9]+.csv` pattern for each 2D surfaces to graph in order to disambiguate which files belong to which controller without having the controller name in the filepath (contains dots), so we do that here. *Heatmap* does not require that, but for the heatmap set we generate it IS helpful to have an easy way to differentiate primary vs. other controllers, so we do it unconditionally here to handle both cases.

_gen_dual_heatmaps(*batch_leaf: str*, *criteria: sierra.core.variables.batch_criteria.BivarBatchCriteria*, *cmdopts: Dict[str, Any]*, *dest_stem: str*, *title: str*, *label: str*, *legend: List[str]*, *comp_type: str*) → None
: Generate a set of *DualHeatmap* graphs.

  Graphs contain all pairings of (primary controller, other), one per graph, within the specified scenario after input files have been gathered from each controller into *cc_csv_root*. Only valid if the comparison type is `HMraw`.

_gen_graph3D(*batch_leaf: str*, *criteria:* sierra.core.variables.batch_criteria.BivarBatchCriteria, *cmdopts: Dict[str, Any]*, *dest_stem: str*, *title: str*, *zlabel: str*, *legend: List[str]*, *comp_type: str*) → None

Generate a graph comparing the specified controllers within a scenario.

Graph contains the specified controllers within thes pecified scenario after input files have been gathered from each controllers into `cc_csv_root`.

_gen_graphs1D(*batch_leaf: str*, *criteria:* sierra.core.variables.batch_criteria.BivarBatchCriteria, *cmdopts: Dict[str, Any]*, *dest_stem: str*, *title: str*, *label: str*, *primary_axis: int*, *inc_exps: Optional[str]*, *legend: List[str]*) → None

_gen_graphs2D(*batch_leaf: str*, *criteria:* sierra.core.variables.batch_criteria.BivarBatchCriteria, *cmdopts: Dict[str, Any]*, *dest_stem: str*, *title: str*, *label: str*, *legend: List[str]*, *comp_type: str*) → None

_gen_paired_heatmaps(*batch_leaf: str*, *criteria:* sierra.core.variables.batch_criteria.BivarBatchCriteria, *cmdopts: Dict[str, Any]*, *dest_stem: str*, *title: str*, *label: str*, *comp_type: str*) → None

Generate a set of `Heatmap` graphs.

Uses a configured controller of primary interest against all other controllers (one graph per pairing), after input files have been gathered from each controller into `cc_csv_root`.

_gen_zaxis_label(*label: str*, *comp_type: str*) → str

If the comparison type is not "raw", put it on the graph as Z axis title.

_leaf_select(*candidate: str*) → bool

Determine if a controller can be included in the comparison for a scenario.

You can only compare controllers within the scenario directly generated from the value of `--batch-criteria`; other scenarios will (probably) cause file not found errors.

sierra.core.pipeline.stage5.pipeline_stage5

Stage 5 of the experimental pipeline: comparing deliverables.

- `PipelineStage5`: Compare controllers within or across scenarios.

class sierra.core.pipeline.stage5.pipeline_stage5.**PipelineStage5**(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*)

Compare controllers within or across scenarios.

This can be either:

1. Compare a set of controllers within the same scenario using performance measures specified in YAML configuration.
2. Compare a single controller across a set ofscenarios using performance measures specified in YAML configuration.

This stage is idempotent.

cmdopts
Dictionary of parsed cmdline parameters.

controllers
List of controllers to compare.

main_config
Dictionary of parsed main YAML configuration.

stage5_config
Dictionary of parsed stage5 YAML configuration.

output_roots
Dictionary containing output directories for intra- and inter-scenario graph generation.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.pipeline.stage5.pipeline_stage5', '__doc__': 'Compare controllers within or across scenarios.\n\n This can be either:\n\n #. Compare a set of controllers within the same scenario using performance\n measures specified in YAML configuration.\n\n #. Compare a single controller across a set ofscenarios using performance\n measures specified in YAML configuration.\n\n This stage is idempotent.\n\n Attributes:\n\n cmdopts: Dictionary of parsed cmdline parameters.\n\n controllers: List of controllers to compare.\n\n main_config: Dictionary of parsed main YAML configuration.\n\n stage5_config: Dictionary of parsed stage5 YAML configuration.\n\n output_roots: Dictionary containing output directories for intra- and\n inter-scenario graph generation.\n\n ', '__init__': <function PipelineStage5.__init__>, 'run': <function PipelineStage5.run>, '_run_cc': <function PipelineStage5._run_cc>, '_run_sc': <function PipelineStage5._run_sc>, '_verify_comparability': <function PipelineStage5._verify_comparability>, '__dict__': <attribute '__dict__' of 'PipelineStage5' objects>, '__weakref__': <attribute '__weakref__' of 'PipelineStage5' objects>, '__annotations__': {}})

__doc__ = 'Compare controllers within or across scenarios.\n\n This can be either:\n\n #. Compare a set of controllers within the same scenario using performance\n measures specified in YAML configuration.\n\n #. Compare a single controller across a set ofscenarios using performance\n measures specified in YAML configuration.\n\n This stage is idempotent.\n\n Attributes:\n\n cmdopts: Dictionary of parsed cmdline parameters.\n\n controllers: List of controllers to compare.\n\n main_config: Dictionary of parsed main YAML configuration.\n\n stage5_config: Dictionary of parsed stage5 YAML configuration.\n\n output_roots: Dictionary containing output directories for intra- and\n inter-scenario graph generation.\n\n '

__init__(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.core.pipeline.stage5.pipeline_stage5'

__weakref__
list of weak references to the object (if defined)

_run_cc(*cli_args*)

_run_sc(*cli_args*)

_verify_comparability(*controllers*, *cli_args*)
Check if the specified controllers can be compared.

Comparable controllers have all been run on the same set of batch experiments. If they have not, it is not *necessarily* an error, but probably should be looked at, so it is only a warning, not fatal.

run(*cli_args*) → None
Run stage 5 of the experimental pipeline.

If `--controller-comparison` was passed:

1. *UnivarIntraScenarioComparator* or *BivarIntraScenarioComparator* as appropriate, depending on which type of *BatchCriteria* was selected on the cmdline.

If `--scenario-comparison` was passed:

1. *UnivarInterScenarioComparator* (only valid for univariate batch criteria currently).

sierra.core.platform

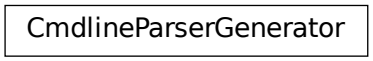
Terminal interface for pltaform plugins.

Classes for generating the commands to run *experiments* on multiple *platforms* using multiple execution methods.

- *CmdlineParserGenerator*: Dispatcher to generate additional platform-dependent cmdline arguments.
- *ExpRunShellCmdsGenerator*: Dispatcher for shell cmd generation for an *Experimental Run*.
- *ExpShellCmdsGenerator*: Dispatcher for shell cmd generation for an *Experiment*.
- *ParsedCmdlineConfigurer*: Dispatcher for configuring the cmdopts dictionary.
- *ExecEnvChecker*: Base class for verifying execution environments before running experiments.

class sierra.core.platform.**CmdlineParserGenerator**(*platform: str*)
Dispatcher to generate additional platform-dependent cmdline arguments.

Inheritance



__call__() → argparse.ArgumentParser
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.platform', '__doc__': '\n Dispatcher to generate additional platform-dependent cmdline arguments.\n ', '__init__': <function CmdlineParserGenerator.__init__>, '__call__': <function CmdlineParserGenerator.__call__>, '__dict__': <attribute '__dict__' of 'CmdlineParserGenerator' objects>, '__weakref__': <attribute '__weakref__' of 'CmdlineParserGenerator' objects>, '__annotations__': {}})

__doc__ = '\n Dispatcher to generate additional platform-dependent cmdline arguments.\n '

__init__(*platform: str*) → None

__module__ = 'sierra.core.platform'

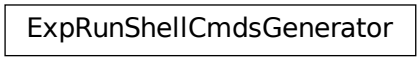
__weakref__
list of weak references to the object (if defined)

class sierra.core.platform.**ExpRunShellCmdsGenerator**(*cmdopts: Dict[str, Any]*, *criteria:* sierra.core.variables.batch_criteria.BatchCriteria, *n_robots: int*, *exp_num: int*)

Dispatcher for shell cmd generation for an *Experimental Run*.

Dispatches generation to the selected platform and execution environment. Called during stage 1 to add shell commands which should be run immediately before and after the shell command to actually execute a single *Experimental Run* to the commands file to be fed to whatever the tool a given execution environment environment uses to run cmds (e.g., GNU parallel).

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.platform', '__doc__': 'Dispatcher for shell cmd generation for an :term:`Experimental Run`.\n\n Dispatches generation to the selected platform and execution environment.\n Called during stage 1 to add shell commands which should be run immediately\n before and after the shell command to actually execute a single\n :term:`Experimental Run` to the commands file to be fed to whatever the tool\n a given execution environment environment uses to run cmds (e.g., GNU\n parallel).\n\n ', '__init__': <function ExpRunShellCmdsGenerator.__init__>, 'pre_run_cmds': <function ExpRunShellCmdsGenerator.pre_run_cmds>, 'exec_run_cmds': <function ExpRunShellCmdsGenerator.exec_run_cmds>, 'post_run_cmds': <function ExpRunShellCmdsGenerator.post_run_cmds>, '__dict__': <attribute '__dict__' of 'ExpRunShellCmdsGenerator' objects>, '__weakref__': <attribute '__weakref__' of 'ExpRunShellCmdsGenerator' objects>, '__annotations__': {}})

__doc__ = 'Dispatcher for shell cmd generation for an :term:`Experimental Run`.\n\n Dispatches generation to the selected platform and execution environment.\n Called during stage 1 to add shell commands which should be run immediately\n before and after the shell command to actually execute a single\n :term:`Experimental Run` to the commands file to be fed to whatever the tool\n a given execution environment environment uses to run cmds (e.g., GNU\n parallel).\n\n '

__init__(*cmdopts: Dict[str, Any]*, *criteria:* sierra.core.variables.batch_criteria.BatchCriteria, *n_robots: int*, *exp_num: int*) → None

__module__ = 'sierra.core.platform'

__weakref__
list of weak references to the object (if defined)

exec_run_cmds(*host: str*, *input_fpath: pathlib.Path*, *run_num: int*) → List[*sierra.core.types.ShellCmdSpec*]

post_run_cmds(*host: str*) → List[*sierra.core.types.ShellCmdSpec*]

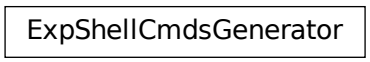
pre_run_cmds(*host: str*, *input_fpath: pathlib.Path*, *run_num: int*) → List[*sierra.core.types.ShellCmdSpec*]

class sierra.core.platform.**ExpShellCmdsGenerator**(*cmdopts: Dict[str, Any]*, *exp_num: int*)

Dispatcher for shell cmd generation for an *Experiment*.

Dispatches generation to the selected platform and execution environment. Called during stage 2 to run shell commands immediately before running a given *Experiment*, to run shell commands to actually run the experiment, and to run shell commands immediately after the experiment finishes.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.platform', '__doc__': 'Dispatcher for shell cmd generation for an :term:`Experiment`.\n\n Dispatches generation to the selected platform and execution environment.\n Called during stage 2 to run shell commands immediately before running a\n given :term:`Experiment`, to run shell commands to actually run the\n experiment, and to run shell commands immediately after the experiment\n finishes.\n\n ', '__init__': <function ExpShellCmdsGenerator.__init__>, 'pre_exp_cmds': <function ExpShellCmdsGenerator.pre_exp_cmds>, 'exec_exp_cmds': <function ExpShellCmdsGenerator.exec_exp_cmds>, 'post_exp_cmds': <function ExpShellCmdsGenerator.post_exp_cmds>, '__dict__': <attribute '__dict__' of 'ExpShellCmdsGenerator' objects>, '__weakref__': <attribute '__weakref__' of 'ExpShellCmdsGenerator' objects>, '__annotations__': {}})

__doc__ = 'Dispatcher for shell cmd generation for an :term:`Experiment`.\n\n Dispatches generation to the selected platform and execution environment.\n Called during stage 2 to run shell commands immediately before running a\n given :term:`Experiment`, to run shell commands to actually run the\n experiment, and to run shell commands immediately after the experiment\n finishes.\n\n '

__init__(*cmdopts: Dict[str, Any]*, *exp_num: int*) → None

__module__ = 'sierra.core.platform'

__weakref__

list of weak references to the object (if defined)

exec_exp_cmds(*exec_opts: Dict[str, str]*) → List[*sierra.core.types.ShellCmdSpec*]

post_exp_cmds() → List[*sierra.core.types.ShellCmdSpec*]

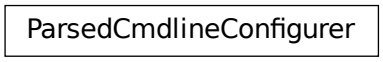
pre_exp_cmds() → List[*sierra.core.types.ShellCmdSpec*]

class sierra.core.platform.**ParsedCmdlineConfigurer**(*platform: str*, *exec_env: str*)

Dispatcher for configuring the cmdopts dictionary.

Dispatches configuring to the selected platform and execution environment. Called before the pipeline starts to add new/modify existing cmdline arguments after initial parsing.

Inheritance



__call__(*args: argparse.Namespace*) → argparse.Namespace
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.platform', '__doc__': 'Dispatcher for configuring the cmdopts dictionary.\n\n Dispatches configuring to the selected platform and execution environment.\n Called before the pipeline starts to add new/modify existing cmdline\n arguments after initial parsing.\n\n ', '__init__': <function ParsedCmdlineConfigurer.__init__>, '__call__': <function ParsedCmdlineConfigurer.__call__>, '__dict__': <attribute '__dict__' of 'ParsedCmdlineConfigurer' objects>, '__weakref__': <attribute '__weakref__' of 'ParsedCmdlineConfigurer' objects>, '__annotations__': {}})

__doc__ = 'Dispatcher for configuring the cmdopts dictionary.\n\n Dispatches configuring to the selected platform and execution environment.\n Called before the pipeline starts to add new/modify existing cmdline\n arguments after initial parsing.\n\n '

__init__(*platform: str*, *exec_env: str*) → None

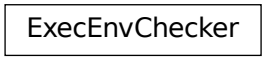
__module__ = 'sierra.core.platform'

__weakref__
list of weak references to the object (if defined)

class sierra.core.platform.**ExecEnvChecker**(*cmdopts: Dict[str, Any]*)
Base class for verifying execution environments before running experiments.

Platforms and/or execution environments needed to perform verification should derive from this class to use the common functionality present in it.

Inheritance



__call__() → None
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.platform', '__doc__': 'Base class for verifying execution environments before running experiments.\n\n Platforms and/or execution environments needed to perform verification\n should derive from this class to use the common functionality present in it.\n\n ', 'parse_nodefile': <staticmethod object>, '_parse_nodefile_line': <staticmethod object>, '__init__': <function ExecEnvChecker.__init__>, '__call__': <function ExecEnvChecker.__call__>, 'check_connectivity': <function ExecEnvChecker.check_connectivity>, 'check_for_simulator': <function ExecEnvChecker.check_for_simulator>, '__dict__': <attribute '__dict__' of 'ExecEnvChecker' objects>, '__weakref__': <attribute '__weakref__' of 'ExecEnvChecker' objects>, '__annotations__': {}})

__doc__ = 'Base class for verifying execution environments before running experiments.\n\n Platforms and/or execution environments needed to perform verification\n should derive from this class to use the common functionality present in it.\n\n '

__init__(*cmdopts: Dict[str, Any]*)

__module__ = 'sierra.core.platform'

__weakref__
list of weak references to the object (if defined)

static _parse_nodefile_line(*line: str*) → Optional[*sierra.core.types.ParsedNodefileSpec*]

check_connectivity(*login: str*, *hostname: str*, *port: int*, *host_type: str*) → None

check_for_simulator(*name: str*)

static parse_nodefile(*nodefile: str*) → List[*sierra.core.types.ParsedNodefileSpec*]

sierra.core.plugin

Sanity checks for verifying selected plugins.

Checks that selected plugins implement the necessary classes and functions. Currently checkes: `--storage-medium`, `--exec-env`, and `--platform`.

- *storage_sanity_checks()*: Check the selected `--storage-medium` plugin.
- *platform_sanity_checks()*: Check the selected `--platform` plugin.
- *exec_env_sanity_checks()*: Check the selected `--exec-env` plugin.

sierra.core.plugin.**storage_sanity_checks**(*module*) → None
Check the selected `--storage-medium` plugin.

sierra.core.plugin.**platform_sanity_checks**(*module*) → None
Check the selected `--platform` plugin.

sierra.core.plugin.**exec_env_sanity_checks**(*module*) → None
Check the selected `--exec-env` plugin.

sierra.core.plugin_manager

Simple plugin managers to make SIERRA OPEN/CLOSED.

- *module_exists()*: Check if a module exists before trying to import it.
- *module_load()*: Import the specified module.
- *bc_load()*: Load the specified *Batch Criteria*.
- *module_load_tiered()*: Attempt to load the specified python module with tiered precedence.

sierra.core.plugin_manager.**module_exists**(*name: str*) → bool
Check if a module exists before trying to import it.

sierra.core.plugin_manager.**module_load**(*name: str*) → module
Import the specified module.

sierra.core.plugin_manager.**bc_load**(*cmdopts: Dict[str, Any]*, *category: str*)
Load the specified *Batch Criteria*.

sierra.core.plugin_manager.**module_load_tiered**(*path: str*, *project: Optional[str] = None*, *platform: Optional[str] = None*) → module
Attempt to load the specified python module with tiered precedence.

Generally, the precedence is project -> project submodule -> platform module -> SIERRA core module, to allow users to override SIERRA core functionality with ease. Specifically:

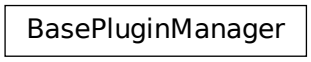
1. Check if the requested module is a project. If it is, return it.
2. Check if the requested module is a part of a project (i.e., `<project>.<path>` exists). If it does, return it. This requires that *SIERRA_PLUGIN_PATH* to be set properly.
3. Check if the requested module is provided by the platform plugin (i.e., `sierra.platform.<platform>.<path>` exists). If it does, return it.
4. Check if the requested module is part of the SIERRA core (i.e., `sierra.core.<path>` exists). If it does, return it.

If no match was found using any of these, throw an error.

- *BasePluginManager*: Base class for common functionality.
- *FilePluginManager*: Plugins are `.py` files within a root plugin directory.
- *DirectoryPluginManager*: Plugins are *directories* found in a root plugin directory.
- *ProjectPluginManager*: Plugins are *directories* found in a root plugin directory.
- *CompositePluginManager*: Base class for common functionality.

class sierra.core.plugin_manager.**BasePluginManager**
Base class for common functionality.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.plugin_manager', '__doc__': '\n Base class for common functionality.\n ', '__init__': <function BasePluginManager.__init__>, 'available_plugins': <function BasePluginManager.available_plugins>, 'load_plugin': <function BasePluginManager.load_plugin>, 'get_plugin': <function BasePluginManager.get_plugin>, 'get_plugin_module': <function BasePluginManager.get_plugin_module>, 'has_plugin': <function BasePluginManager.has_plugin>, '__dict__': <attribute '__dict__' of 'BasePluginManager' objects>, '__weakref__': <attribute '__weakref__' of 'BasePluginManager' objects>, '__annotations__': {'loaded': 'tp.Dict[str, tp.Dict]'}})

__doc__ = '\n Base class for common functionality.\n '

__init__() → None

__module__ = 'sierra.core.plugin_manager'

__weakref__
list of weak references to the object (if defined)

available_plugins()

get_plugin(*name: str*) → dict

get_plugin_module(*name: str*) → module

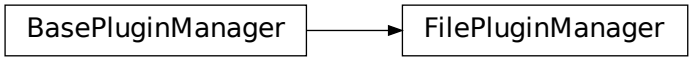
has_plugin(*name: str*) → bool

load_plugin(*name: str*) → None
Load a plugin module.

class sierra.core.plugin_manager.**FilePluginManager**
Plugins are `.py` files within a root plugin directory.

Intended for use with *models*.

Inheritance



__doc__ = 'Plugins are ``.py`` files within a root plugin directory.\n\n Intended for use with :term:`models <Model>`.\n\n '

__init__() → None

__module__ = 'sierra.core.plugin_manager'

available_plugins() → Dict[str, Dict]
Get the available plugins in the configured plugin root.

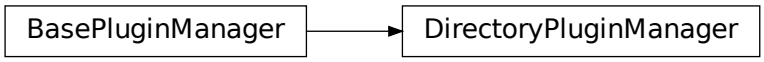
initialize(*project: str*, *search_root: pathlib.Path*) → None

loaded: Dict[str, Dict]

class sierra.core.plugin_manager.**DirectoryPluginManager**(*search_root: pathlib.Path*)
Plugins are *directories* found in a root plugin directory.

Intended for use with *Pipeline plugins*.

Inheritance



__doc__ = 'Plugins are `directories` found in a root plugin directory.\n\n Intended for use with :term:`Pipeline plugins <plugin>`.\n\n '

__init__(*search_root: pathlib.Path*) → None

__module__ = 'sierra.core.plugin_manager'

available_plugins()
Find all pipeline plugins in all directories within the search root.

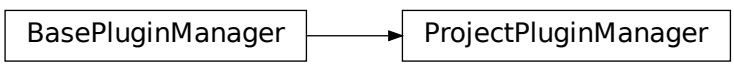
initialize(*project: str*) → None

loaded: Dict[str, Dict]

class sierra.core.plugin_manager.**ProjectPluginManager**(*search_root: pathlib.Path*, *project: str*)
Plugins are *directories* found in a root plugin directory.

Intended for use with *Project plugins*.

Inheritance



__doc__ = 'Plugins are `directories` found in a root plugin directory.\n\n Intended for use with :term:`Project plugins <plugin>`.\n\n '

__init__(*search_root: pathlib.Path*, *project: str*) → None

__module__ = 'sierra.core.plugin_manager'

available_plugins()

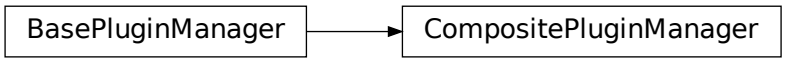
Find all pipeline plugins in all directories within the search root.

initialize(*project: str*) → None

loaded: Dict[str, Dict]

class sierra.core.plugin_manager.**CompositePluginManager**

Inheritance



__doc__ = None

__init__() → None

__module__ = 'sierra.core.plugin_manager'

available_plugins()

initialize(*project: str*, *search_path: List[pathlib.Path]*) → None

loaded: Dict[str, Dict]

sierra.core.root_dirpath_generator

Functions for generating root directory paths for a batch experiment.

- The batch experiment root. ALL files (inputs and outputs) are written to this directory, which will be under `--sierra-root`. Named using a combination of `--scenario` (block distribution + arena dimensions) and `--batch-criteria` in order to guarantee uniqueness among batch roots anytime the batch criteria or scenario change.
- The batch input root. All input files will be generated under this root directory. Named `<batch experiment root>/exp-inputs`.
- The batch output root. All output files will accrue under this root directory. Each experiment will get their own directory in this root for its outputs to accrue into. Named `<batch experiment root>/exp-outputs`.
- The batch graph root. All generated graphs will acrrue under this root directory. Each experiment will get their own directory in this root for their graphs to accrue into. Named `<batch experiment root>/graphs`.
- The batch model root. All model outputs will accrue under this root directory. Each experiment will get their own directory in this root for their model outputs to accrue into. Named `<batch experiment root>/models`.

- The batch statistics root. All statistics generated during stage 3 will accrue under this root directory. Each experiment will get their own directory in this root for their statistics. Named <batch experiment root>/statistics.
- The batch imagizing root. All images generated during stage 3 will accrue under this root directory. Each experiment will get their own directory in this root for their images. Named <batch experiment root>/images.
- The batch video root. All videos rendered during stage 4 will accrue under this root directory. Each experiment will get their own directory in this root for their videos. Named <batch experiment root>/videos.
- The batch scratch root. All GNU parallel outputs, --exec-env artifacts will appear under here. Each experiment will get their own directory in this root for their own scratch. This root is separate from experiment inputs to make checking for segfaults, tar-ing experiments, etc. easier. Named <batch experiment root>/scratch.
- *from_cmdline()*: Generate directory paths directly from cmdline arguments.
- *regen_from_exp()*: Regenerate directory paths from a previously created batch experiment.
- *parse_batch_leaf()*: Parse a batch root (dirpath leaf).
- *gen_batch_root()*: Generate the directory path for the batch root directory.

sierra.core.root_dirpath_generator.**from_cmdline**(*args: argparse.Namespace*) → Dict[str, pathlib.Path]
Generate directory paths directly from cmdline arguments.

sierra.core.root_dirpath_generator.**regen_from_exp**(*sierra_rpath: str*, *project: str*, *batch_leaf: str*, *controller: str*) → Dict[str, pathlib.Path]
Regenerate directory paths from a previously created batch experiment.

Parameters

- **sierra_rpath** – The path to the root directory where SIERRA should store everything.
- **project** – The name of the project plugin used.
- **criteria** – List of strings from the cmdline specification of the batch criteria.
- **batch_root** – The name of the directory that will be the root of the batch experiment (not including its parent).
- **controller** – The name of the controller used.

sierra.core.root_dirpath_generator.**parse_batch_leaf**(*root: str*) → Tuple[str, str, List[str]]
Parse a batch root (dirpath leaf).

Parsed into (template input file basename, scenario, batch criteria list) string components as they would have been specified on the cmdline.

sierra.core.root_dirpath_generator.**gen_batch_root**(*root: str*, *project: str*, *criteria: List[str]*, *scenario: str*, *controller: str*, *template_stem: str*) → pathlib.Path
Generate the directory path for the batch root directory.

The directory path depends on all of the input arguments to this function, and if ANY of the arguments change, so will the generated path.

Batch root is: <sierra_root>/<project>/<template_basename>-<scenario>+<criteria0>+<criteria1>

Parameters

- **root** – The path to the root directory where SIERRA should store everything.
- **project** – The name of the project plugin used.
- **criteria** – List of strings from the cmdline specification of the batch criteria.

- **scenario** – The cmdline specification of `--scenario`
- **batch_root** – The name of the directory that will be the root of the batch experiment (not including its parent).
- **controller** – The name of the controller used.

sierra.core.ros1

sierra.core.ros1.callbacks

Common classes and callbacks *Platforms* using *ROS1*.

- *population_size_from_pickle()*: Undocumented.
- *population_size_from_def()*: Undocumented.
- *robot_prefix_extract()*: Undocumented.

sierra.core.ros1.callbacks.**population_size_from_pickle**(*adds_def: Union[*sierra.core.experiment.xml.AttrChangeSet, sierra.core.experiment.xml.TagAddList*]*, *main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*) → int

sierra.core.ros1.callbacks.**population_size_from_def**(*exp_def:* sierra.core.experiment.definition.XMLExpDef, *main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*) → int

sierra.core.ros1.callbacks.**robot_prefix_extract**(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*) → str

sierra.core.ros1.cmdline

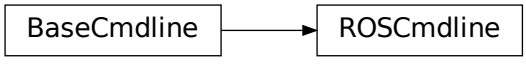
Common cmdline classes *Platforms* using *ROS1*.

- *ROSCmdline*: Defines *ROS1* common command line arguments.
- *ROSCmdlineValidator*: Perform checks on parsed ROS cmdline arguments.

class sierra.core.ros1.cmdline.**ROSCmdline**(*stages: List[int]*)

Defines *ROS1* common command line arguments.

Inheritance



__doc__ = 'Defines :term:`ROS1` common command line arguments.\n\n '

__init__(*stages: List[int]*) → None

__module__ = 'sierra.core.ros1.cmdline'

static cmdopts_update(*cli_args*, *cmdopts: Dict[str, Any]*) → None
Update cmdopts with ROS-specific cmdline options.

init_cli(*stages: List[int]*) → None

init_multistage() → None

init_stage1() → None

scaffold_cli() → None

class sierra.core.ros1.cmdline.**ROSCmdlineValidator**
Perform checks on parsed ROS cmdline arguments.

Inheritance

__call__(*args: argparse.Namespace*) → None
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.ros1.cmdline', '__doc__': '\n Perform checks on parsed ROS cmdline arguments.\n ', '__call__': <function ROSCmdlineValidator.__call__>, '__dict__': <attribute '__dict__' of 'ROSCmdlineValidator' objects>, '__weakref__': <attribute '__weakref__' of 'ROSCmdlineValidator' objects>, '__annotations__': {}})

__doc__ = '\n Perform checks on parsed ROS cmdline arguments.\n '

__module__ = 'sierra.core.ros1.cmdline'

__weakref__
list of weak references to the object (if defined)

sierra.core.ros1.generators

Classes for generating XML changes common to all *ROS1* platforms.

I.e., changes which are platform-specific, but applicable to all projects using ROS1.

- *ROSExpDefGenerator*: Generates XML changes to input files that common to all ROS experiments.
- *ROSExpRunDefUniqueGenerator*: Generate XML changes unique to a experimental runs for ROS experiments.

class sierra.core.ros1.generators.**ROSExpDefGenerator**(*exp_spec: sierra.core.experiment.spec.ExperimentSpec*, *controller: str*, *cmdopts: Dict[str, Any]*, ***kwargs*)

Generates XML changes to input files that common to all ROS experiments.

ROS1 requires up to 2 input files per run:

- The launch file containing robot definitions, world definitions (for simulations only).
- The parameter file for project code (optional).

Putting everything in 1 file would require extensively using the ROS1 parameter server which does NOT accept parameters specified in XML–only YAML. So requiring some conventions on the .launch input file seemed more reasonable.

controller
The controller used for the experiment.

cmdopts
Dictionary of parsed cmdline parameters.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.ros1.generators', '__doc__': 'Generates XML changes to input files that common to all ROS experiments.\n\n ROS1 requires up to 2 input files per run:\n\n - The launch file containing robot definitions, world definitions (for\n simulations only).\n\n - The parameter file for project code (optional).\n\n Putting everything in 1 file would require extensively using the ROS1\n parameter server which does NOT accept parameters specified in XML--only\n YAML. So requiring some conventions on the .launch input file seemed more\n reasonable.\n\n Attributes:\n\n controller: The controller used for the experiment.\n cmdopts: Dictionary of parsed cmdline parameters.\n\n ', '__init__': <function ROSExpDefGenerator.__init__>, 'generate': <function ROSExpDefGenerator.generate>, '_generate_experiment': <function ROSExpDefGenerator._generate_experiment>, '__dict__': <attribute '__dict__' of 'ROSExpDefGenerator' objects>, '__weakref__': <attribute '__weakref__' of 'ROSExpDefGenerator' objects>, '__annotations__': {}})

__doc__ = 'Generates XML changes to input files that common to all ROS experiments.\n\n ROS1 requires up to 2 input files per run:\n\n - The launch file containing robot definitions, world definitions (for\n simulations only).\n\n - The parameter file for project code (optional).\n\n Putting everything in 1 file would require extensively using the ROS1\n parameter server which does NOT accept parameters specified in XML--only\n YAML. So requiring some conventions on the .launch input file seemed more\n reasonable.\n\n Attributes:\n\n controller: The controller used for the experiment.\n cmdopts: Dictionary of parsed cmdline parameters.\n\n '

__init__(*exp_spec:* sierra.core.experiment.spec.ExperimentSpec, *controller: str*, *cmdopts: Dict[str, Any]*, ***kwargs*) → None

__module__ = 'sierra.core.ros1.generators'

__weakref__
list of weak references to the object (if defined)

_generate_experiment(*exp_def:* sierra.core.experiment.definition.XMLExpDef) → None
Generate XML tag changes to setup basic experiment parameters.

Writes generated changes to the simulation definition pickle file.

generate() → *sierra.core.experiment.definition.XMLExpDef*

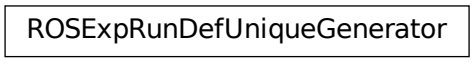
class sierra.core.ros1.generators.**ROSExpRunDefUniqueGenerator**(*run_num: int*, *run_output_path: pathlib.Path*, *launch_stem_path: pathlib.Path*, *random_seed: int*, *cmdopts: Dict[str, Any]*)

Generate XML changes unique to a experimental runs for ROS experiments.

These include:

- Random seeds for each: term: *Experimental Run*.
- Unique parameter file for each: term: *Experimental Run*.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.ros1.generators', '__doc__': '\n Generate XML changes unique to a experimental runs for ROS experiments.\n\n These include:\n\n - Random seeds for each: term: `Experimental Run`.\n\n - Unique parameter file for each: term: `Experimental Run`.\n ', '__init__': <function ROSExpRunDefUniqueGenerator.__init__>, 'generate': <function ROSExpRunDefUniqueGenerator.generate>, 'generate_random': <function ROSExpRunDefUniqueGenerator.generate_random>, 'generate_paramfile': <function ROSExpRunDefUniqueGenerator.generate_paramfile>, '__dict__': <attribute '__dict__' of 'ROSExpRunDefUniqueGenerator' objects>, '__weakref__': <attribute '__weakref__' of 'ROSExpRunDefUniqueGenerator' objects>, '__annotations__': {}})

__doc__ = '\n Generate XML changes unique to a experimental runs for ROS experiments.\n\n These include:\n\n - Random seeds for each: term: `Experimental Run`.\n\n - Unique parameter file for each: term: `Experimental Run`.\n '

__init__(*run_num: int*, *run_output_path: pathlib.Path*, *launch_stem_path: pathlib.Path*, *random_seed: int*, *cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.core.ros1.generators'

__weakref__
list of weak references to the object (if defined)

generate(*exp_def:* sierra.core.experiment.definition.XMLExpDef)

generate_paramfile(*exp_def:* sierra.core.experiment.definition.XMLExpDef) → None
Generate XML changes for the parameter file for an experimental run.

generate_random(*exp_def:* sierra.core.experiment.definition.XMLExpDef) → None
Generate XML changes for random seeding for an experimental run.

sierra.core.ros1.variables

sierra.core.ros1.variables.exp_setup

Classes for the `--exp-setup` cmdline option for ROS1 platforms.

See *Experiment Setup* for usage documentation.

- *ExpSetup*: Defines the experimental setup for ROS experiments.

class sierra.core.ros1.variables.exp_setup.**ExpSetup**(*n_secs_per_run: int*, *n_datapoints: int*, *n_ticks_per_sec: int*, *barrier_start: bool*, *robots_need_timekeeper: bool*)

Defines the experimental setup for ROS experiments.

n_secs_per_run
The *Experimental Run* duration in seconds, NOT *Ticks* or timesteps.

n_datapoints
How many datapoints to capture during the experimental run.

n_ticks_per_sec
How many times per second robot controllers will be run.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.ros1.variables.exp_setup', '__doc__': '\n Defines the experimental setup for ROS experiments.\n\n Attributes:\n\n n_secs_per_run: The :term:`Experimental Run` duration in seconds, NOT\n :term:`Ticks <Tick>` or timesteps.\n\n n_datapoints: How many datapoints to capture during the experimental\n run.\n\n n_ticks_per_sec: How many times per second robot controllers will be\n run.\n\n ', '__init__': <function ExpSetup.__init__>, 'gen_attr_changelist': <function ExpSetup.gen_attr_changelist>, 'gen_tag_rmlist': <function ExpSetup.gen_tag_rmlist>, 'gen_tag_addlist': <function ExpSetup.gen_tag_addlist>, 'gen_files': <function ExpSetup.gen_files>, '__dict__': <attribute '__dict__' of 'ExpSetup' objects>, '__weakref__': <attribute '__weakref__' of 'ExpSetup' objects>, '__annotations__': {}})

__doc__ = '\n Defines the experimental setup for ROS experiments.\n\n Attributes:\n\n n_secs_per_run: The :term:`Experimental Run` duration in seconds, NOT\n :term:`Ticks <Tick>` or timesteps.\n\n n_datapoints: How many datapoints to capture during the experimental\n run.\n\n n_ticks_per_sec: How many times per second robot controllers will be\n run.\n\n '

__init__(*n_secs_per_run: int*, *n_datapoints: int*, *n_ticks_per_sec: int*, *barrier_start: bool*, *robots_need_timekeeper: bool*) → None

__module__ = 'sierra.core.ros1.variables.exp_setup'

__weakref__
list of weak references to the object (if defined)

gen_attr_changelist() → List[*sierra.core.experiment.xml.AttrChangeSet*]

gen_files() → None

gen_tag_addlist() → List[*sierra.core.experiment.xml.TagAddList*]

gen_tag_rmlist() → List[*sierra.core.experiment.xml.TagRmList*]

sierra.core.startup

Startup checks performed by SIERRA.

Tests for compatibility and testing for the required packages in its environment.

sierra.core.stat_kernels

Kernels for the different types of statistics generated from experiments.

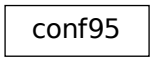
- *conf95*: Generate stddev statistics plotting for 95% confidence intervals.
- *mean*: Generate mean statistics only. Applicable to line graphs and heatmaps.
- *bw*: Generate statistics for plotting box and whisker plots around data points.

class sierra.core.stat_kernels.**conf95**
Generate stddev statistics plotting for 95% confidence intervals.

Applicable to:

- *StackedLineGraph*
- *SummaryLineGraph*

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.stat_kernels', '__doc__': 'Generate stddev statistics plotting for 95% confidence intervals.\n\n Applicable to:\n\n - :class:`~sierra.core.graphs.stacked_line_graph.StackedLineGraph`\n - :class:`~sierra.core.graphs.summary_line_graph.SummaryLineGraph`\n\n ', 'from_groupby': <staticmethod object>, 'from_pm': <staticmethod object>, '__dict__': <attribute '__dict__' of 'conf95' objects>, '__weakref__': <attribute '__weakref__' of 'conf95' objects>, '__annotations__': {}})

__doc__ = 'Generate stddev statistics plotting for 95% confidence intervals.\n\n Applicable to:\n\n - :class:`~sierra.core.graphs.stacked_line_graph.StackedLineGraph`\n - :class:`~sierra.core.graphs.summary_line_graph.SummaryLineGraph`\n\n '

__module__ = 'sierra.core.stat_kernels'

__weakref__
list of weak references to the object (if defined)

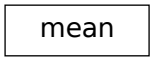
static from_groupby(*groupby: pandas.core.groupby.generic.DataFrameGroupBy*) → Dict[str, pandas.core.frame.DataFrame]

static from_pm(*dfs: Dict[str, pandas.core.frame.DataFrame]*) → Dict[str, pandas.core.frame.DataFrame]

class sierra.core.stat_kernels.**mean**

Generate mean statistics only. Applicable to line graphs and heatmaps.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.stat_kernels', '__doc__': '\n Generate mean statistics only. Applicable to line graphs and heatmaps.\n ', 'from_groupby': <staticmethod object>, 'from_pm': <staticmethod object>, '__dict__': <attribute '__dict__' of 'mean' objects>, '__weakref__': <attribute '__weakref__' of 'mean' objects>, '__annotations__': {}})

__doc__ = '\n Generate mean statistics only. Applicable to line graphs and heatmaps.\n '

__module__ = 'sierra.core.stat_kernels'

__weakref__
list of weak references to the object (if defined)

static from_groupby(*groupby: pandas.core.groupby.generic.DataFrameGroupBy*) → Dict[str, pandas.core.frame.DataFrame]

static from_pm(*dfs: Dict[str, pandas.core.frame.DataFrame]*) → Dict[str, pandas.core.frame.DataFrame]

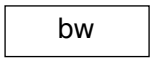
class sierra.core.stat_kernels.**bw**

Generate statistics for plotting box and whisker plots around data points.

Applicable to:

- *StackedLineGraph*
- *SummaryLineGraph*

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.stat_kernels', '__doc__': '\n Generate statistics for plotting box and whisker plots around data points.\n\n Applicable to:\n\n -
:class:`~sierra.core.graphs.stacked_line_graph.StackedLineGraph`\n -
:class:`~sierra.core.graphs.summary_line_graph.SummaryLineGraph`\n ',
'from_groupby': <staticmethod object>, 'from_pm': <staticmethod object>,
'__dict__': <attribute '__dict__' of 'bw' objects>, '__weakref__': <attribute
'__weakref__' of 'bw' objects>, '__annotations__': {}})

__doc__ = '\n Generate statistics for plotting box and whisker plots around data points.\n\n Applicable to:\n\n -
:class:`~sierra.core.graphs.stacked_line_graph.StackedLineGraph`\n -
:class:`~sierra.core.graphs.summary_line_graph.SummaryLineGraph`\n '

__module__ = 'sierra.core.stat_kernels'

__weakref__
list of weak references to the object (if defined)

static from_groupby(*groupby: pandas.core.groupby.generic.DataFrameGroupBy*) → Dict[str, pandas.core.frame.DataFrame]

static from_pm(*dfs: Dict[str, pandas.core.frame.DataFrame]*) → Dict[str, pandas.core.groupby.generic.DataFrameGroupBy]

sierra.core.storage

Terminal interfare for the various storage plugins that come with SIERRA.

See *Creating a New Storage Plugin* for more details.

- `DataFrameWriter`: Dispatcher to write a dataframe to the filesystem.
- `DataFrameReader`: Dispatcher to read a dataframe from the filesystem.

class sierra.core.storage.**DataFrameWriter**(*medium: str*)

Dispatcher to write a dataframe to the filesystem.

Inheritance

__call__(*df: pandas.core.frame.DataFrame*, *path: Union[pathlib.Path, str]*, ***kwargs*) → None

Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.storage', '__doc__': '\n Dispatcher to write a dataframe to the filesystem.\n ', '__init__': <function DataFrameWriter.__init__>, '__call__': <function DataFrameWriter.__call__>, '__dict__': <attribute '__dict__' of 'DataFrameWriter' objects>, '__weakref__': <attribute '__weakref__' of 'DataFrameWriter' objects>, '__annotations__': {}})

__doc__ = '\n Dispatcher to write a dataframe to the filesystem.\n '

__init__(*medium: str*)

__module__ = 'sierra.core.storage'

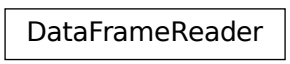
__weakref__

list of weak references to the object (if defined)

class sierra.core.storage.**DataFrameReader**(*medium: str*)

Dispatcher to read a dataframe from the filesystem.

Inheritance



__call__(*path: Union[pathlib.Path, str]*, ***kwargs*) → pandas.core.frame.DataFrame
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.storage', '__doc__': '\n Dispatcher to read a dataframe from the filesystem.\n\n ', '__init__': <function DataFrameReader.__init__>, '__call__': <function DataFrameReader.__call__>, '__dict__': <attribute '__dict__' of 'DataFrameReader' objects>, '__weakref__': <attribute '__weakref__' of 'DataFrameReader' objects>, '__annotations__': {}})

__doc__ = '\n Dispatcher to read a dataframe from the filesystem.\n\n '

__init__(*medium: str*)

__module__ = 'sierra.core.storage'

__weakref__
list of weak references to the object (if defined)

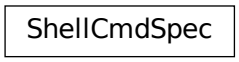
sierra.core.types

Custom types defined by SIERRA for more readable type hints.

- *ShellCmdSpec*: Undocumented.
- *YAMLConfigFileSpec*: Undocumented.
- *ParsedNodefileSpec*: Undocumented.
- *OSPackagesSpec*: Undocumented.

class sierra.core.types.**ShellCmdSpec**(*cmd: str*, *shell: bool*, *wait: bool*, *env: Optional[bool] = False*)

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.types', '__init__': <function ShellCmdSpec.__init__>, '__dict__': <attribute '__dict__' of 'ShellCmdSpec' objects>, '__weakref__': <attribute '__weakref__' of 'ShellCmdSpec' objects>, '__doc__': None, '__annotations__': {}})

__doc__ = None

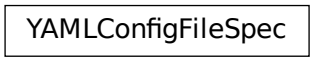
__init__(*cmd: str*, *shell: bool*, *wait: bool*, *env: Optional[bool] = False*) → None

__module__ = 'sierra.core.types'

__weakref__
list of weak references to the object (if defined)

class sierra.core.types.**YAMLConfigFileSpec**(*main: str*, *controllers: str*, *models: str*, *stage5: str*)

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.types', '__init__': <function YAMLConfigFileSpec.__init__>, '__dict__': <attribute '__dict__' of 'YAMLConfigFileSpec' objects>, '__weakref__': <attribute '__weakref__' of 'YAMLConfigFileSpec' objects>, '__doc__': None, '__annotations__': {}})

__doc__ = None

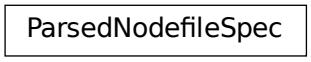
__init__(*main: str*, *controllers: str*, *models: str*, *stage5: str*) → None

__module__ = 'sierra.core.types'

__weakref__
list of weak references to the object (if defined)

class sierra.core.types.**ParsedNodefileSpec**(*hostname: str*, *n_cores: int*, *login: str*, *port: int*)

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.types', '__init__': <function ParsedNodefileSpec.__init__>, '__dict__': <attribute '__dict__' of 'ParsedNodefileSpec' objects>, '__weakref__': <attribute '__weakref__' of 'ParsedNodefileSpec' objects>, '__doc__': None, '__annotations__': {}})

__doc__ = None

__init__(*hostname: str*, *n_cores: int*, *login: str*, *port: int*) → None

__module__ = 'sierra.core.types'

__weakref__
list of weak references to the object (if defined)

class sierra.core.types.**OSPackagesSpec**(*kernel: str*, *name: str*, *pkgs: Dict[str, bool]*)

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.types', '__init__': <function OSPackagesSpec.__init__>, '__dict__': <attribute '__dict__' of 'OSPackagesSpec' objects>, '__weakref__': <attribute '__weakref__' of 'OSPackagesSpec' objects>, '__doc__': None, '__annotations__': {}})

__doc__ = None

__init__(*kernel: str*, *name: str*, *pkgs: Dict[str, bool]*) → None

__module__ = 'sierra.core.types'

__weakref__
list of weak references to the object (if defined)

sierra.core.utils

Miscellaneous bits used in mutiple places but that don't fit anywhere else.

- *dir_create_checked()*: Create a directory idempotently.
- *path_exists()*: Check if a path exists, trying multiple times.
- *get_primary_axis()*: Determine axis in a bivariate batch criteria is the primary axis.
- *exp_range_calc()*: Get the range of experiments to run/do stuff with. SUPER USEFUL.
- *exp_include_filter()*: Calculate which experiments to include in a calculation for something.
- *apply_to_expdef()*: Apply a generated XML modifictions to an experiment definition.
- *pickle_modifications()*: After applying XML modifications, pickle changes for later retrieval.
- *exp_template_path()*: Calculate the path to the template input file in the batch experiment root.
- *get_n_robots()*: Get the # robots used for a specific *Experiment*.
- *df_fill()*: Fill missing cells in a dataframe according to the specified fill policy.

sierra.core.utils.**dir_create_checked**(*path: Union[pathlib.Path, str]*, *exist_ok: bool*) → None
Create a directory idempotently.

If the directory exists and it shouldn't, raise an error.

sierra.core.utils.**path_exists**(*path: Union[pathlib.Path, str]*) → bool
Check if a path exists, trying multiple times.

This is necessary for working on HPC systems where if a given directory/filesystem is under heavy pressure the first check or two might time out as the FS goes and executes the query over the network.

sierra.core.utils.**get_primary_axis**(*criteria*, *primary_axis_bc: List*, *cmdopts: Dict[str, Any]*) → int
Determine axis in a bivariate batch criteria is the primary axis.

This is obtained on a per-query basis depending on the query context, or can be overriden on the cmdline.

sierra.core.utils.**exp_range_calc**(*cmdopts: Dict[str, Any]*, *root_dir: pathlib.Path*, *criteria*) → List[pathlib.Path]
Get the range of experiments to run/do stuff with. SUPER USEFUL.

sierra.core.utils.**exp_include_filter**(*inc_spec: Optional[str]*, *target: List*, *n_exps: int*)
Calculate which experiments to include in a calculation for something.

Take a input list of experiment numbers to include, and returns the sublist specified by the inc_spec (of the form [x:y]). inc_spec is an *absolute* specification; if a given performance measure excludes exp0 then that case is handled internally so that array/list shapes work out when generating graphs if this function is used consistently everywhere.

sierra.core.utils.**apply_to_expdef**(*var*, *exp_def:* sierra.core.experiment.definition.XMLExpDef) → Tuple[Optional[*sierra.core.experiment.xml.TagRmList*], Optional[*sierra.core.experiment.xml.TagAddList*], Optional[*sierra.core.experiment.xml.AttrChangeSet*]]
Apply a generated XML modifictions to an experiment definition.

In this order:

1. Remove existing XML tags
2. Add new XML tags
3. Change existing XML attributes

sierra.core.utils.**pickle_modifications**(*adds: Optional[*sierra.core.experiment.xml.TagAddList*]*, *chgs: Optional[*sierra.core.experiment.xml.AttrChangeSet*]*, *path: pathlib.Path*) → None
After applying XML modifications, pickle changes for later retrieval.

sierra.core.utils.**exp_template_path**(*cmdopts: Dict[str, Any]*, *batch_input_root: pathlib.Path*, *dirname: str*) → pathlib.Path
Calculate the path to the template input file in the batch experiment root.

The file at this path will be Used as the de-facto template for generating per-run input files.

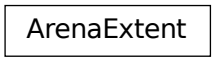
sierra.core.utils.**get_n_robots**(*main_config: Dict[str, Any]*, *cmdopts: Dict[str, Any]*, *exp_input_root: pathlib.Path*, *exp_def:* sierra.core.experiment.definition.XMLExpDef) → int
Get the # robots used for a specific *Experiment*.

sierra.core.utils.**df_fill**(*df: pandas.core.frame.DataFrame*, *policy: str*) → pandas.core.frame.DataFrame
Fill missing cells in a dataframe according to the specified fill policy.

- *ArenaExtent*: Representation of a 2D or 3D section/chunk/volume of the arena.
- *Sigmoid*: Sigmoid activation function.
- *ReLu*: Rectified Linear Unit (ReLU) activation function.

class sierra.core.utils.**ArenaExtent**(*dims:* sierra.core.vector.Vector3D, *origin:* sierra.core.vector.Vector3D = *(0, 0, 0)*)
Representation of a 2D or 3D section/chunk/volume of the arena.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.core.utils', '__doc__': 'Representation of a 2D or 3D section/chunk/volume of the arena.', 'from_corners': <staticmethod object>, '__init__': <function ArenaExtent.__init__>, 'contains': <function ArenaExtent.contains>, 'area': <function ArenaExtent.area>, 'xsize': <function ArenaExtent.xsize>, 'ysize': <function ArenaExtent.ysize>, 'zsize': <function ArenaExtent.zsize>, 'origin': <function ArenaExtent.origin>, '__str__': <function ArenaExtent.__str__>, '__dict__': <attribute '__dict__' of 'ArenaExtent' objects>, '__weakref__': <attribute '__weakref__' of 'ArenaExtent' objects>, '__annotations__': {}})

__doc__ = 'Representation of a 2D or 3D section/chunk/volume of the arena.'

__init__(*dims:* sierra.core.vector.Vector3D, *origin:* sierra.core.vector.Vector3D = *(0, 0, 0)*) → None

__module__ = 'sierra.core.utils'

__str__() → str
Return str(self).

__weakref__
list of weak references to the object (if defined)

area() → float

contains(*pt:* sierra.core.vector.Vector3D) → bool

static from_corners(*ll:* sierra.core.vector.Vector3D, *ur:* sierra.core.vector.Vector3D) → *sierra.core.utils.ArenaExtent*
Initialize an extent via LL and UR corners.

As opposed to an origin and a set of dimensions.

origin() → *sierra.core.vector.Vector3D*

xsize() → int

ysize() → int

zsize() → int

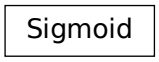
class sierra.core.utils.**Sigmoid**(*x: float*)

Sigmoid activation function.

$$f(x) = \tag{16.1}$$

rac{1}{1+e^{-x}}

Inheritance



__call__() → float
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.utils', '__doc__': '\n Sigmoid activation function.\n\n .. math::\n f(x) = \x0crac{1}{1+e^{-x}}\n\n ', '__init__': <function Sigmoid.__init__>, '__call__': <function Sigmoid.__call__>, '__dict__': <attribute '__dict__' of 'Sigmoid' objects>, '__weakref__': <attribute '__weakref__' of 'Sigmoid' objects>, '__annotations__': {}})

__doc__ = '\n Sigmoid activation function.\n\n .. math::\n f(x) = \x0crac{1}{1+e^{-x}}\n\n '

__init__(*x: float*) → None

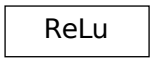
__module__ = 'sierra.core.utils'

__weakref__
list of weak references to the object (if defined)

class sierra.core.utils.**ReLu**(*x: float*)
Rectified Linear Unit (ReLU) activation function.

$$f(x) = max(0, x) = xifx > 0 \quad = 0else \tag{16.2}$$

Inheritance



__call__()
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.utils', '__doc__': '\n Rectified Linear Unit (ReLU) activation function.\n\n .. math::\n\n \\begin{aligned}\n f(x) = max(0,x) &= x \\textit{if} x > 0\n &= 0 \\textit{else}\n \\end{aligned}\n ', '__init__': <function ReLu.__init__>, '__call__': <function ReLu.__call__>, '__dict__': <attribute '__dict__' of 'ReLu' objects>, '__weakref__': <attribute '__weakref__' of 'ReLu' objects>, '__annotations__': {}})

__doc__ = '\n Rectified Linear Unit (ReLU) activation function.\n\n .. math::\n\n \\begin{aligned}\n f(x) = max(0,x) &= x \\textit{if} x > 0\n &= 0 \\textit{else}\n \\end{aligned}\n '

__init__(*x: float*)

__module__ = 'sierra.core.utils'

__weakref__
list of weak references to the object (if defined)

- *utf8open*

sierra.core.utils.**utf8open**
Explictly specify that the type of file being opened is UTF-8, which is should be for almost everything in SIERRA.

```
functools.partial(<built-in function open>, encoding='UTF-8')
```

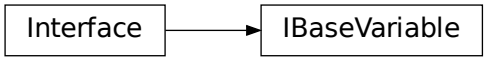
sierra.core.variables

sierra.core.variables.base_variable

- *IBaseVariable*: Interface that all variables must implement.

class sierra.core.variables.base_variable.**IBaseVariable**
Interface that all variables must implement.

Inheritance



__doc__ = 'Interface that all variables must implement.\n\n '

__module__ = 'sierra.core.variables.base_variable'

gen_attr_changelist() → List[*sierra.core.experiment.xml.AttrChangeSet*]
Generate XML attributes to change in a batch experiment definition.

Modifications are sets, one per experiment in the batch, because the order you apply them doesn't matter.

gen_files() → None
Generate one or more new files to add to the batch experiment definition.

Presumably, the created files will be referenced in the template input file by path.

gen_tag_addlist() → List[*sierra.core.experiment.xml.TagAddList*]
Generate XML tags to add to the batch experiment definition.

Modifications are lists, one per experiment in the batch, because the order you apply them matters.

gen_tag_rmlist() → List[*sierra.core.experiment.xml.TagRmList*]
Generate XML tags to remove from the batch experiment definition.

Modifications are lists, one per experiment in the batch, because the order you apply them matters.

sierra.core.variables.batch_criteria

Base classes used to define *Batch Experiments*.

- *BatchCriteria*: Defines experiments via lists of sets of changes to make to an XML file.
- *IConcreteBatchCriteria*: 'Final' interface for user-visible batch criteria.
- *UnivarBatchCriteria*: Base class for a univariate batch criteria.
- *BivarBatchCriteria*: Combination of the definition of two separate batch criteria.

class sierra.core.variables.batch_criteria.**BatchCriteria**(*cli_arg: str*, *main_config: Dict[str, Any]*, *batch_input_root: pathlib.Path*)
Defines experiments via lists of sets of changes to make to an XML file.

cli_arg
Unparsed batch criteria string from command line.

main_config
Parsed dictionary of main YAML configuration.

batch_input_root
Absolute path to the directory where batch experiment directories should be created.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.core.variables.batch_criteria', '__doc__': 'Defines experiments via lists of sets of changes to make to an XML file.\n\n Attributes:\n\n cli_arg: Unparsed batch criteria string from command line.\n\n main_config: Parsed dictionary of main YAML configuration.\n\n batch_input_root: Absolute path to the directory where batch experiment\n directories should be created.\n\n ', '__init__': <function BatchCriteria.__init__>, 'gen_attr_changelist': <function BatchCriteria.gen_attr_changelist>, 'gen_tag_rmlist': <function BatchCriteria.gen_tag_rmlist>, 'gen_tag_addlist': <function BatchCriteria.gen_tag_addlist>, 'gen_files': <function BatchCriteria.gen_files>, 'gen_exp_names': <function BatchCriteria.gen_exp_names>, 'arena_dims': <function BatchCriteria.arena_dims>, 'n_exp': <function BatchCriteria.n_exp>, 'pickle_exp_defs': <function BatchCriteria.pickle_exp_defs>, 'scaffold_exps': <function BatchCriteria.scaffold_exps>, '_scaffold_expi': <function BatchCriteria._scaffold_expi>, '__dict__': <attribute '__dict__' of 'BatchCriteria' objects>, '__weakref__': <attribute '__weakref__' of 'BatchCriteria' objects>, '__annotations__': {}})

__doc__ = 'Defines experiments via lists of sets of changes to make to an XML file.\n\n Attributes:\n\n cli_arg: Unparsed batch criteria string from command line.\n\n main_config: Parsed dictionary of main YAML configuration.\n\n batch_input_root: Absolute path to the directory where batch experiment\n directories should be created.\n\n '

__init__(*cli_arg: str*, *main_config: Dict[str, Any]*, *batch_input_root: pathlib.Path*) → None

__module__ = 'sierra.core.variables.batch_criteria'

__weakref__
list of weak references to the object (if defined)

_scaffold_expi(*expi_def:* sierra.core.experiment.definition.XMLExpDef, *modsi*, *is_compound: bool*, *i: int*, *cmdopts: Dict[str, Any]*) → None

arena_dims(*cmdopts: Dict[str, Any]*) → List[*sierra.core.utils.ArenaExtent*]
Get the arena dimensions used for each experiment in the batch.

Not applicable to all criteria.

Must be implemented on a per-platform basis, as different platforms have different means of computing the size of the arena.

gen_attr_changelist() → List[*sierra.core.experiment.xml.AttrChangeSet*]

gen_exp_names(*cmdopts: Dict[str, Any]*) → List[str]
Generate list of experiment names from the criteria.

Used for creating unique directory names for each experiment in the batch.

Returns List of experiments names for current experiment.

gen_files() → None

gen_tag_addlist() → List[*sierra.core.experiment.xml.TagAddList*]

gen_tag_rmlist() → List[*sierra.core.experiment.xml.TagRmList*]

n_exp() → int

pickle_exp_defs(*cmdopts: Dict[str, Any]*) → None

scaffold_exps(*batch_def:* sierra.core.experiment.definition.XMLExpDef, *cmdopts: Dict[str, Any]*) → None

Scaffold a batch experiment.

Takes the raw template input file and apply XML modifications from the batch criteria for all experiments, and save the result in each experiment's input directory.

class sierra.core.variables.batch_criteria.**IConcreteBatchCriteria**

'Final' interface for user-visible batch criteria.

Inheritance

__doc__ = "\n 'Final' interface for user-visible batch criteria.\n "

__module__ = 'sierra.core.variables.batch_criteria'

graph_xlabel(*cmdopts: Dict[str, Any]*) → str

Get the X-label for a graph.

Returns The X-label that should be used for the graphs of various performance measures across batch criteria.

graph_xticklabels(*cmdopts: Dict[str, Any]*, *exp_names: Optional[List[str]] = None*) → List[str]

Calculate X axis tick labels for graph generation.

Parameters

- **cmdopts** – Dictionary of parsed command line options.
- **exp_names** – If not None, then these directories will be used to calculate the labels, rather than the results of gen_exp_names().

graph_xticks(*cmdopts: Dict[str, Any]*, *exp_names: Optional[List[str]] = None*) → List[float]

Calculate X axis ticks for graph generation.

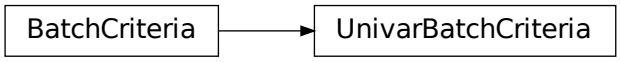
Parameters

- **cmdopts** – Dictionary of parsed command line options.
- **exp_names** – If not None, then this list of directories will be used to calculate the ticks, rather than the results of gen_exp_names().

class sierra.core.variables.batch_criteria.**UnivarBatchCriteria**(*cli_arg: str*, *main_config: Dict[str, Any]*, *batch_input_root: pathlib.Path*)

Base class for a univariate batch criteria.

Inheritance



__doc__ = '\n Base class for a univariate batch criteria.\n '

__module__ = 'sierra.core.variables.batch_criteria'

is_bivar() → bool

is_univar() → bool

populations(*cmdopts: Dict[str, Any]*, *exp_names: Optional[List[str]] = None*) → List[int]

Calculate system sizes used the batch experiment, sorted.

Parameters

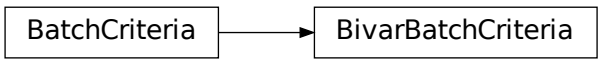
- **cmdopts** – Dictionary of parsed command line options.
- **exp_names** – If is not *None*, then these directories will be used to calculate the system sizes, rather than the results of `gen_exp_names()`.

class sierra.core.variables.batch_criteria.**BivarBatchCriteria**(*criteria1:* sierra.core.variables.batch_criteria.IConcreteBatchCri *criteria2:* sierra.core.variables.batch_criteria.IConcreteBatchCri

Combination of the definition of two separate batch criteria.

Changed in version 1.2.20: Bivariate batch criteria can be compound: one criteria can create and the other modify XML tags to create an experiment definition.

Inheritance



__doc__ = '\n Combination of the definition of two separate batch criteria.\n\n .. versionchanged:: 1.2.20\n\n Bivariate batch criteria can be compound: one criteria can create and\n the other modify XML tags to create an experiment definition.\n\n '

__init__(*criteria1:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria, *criteria2:* sierra.core.variables.batch_criteria.IConcreteBatchCriteria) → None

__module__ = 'sierra.core.variables.batch_criteria'

exp_scenario_name(*exp_num: int*) → str
Given the expeperiment number, compute a parsable scenario name.

It is necessary to query this function after generating the changelist in order to create generator classes for each experiment in the batch with the correct name and definition in some cases.

Can only be called if constant density is one of the sub-criteria.

gen_attr_changelist() → List[*sierra.core.experiment.xml.AttrChangeSet*]

gen_exp_names(*cmdopts: Dict[str, Any]*) → List[str]
Generate a SORTED list of strings for all experiment names.

These will be used as directory LEAF names–and don't include the parents.

gen_tag_addlist() → List[*sierra.core.experiment.xml.TagAddList*]

gen_tag_rmlist() → List[*sierra.core.experiment.xml.TagRmList*]

graph_xlabel(*cmdopts: Dict[str, Any]*) → str

graph_xticklabels(*cmdopts: Dict[str, Any]*, *exp_names: Optional[List[str]] = None*) → List[str]

graph_xticks(*cmdopts: Dict[str, Any]*, *exp_names: Optional[List[str]] = None*) → List[float]

graph_ylabel(*cmdopts: Dict[str, Any]*) → str

graph_yticklabels(*cmdopts: Dict[str, Any]*, *exp_names: Optional[List[str]] = None*) → List[str]

graph_yticks(*cmdopts: Dict[str, Any]*, *exp_names: Optional[List[str]] = None*) → List[float]

is_bivar() → bool

is_univar() → bool

n_robots(*exp_num: int*) → int

populations(*cmdopts: Dict[str, Any]*) → List[List[int]]
Generate a 2D array of system sizes used the batch experiment.

Sizes are in the same order as the directories returned from *gen_exp_names()* for each criteria along each axis.

set_batch_input_root(*root: pathlib.Path*) → None

sierra.core.variables.exp_setup

Reusable classes for configuring general aspects of experiments.

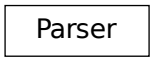
Aspects include experiment length, controller frequency, etc.

- *Parser*: Enforces the cmdline definition of `--exp-setup`.

class sierra.core.variables.exp_setup.**Parser**(*dflts: Dict[str, Union[str, int]]*)
Enforces the cmdline definition of `--exp-setup`.

See *Experiment Setup* for documentation.

Inheritance



__call__(*arg: str*) → Dict[str, Any]
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.variables.exp_setup', '__doc__': 'Enforces the cmdline definition of ``--exp-setup``.\n\n See :ref:`ln-sierra-vars-expsetup` for documentation.\n\n ', '__init__': <function Parser.__init__>, '__call__': <function Parser.__call__>, '__dict__': <attribute '__dict__' of 'Parser' objects>, '__weakref__': <attribute '__weakref__' of 'Parser' objects>, '__annotations__': {}})

__doc__ = 'Enforces the cmdline definition of ``--exp-setup``.\n\n See :ref:`ln-sierra-vars-expsetup` for documentation.\n\n '

__init__(*dflts: Dict[str, Union[str, int]]*)

__module__ = 'sierra.core.variables.exp_setup'

__weakref__
list of weak references to the object (if defined)

sierra.core.variables.population_size

Reusable classes related to the homogeneous populations of agents.

- *BasePopulationSize*: Base class for changing the # agents/robots to reduce code duplication.
- *Parser*: A base parser for use in changing the # robots/agents.

class sierra.core.variables.population_size.**BasePopulationSize**(**args*, ***kwargs*)
Base class for changing the # agents/robots to reduce code duplication.

Inheritance

__doc__ = '\n Base class for changing the # agents/robots to reduce code duplication.\n '

__init__(**args*, ***kwargs*) → None

__module__ = 'sierra.core.variables.population_size'

graph_xlabel(*cmdopts: Dict[str, Any]*) → str

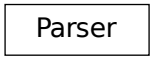
graph_xticklabels(*cmdopts: Dict[str, Any]*, *exp_names: Optional[List[str]] = None*) → List[str]

graph_xticks(*cmdopts: Dict[str, Any]*, *exp_names: Optional[List[str]] = None*) → List[float]

class sierra.core.variables.population_size.**Parser**

A base parser for use in changing the # robots/agents.

Inheritance



__call__(*arg: str*) → Dict[str, Any]

Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.core.variables.population_size', '__doc__': 'A base parser for use in changing the # robots/agents.\n\n ', '__call__': <function Parser.__call__>, 'to_sizes': <function Parser.to_sizes>, '__dict__': <attribute '__dict__' of 'Parser' objects>, '__weakref__': <attribute '__weakref__' of 'Parser' objects>, '__annotations__': {}})

__doc__ = 'A base parser for use in changing the # robots/agents.\n\n '

__module__ = 'sierra.core.variables.population_size'

__weakref__

list of weak references to the object (if defined)

to_sizes(*attr: Dict[str, Any]*) → List[int]

Generate the system sizes for each experiment in a batch.

sierra.core.variables.variable_density

- *VariableDensity*: A univariate range for variable density (# THINGS/m^2).
- *Parser*: Enforces specification of a *VariableDensity* derived batch criteria.

class sierra.core.variables.variable_density.**VariableDensity**(*cli_arg: str*, *main_config: Dict[str, Any]*, *batch_input_root: pathlib.Path*, *densities: List[float]*, *extent:* sierra.core.utils.ArenaExtent)

A univariate range for variable density (# THINGS/m^2).

THINGS is varied as arena size is held constant. This class is a base class which should NEVER be used on its own.

densities
List of densities to use.

dist_type
The type of block distribution to use.

changes
List of sets of changes to apply to generate the specified arena sizes.

Inheritance

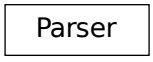
__doc__ = 'A univariate range for variable density (# THINGS/m^2).\n\n # THINGS is varied as arena size is held constant. This class is a base\n class which should NEVER be used on its own.\n\n Attributes:\n\n densities: List of densities to use.\n\n dist_type: The type of block distribution to use.\n\n changes: List of sets of changes to apply to generate the specified\n arena sizes.\n\n '

__init__(*cli_arg: str*, *main_config: Dict[str, Any]*, *batch_input_root: pathlib.Path*, *densities: List[float]*, *extent:* sierra.core.utils.ArenaExtent) → None

__module__ = 'sierra.core.variables.variable_density'

class sierra.core.variables.variable_density.**Parser**
Enforces specification of a *VariableDensity* derived batch criteria.

Inheritance



__call__(*arg: str*) → Dict[str, Any]
Parse the cmdline argument.

Returns density_min: Floating point value of target minimum density. density_max: Floating point value of target maximum density. cardinality: # densities in [min,max] that should be created.

Return type dict

__dict__ = mappingproxy({'__module__': 'sierra.core.variables.variable_density', '__doc__': 'Enforces specification of a :class:`VariableDensity` derived batch criteria.\n\n ', '__call__': <function Parser.__call__>, '_parse_density': <staticmethod object>, '__dict__': <attribute '__dict__' of 'Parser' objects>, '__weakref__': <attribute '__weakref__' of 'Parser' objects>, '__annotations__': {}})

__doc__ = 'Enforces specification of a :class:`VariableDensity` derived batch criteria.\n\n '

__module__ = 'sierra.core.variables.variable_density'

__weakref__
list of weak references to the object (if defined)

static _parse_density(*chunk: str*, *which: str*) → float

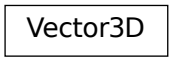
sierra.core.vector

Representation of vectors in 3D space and operations on them..

- *Vector3D*: Represents a point in 3D space and/or a directional vector in 3D space.

class sierra.core.vector.**Vector3D**(*x=0*, *y=0*, *z=0*)
Represents a point in 3D space and/or a directional vector in 3D space.

Inheritance



__add__(*o:* sierra.core.vector.Vector3D) → *sierra.core.vector.Vector3D*

__dict__ = mappingproxy({'__module__': 'sierra.core.vector', '__doc__': 'Represents a point in 3D space and/or a directional vector in 3D space.\n\n ', 'from_str': <staticmethod object>, 'd2norm': <staticmethod object>, '__init__': <function Vector3D.__init__>, '__eq__': <function Vector3D.__eq__>, '__hash__': <function Vector3D.__hash__>, '__len__': <function Vector3D.__len__>, '__add__': <function Vector3D.__add__>, '__sub__': <function Vector3D.__sub__>, '__mul__': <function Vector3D.__mul__>, '__truediv__': <function Vector3D.__truediv__>, '__iadd__': <function Vector3D.__iadd__>, '__isub__': <function Vector3D.__isub__>, '__ge__': <function Vector3D.__ge__>, '__le__': <function Vector3D.__le__>, '__lt__': <function Vector3D.__lt__>, '__neg__': <function Vector3D.__neg__>, '__str__': <function Vector3D.__str__>, '__repr__': <function Vector3D.__repr__>, 'length': <function Vector3D.length>, 'cross': <function Vector3D.cross>, 'dot': <function Vector3D.dot>, 'normalize': <function Vector3D.normalize>, 'perpendicularize': <function Vector3D.perpendicularize>, '__dict__': <attribute '__dict__' of 'Vector3D' objects>, '__weakref__': <attribute '__weakref__' of 'Vector3D' objects>, '__annotations__': {}})

__doc__ = 'Represents a point in 3D space and/or a directional vector in 3D space.\n\n '

__eq__(*other: object*) → bool
Return self==value.

__ge__(*other:* sierra.core.vector.Vector3D) → bool
Return self>=value.

__hash__() → int
Return hash(self).

__iadd__(*o:* sierra.core.vector.Vector3D) → *sierra.core.vector.Vector3D*

__init__(*x=0*, *y=0*, *z=0*)

__isub__(*o:* sierra.core.vector.Vector3D) → *sierra.core.vector.Vector3D*

__le__(*other:* sierra.core.vector.Vector3D) → bool
Return self<=value.

__len__() → int

__lt__(*other:* sierra.core.vector.Vector3D) → bool
Determine if one vector is less than another by coordinate comparison.

__module__ = 'sierra.core.vector'

__mul__(*o: Union[float, int]*) → *sierra.core.vector.Vector3D*

__neg__() → *sierra.core.vector.Vector3D*

__repr__() → str
Return repr(self).

__str__() → str
Return str(self).

__sub__(*o:* sierra.core.vector.Vector3D) → *sierra.core.vector.Vector3D*

__truediv__(*o: Union[float, int]*) → *sierra.core.vector.Vector3D*

__weakref__
list of weak references to the object (if defined)

cross(*rhs:* sierra.core.vector.Vector3D) → *sierra.core.vector.Vector3D*

static d2norm(*lhs:* sierra.core.vector.Vector3D, *rhs:* sierra.core.vector.Vector3D) → float

dot(*rhs:* sierra.core.vector.Vector3D) → *sierra.core.vector.Vector3D*

static from_str(*s: str*, *astype=<class 'int'>*) → *sierra.core.vector.Vector3D*

length() → float

normalize() → *sierra.core.vector.Vector3D*

perpendicularize() → *sierra.core.vector.Vector3D*

16.2 Plugins

16.2.1 sierra.plugins

sierra.plugins.hpc

sierra.plugins.hpc.adhoc

sierra.plugins.hpc.adhoc.plugin

HPC plugin for running experiments with an ad-hoc set of compute nodes.

E.g., whatever computers you happen to have laying around in the lab.

- *ParsedCmdlineConfigurer*: Configure SIERRA for ad-hoc HPC.
- *ExpShellCmdsGenerator*: Generate the cmd to invoke GNU Parallel in the ad-hoc HPC environment.

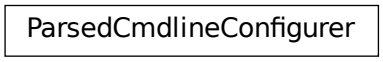
class sierra.plugins.hpc.adhoc.plugin.**ParsedCmdlineConfigurer**(*exec_env: str*)

Configure SIERRA for ad-hoc HPC.

May use the following environment variables:

- `SIERRA_NODEFILE` - If this is not defined `--nodefile` must be passed.

Inheritance



__call__(*args: argparse.Namespace*) → None

Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.plugins.hpc.adhoc.plugin', '__doc__': 'Configure SIERRA for ad-hoc HPC.\n\n May use the following environment variables:\n\n - ``SIERRA_NODEFILE`` - If this is not defined ``--nodefile`` must be\n passed.\n\n ', '__init__': <function ParsedCmdlineConfigurer.__init__>, '__call__': <function ParsedCmdlineConfigurer.__call__>, '__dict__': <attribute '__dict__' of 'ParsedCmdlineConfigurer' objects>, '__weakref__': <attribute '__weakref__' of 'ParsedCmdlineConfigurer' objects>, '__annotations__': {}})

__doc__ = 'Configure SIERRA for ad-hoc HPC.\n\n May use the following environment variables:\n\n - ``SIERRA_NODEFILE`` - If this is not defined ``--nodefile`` must be\n passed.\n\n '

__init__(*exec_env: str*) → None

__module__ = 'sierra.plugins.hpc.adhoc.plugin'

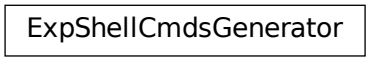
__weakref__

list of weak references to the object (if defined)

class sierra.plugins.hpc.adhoc.plugin.**ExpShellCmdsGenerator**(*cmdopts: Dict[str, Any]*, *exp_num: int*)

Generate the cmd to invoke GNU Parallel in the ad-hoc HPC environment.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.plugins.hpc.adhoc.plugin', '__doc__': 'Generate the cmd to invoke GNU Parallel in the ad-hoc HPC environment.\n ', '__init__': <function ExpShellCmdsGenerator.__init__>, 'pre_exp_cmds': <function ExpShellCmdsGenerator.pre_exp_cmds>, 'post_exp_cmds': <function ExpShellCmdsGenerator.post_exp_cmds>, 'exec_exp_cmds': <function ExpShellCmdsGenerator.exec_exp_cmds>, '__dict__': <attribute '__dict__' of 'ExpShellCmdsGenerator' objects>, '__weakref__': <attribute '__weakref__' of 'ExpShellCmdsGenerator' objects>, '__annotations__': {}})

__doc__ = 'Generate the cmd to invoke GNU Parallel in the ad-hoc HPC environment.\n '

__init__(*cmdopts: Dict[str, Any]*, *exp_num: int*) → None

__module__ = 'sierra.plugins.hpc.adhoc.plugin'

__weakref__
list of weak references to the object (if defined)

exec_exp_cmds(*exec_opts: Dict[str, str]*) → List[*sierra.core.types.ShellCmdSpec*]

post_exp_cmds() → List[*sierra.core.types.ShellCmdSpec*]

pre_exp_cmds() → List[*sierra.core.types.ShellCmdSpec*]

sierra.plugins.hpc.local

sierra.plugins.hpc.local.plugin

HPC plugin for running SIERRA locally.

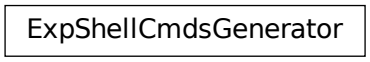
Not necessarily HPC, but it fits well enough under that semantic umbrella.

- *ExpShellCmdsGenerator*: Generate the command to invoke GNU parallel for local HPC.

class sierra.plugins.hpc.local.plugin.**ExpShellCmdsGenerator**(*cmdopts: Dict[str, Any]*, *exp_num: int*)

Generate the command to invoke GNU parallel for local HPC.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.plugins.hpc.local.plugin', '__doc__': '\n Generate the command to invoke GNU parallel for local HPC.\n ', '__init__': <function ExpShellCmdsGenerator.__init__>, 'pre_exp_cmds': <function ExpShellCmdsGenerator.pre_exp_cmds>, 'post_exp_cmds': <function ExpShellCmdsGenerator.post_exp_cmds>, 'exec_exp_cmds': <function ExpShellCmdsGenerator.exec_exp_cmds>, '__dict__': <attribute '__dict__' of 'ExpShellCmdsGenerator' objects>, '__weakref__': <attribute '__weakref__' of 'ExpShellCmdsGenerator' objects>, '__annotations__': {}})

__doc__ = '\n Generate the command to invoke GNU parallel for local HPC.\n '

__init__(*cmdopts: Dict[str, Any]*, *exp_num: int*) → None

__module__ = 'sierra.plugins.hpc.local.plugin'

__weakref__
list of weak references to the object (if defined)

exec_exp_cmds(*exec_opts: Dict[str, str]*) → List[*sierra.core.types.ShellCmdSpec*]

post_exp_cmds() → List[*sierra.core.types.ShellCmdSpec*]

pre_exp_cmds() → List[*sierra.core.types.ShellCmdSpec*]

sierra.plugins.hpc.pbs

sierra.plugins.hpc.pbs.plugin

HPC plugin for running SIERRA on HPC clusters using the TORQUE-PBS scheduler.

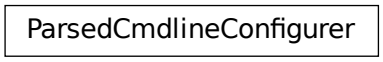
- *ParsedCmdlineConfigurer*: Configure SIERRA for PBS HPC.

class sierra.plugins.hpc.pbs.plugin.**ParsedCmdlineConfigurer**(*exec_env: str*)
Configure SIERRA for PBS HPC.

Uses the following environment variables (if any of them are not defined an assertion will be triggered):

- PBS_NUM_PPN
- PBS_NODEFILE
- PBS_JOBID

Inheritance



__call__(*args: argparse.Namespace*) → None
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.plugins.hpc.pbs.plugin', '__doc__': 'Configure SIERRA for PBS HPC.\n\n Uses the following environment variables (if any of them are not defined an\n assertion will be triggered):\n\n - ``PBS_NUM_PPN``\n - ``PBS_NODEFILE``\n - ``PBS_JOBID``\n\n ', '__init__': <function ParsedCmdlineConfigurer.__init__>, '__call__': <function ParsedCmdlineConfigurer.__call__>, '__dict__': <attribute '__dict__' of 'ParsedCmdlineConfigurer' objects>, '__weakref__': <attribute '__weakref__' of 'ParsedCmdlineConfigurer' objects>, '__annotations__': {}})

__doc__ = 'Configure SIERRA for PBS HPC.\n\n Uses the following environment variables (if any of them are not defined an\n assertion will be triggered):\n\n - ``PBS_NUM_PPN``\n - ``PBS_NODEFILE``\n - ``PBS_JOBID``\n\n '

__init__(*exec_env: str*) → None

__module__ = 'sierra.plugins.hpc.pbs.plugin'

__weakref__
list of weak references to the object (if defined)

sierra.plugins.hpc.slurm

sierra.plugins.hpc.slurm.plugin

HPC plugin for running SIERRA on HPC clusters using the SLURM scheduler.

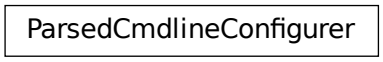
- *ParsedCmdlineConfigurer*: Configure SIERRA for SLURM HPC.
- *ExpShellCmdsGenerator*: Generate the cmd to correctly invoke GNU Parallel on SLURM HPC.

class sierra.plugins.hpc.slurm.plugin.**ParsedCmdlineConfigurer**(*exec_env: str*)
Configure SIERRA for SLURM HPC.

Uses the following environment variables (if any of them are not defined an assertion will be triggered):

- SLURM_CPUS_PER_TASK
- SLURM_TASKS_PER_NODE
- SLURM_JOB_NODELIST
- SLURM_JOB_ID

Inheritance



__call__(*args: argparse.Namespace*) → None
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.plugins.hpc.slurm.plugin', '__doc__': 'Configure SIERRA for SLURM HPC.\n\n Uses the following environment variables (if any of them are not defined an\n assertion will be triggered):\n\n - ``SLURM_CPUS_PER_TASK``\n - ``SLURM_TASKS_PER_NODE``\n - ``SLURM_JOB_NODELIST``\n - ``SLURM_JOB_ID``\n\n ', '__init__': <function ParsedCmdlineConfigurer.__init__>, '__call__': <function ParsedCmdlineConfigurer.__call__>, '__dict__': <attribute '__dict__' of 'ParsedCmdlineConfigurer' objects>, '__weakref__': <attribute '__weakref__' of 'ParsedCmdlineConfigurer' objects>, '__annotations__': {}})

__doc__ = 'Configure SIERRA for SLURM HPC.\n\n Uses the following environment variables (if any of them are not defined an\n assertion will be triggered):\n\n - ``SLURM_CPUS_PER_TASK``\n - ``SLURM_TASKS_PER_NODE``\n - ``SLURM_JOB_NODELIST``\n - ``SLURM_JOB_ID``\n\n '

__init__(*exec_env: str*) → None

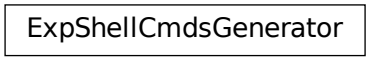
__module__ = 'sierra.plugins.hpc.slurm.plugin'

__weakref__
list of weak references to the object (if defined)

class sierra.plugins.hpc.slurm.plugin.**ExpShellCmdsGenerator**(*cmdopts: Dict[str, Any]*, *exp_num: int*)

Generate the cmd to correctly invoke GNU Parallel on SLURM HPC.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.plugins.hpc.slurm.plugin', '__doc__': 'Generate the cmd to correctly invoke GNU Parallel on SLURM HPC.\n\n ', '__init__': <function ExpShellCmdsGenerator.__init__>, 'pre_exp_cmds': <function ExpShellCmdsGenerator.pre_exp_cmds>, 'post_exp_cmds': <function ExpShellCmdsGenerator.post_exp_cmds>, 'exec_exp_cmds': <function ExpShellCmdsGenerator.exec_exp_cmds>, '__dict__': <attribute '__dict__' of 'ExpShellCmdsGenerator' objects>, '__weakref__': <attribute '__weakref__' of 'ExpShellCmdsGenerator' objects>, '__annotations__': {}})

__doc__ = 'Generate the cmd to correctly invoke GNU Parallel on SLURM HPC.\n\n '

__init__(*cmdopts: Dict[str, Any]*, *exp_num: int*) → None

__module__ = 'sierra.plugins.hpc.slurm.plugin'

__weakref__
list of weak references to the object (if defined)

exec_exp_cmds(*exec_opts: Dict[str, str]*) → List[*sierra.core.types.ShellCmdSpec*]

post_exp_cmds() → List[*sierra.core.types.ShellCmdSpec*]

pre_exp_cmds() → List[*sierra.core.types.ShellCmdSpec*]

sierra.plugins.platform

sierra.plugins.platform.argos

sierra.plugins.platform.argos.cmdline

Command line parsing and validation for the *ARGoS*.

sierra.plugins.platform.argos.generators

sierra.plugins.platform.argos.generators.platform_generators

Classes for generating common XML modifications for *ARGoS*.

I.e., changes which are platform-specific, but applicable to all projects using the platform.

- *PlatformExpDefGenerator*: Init the object.
- *PlatformExpRunDefUniqueGenerator*: Generate XML changes unique to each experimental run.

class sierra.plugins.platform.argos.generators.platform_generators.**PlatformExpDefGenerator**(*exp_spec: sierra.core.experiment.controller: str*, *cmdopts: Dict[str, Any]*, ***kwargs*)

Init the object.

controller
The controller used for the experiment.

cmdopts
Dictionary of parsed cmdline parameters.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.plugins.platform.argos.generators.platform_generators', '__doc__': '\n Init the object.\n\n Attributes:\n\n controller: The controller used for the experiment.\n\n cmdopts: Dictionary of parsed cmdline parameters.\n ', '__init__': <function PlatformExpDefGenerator.__init__>, 'generate': <function PlatformExpDefGenerator.generate>, 'generate_physics': <function PlatformExpDefGenerator.generate_physics>, 'generate_arena_shape': <function PlatformExpDefGenerator.generate_arena_shape>, '_generate_n_robots': <function PlatformExpDefGenerator._generate_n_robots>, '_generate_saa': <function PlatformExpDefGenerator._generate_saa>, '_generate_time': <function PlatformExpDefGenerator._generate_time>, '_generate_threading': <function PlatformExpDefGenerator._generate_threading>, '_generate_library': <function PlatformExpDefGenerator._generate_library>, '_generate_visualization': <function PlatformExpDefGenerator._generate_visualization>, '__dict__': <attribute '__dict__' of 'PlatformExpDefGenerator' objects>, '__weakref__': <attribute '__weakref__' of 'PlatformExpDefGenerator' objects>, '__annotations__': {}})

__doc__ = '\n Init the object.\n\n Attributes:\n\n controller: The controller used for the experiment.\n\n cmdopts: Dictionary of parsed cmdline parameters.\n '

__init__(*exp_spec:* sierra.core.experiment.spec.ExperimentSpec, *controller: str*, *cmdopts: Dict[str, Any]*, ***kwargs*) → None

__module__ = 'sierra.plugins.platform.argos.generators.platform_generators'

__weakref__
list of weak references to the object (if defined)

_generate_library(*exp_def:* sierra.core.experiment.definition.XMLExpDef) → None
Generate XML changes for ARGoS search paths for controller,loop functions.

Set to the name of the plugin passed on the cmdline, unless overriden in configuration. The __CONTROLLER__ tag is changed during stage 1, but since this function is called as part of common def generation, it happens BEFORE that, and so this is OK. If, for some reason that assumption becomes invalid, a warning will be issued about a non-existent XML path, so it won't be a silent error.

Does not write generated changes to the simulation definition pickle file.

_generate_n_robots(*exp_def:* sierra.core.experiment.definition.XMLExpDef) → None
Generate XML changes to setup # robots (if specified on cmdline).

Writes generated changes to the simulation definition pickle file.

_generate_saa(*exp_def:* sierra.core.experiment.definition.XMLExpDef) → None
Generate XML changes to disable selected sensors/actuators.

Some sensors and actuators are computationally expensive in large populations, but not that costly if the # robots is small.

Does not write generated changes to the simulation definition pickle file.

_generate_threading(*exp_def:* sierra.core.experiment.definition.XMLExpDef) → None
Generate XML changes to set the # of cores for a simulation to use.

This may be less than the total # available on the system, depending on the experiment definition and user preferences.

Does not write generated changes to the simulation definition pickle file.

_generate_time(*exp_def:* sierra.core.experiment.definition.XMLExpDef) → None
Generate XML changes to setup simulation time parameters.

Writes generated changes to the simulation definition pickle file.

_generate_visualization(*exp_def:* sierra.core.experiment.definition.XMLExpDef) → None
Generate XML changes to remove visualization elements from input file.

This depends on cmdline parameters, as visualization definitions should be left in if ARGoS should output simulation frames for video creation.

Does not write generated changes to the simulation definition pickle file.

generate() → *sierra.core.experiment.definition.XMLExpDef*
Generate XML modifications common to all ARGoS experiments.

generate_arena_shape(*exp_def:* sierra.core.experiment.definition.XMLExpDef, *shape:* sierra.plugins.platform.argos.variables.arena_shape.ArenaShape) → None
Generate XML changes for the specified arena shape.

Writes generated changes to the simulation definition pickle file.

generate_physics(*exp_def:* sierra.core.experiment.definition.XMLExpDef, *cmdopts: Dict[str, Any]*, *engine_type: str*, *n_engines: int*, *extents: List[*sierra.core.utils.ArenaExtent*]*, *remove_defs: bool = True*) → None
Generate XML changes for the specified physics engines configuration.

Physics engine definition removal is optional, because when mixing 2D/3D engine definitions, you only want to remove the existing definitions BEFORE you have adding first of the mixed definitions. Doing so every time results in only the LAST of the mixed definitions being present in the input file.

Does not write generated changes to the simulation definition pickle file.

class sierra.plugins.platform.argos.generators.platform_generators.**PlatformExpRunDefUniqueGenerator**(*run_int*, *run_path*, *lib.F launpath*, *lib.F ran-dom int*, *cm-dopt Dict Any*)

Generate XML changes unique to each experimental run.

These include:

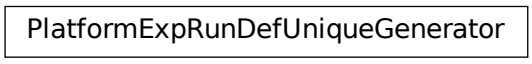
- Random seeds for each simulation.

run_num
The runulation # in the experiment.

run_output_path
Path to simulation output directory within experiment root.

cmdopts
Dictionary containing parsed cmdline options.

Inheritance




```
__dict__ = mappingproxy({'__module__':
'sierra.plugins.platform.argos.generators.platform_generators', '__doc__':
'Generate XML changes unique to each experimental run.\n\n These include:\n\n -
Random seeds for each simulation.\n\n Attributes:\n\n run_num:  The runulation # in
the experiment.\n\n run_output_path:  Path to simulation output directory within
experiment\n root.\n\n cmdopts:  Dictionary containing parsed cmdline options.\n\n
', '__init__':  <function PlatformExpRunDefUniqueGenerator.__init__>,
'_PlatformExpRunDefUniqueGenerator__generate_random':  <function
PlatformExpRunDefUniqueGenerator.__generate_random>, 'generate':  <function
PlatformExpRunDefUniqueGenerator.generate>,
'_PlatformExpRunDefUniqueGenerator__generate_visualization':  <function
PlatformExpRunDefUniqueGenerator.__generate_visualization>, '__dict__':  <attribute
'__dict__' of 'PlatformExpRunDefUniqueGenerator' objects>, '__weakref__':
<attribute '__weakref__' of 'PlatformExpRunDefUniqueGenerator' objects>,
'__annotations__':  {}})
```

__doc__ = 'Generate XML changes unique to each experimental run.\n\n These include:\n\n - Random seeds for each simulation.\n\n Attributes:\n\n run_num: The runulation # in the experiment.\n\n run_output_path: Path to simulation output directory within experiment\n root.\n\n cmdopts: Dictionary containing parsed cmdline options.\n\n '

__generate_random(*exp_def*) → None
Generate XML changes for random seeding for a specific simulation.

__generate_visualization(*exp_def:* sierra.core.experiment.definition.XMLExpDef)
Generate XML changes for setting up rendering for a specific simulation.

__init__(*run_num: int*, *run_output_path: pathlib.Path*, *launch_stem_path: pathlib.Path*, *random_seed: int*, *cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.plugins.platform.argos.generators.platform_generators'

__weakref__
list of weak references to the object (if defined)

generate(*exp_def:* sierra.core.experiment.definition.XMLExpDef)

sierra.plugins.platform.argos.plugin

sierra.plugins.platform.argos.variables

sierra.plugins.platform.argos.variables.arena_shape

- *ArenaShape*: Maps a list of desired arena dimensions sets of XML changes.

class sierra.plugins.platform.argos.variables.arena_shape.**ArenaShape**(*extents: List[*sierra.core.utils.ArenaExtent*]*)

Maps a list of desired arena dimensions sets of XML changes.

This class is a base class which should (almost) never be used on its own. Instead, derived classes defined in this file should be used instead.

extents
List of arena extents.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.plugins.platform.argos.variables.arena_shape', '__doc__': 'Maps a list of desired arena dimensions sets of XML changes.\n\n This class is a base class which should (almost) never be used on its\n own. Instead, derived classes defined in this file should be used instead.\n\n Attributes:\n\n extents: List of arena extents.\n\n ', '__init__': <function ArenaShape.__init__>, 'gen_attr_changelist': <function ArenaShape.gen_attr_changelist>, '_gen_chgs_for_extent': <function ArenaShape._gen_chgs_for_extent>, 'gen_tag_rmlist': <function ArenaShape.gen_tag_rmlist>, 'gen_tag_addlist': <function ArenaShape.gen_tag_addlist>, 'gen_files': <function ArenaShape.gen_files>, '__dict__': <attribute '__dict__' of 'ArenaShape' objects>, '__weakref__': <attribute '__weakref__' of 'ArenaShape' objects>, '__annotations__': {'attr_changes': 'tp.List[xml.AttrChangeSet]'}})

__doc__ = 'Maps a list of desired arena dimensions sets of XML changes.\n\n This class is a base class which should (almost) never be used on its\n own. Instead, derived classes defined in this file should be used instead.\n\n Attributes:\n\n extents: List of arena extents.\n\n '

__init__(*extents: List[*sierra.core.utils.ArenaExtent*]*) → None

__module__ = 'sierra.plugins.platform.argos.variables.arena_shape'

__weakref__
list of weak references to the object (if defined)

_gen_chgs_for_extent(*extent:* sierra.core.utils.ArenaExtent) → *sierra.core.experiment.xml.AttrChangeSet*

gen_attr_changelist() → List[*sierra.core.experiment.xml.AttrChangeSet*]
Generate changes necessary setup ARGoS with the specified arena sizes.

gen_files() → None

gen_tag_addlist() → List[*sierra.core.experiment.xml.TagAddList*]

gen_tag_rmlist() → List[*sierra.core.experiment.xml.TagRmList*]

- *kWALL_WIDTH*

sierra.plugins.platform.argos.variables.arena_shape.**kWALL_WIDTH**
Convert a string or number to a floating point number, if possible.

```
0.4
```

sierra.plugins.platform.argos.variables.cameras

Classes for specifying ARGoS cameras.

Positions, timeline, and interpolation, for manipulating the frame capture/rendering perspective.

- *QTCameraTimeline*: Defines when/how to switch between camera perspectives within ARGoS.
- *QTCameraOverhead*: Defines a single overhead camera perspective within ARGoS.

class sierra.plugins.platform.argos.variables.cameras.**QTCameraTimeline**(*setup:* sierra.plugins.platform.argos.variables.exp *cmdline: str*, *extents:* *List[*sierra.core.utils.ArenaExtent*]*)

Defines when/how to switch between camera perspectives within ARGoS.

interpolate
Should we interpolate between camera positions on our timeline ?

setup
Simulation experiment definitions.

extents
List of (X,Y,Zs) tuple of dimensions of arena areas to generate camera definitions for.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.plugins.platform.argos.variables.cameras', '__doc__': 'Defines when/how to switch between camera perspectives within ARGoS.\n\n Attributes:\n\n interpolate: Should we interpolate between camera positions on our\n timeline ?\n\n setup: Simulation experiment definitions.\n\n extents: List of (X,Y,Zs) tuple of dimensions of arena areas to generate\n camera definitions for.\n\n ', 'kARGOS_N_CAMERAS': 12, '__init__': <function QTCameraTimeline.__init__>, 'gen_attr_changelist': <function QTCameraTimeline.gen_attr_changelist>, 'gen_tag_rmlist': <function QTCameraTimeline.gen_tag_rmlist>, 'gen_tag_addlist': <function QTCameraTimeline.gen_tag_addlist>, 'gen_files': <function QTCameraTimeline.gen_files>, '_gen_keyframes': <function QTCameraTimeline._gen_keyframes>, '_gen_camera_config': <function QTCameraTimeline._gen_camera_config>, '__dict__': <attribute '__dict__' of 'QTCameraTimeline' objects>, '__weakref__': <attribute '__weakref__' of 'QTCameraTimeline' objects>, '__annotations__': {'tag_adds': 'tp.List[xml.TagAddList]'}})

__doc__ = 'Defines when/how to switch between camera perspectives within ARGoS.\n\n Attributes:\n\n interpolate: Should we interpolate between camera positions on our\n timeline ?\n\n setup: Simulation experiment definitions.\n\n extents: List of (X,Y,Zs) tuple of dimensions of arena areas to generate\n camera definitions for.\n\n '

__init__(*setup:* sierra.plugins.platform.argos.variables.exp_setup.ExpSetup, *cmdline:* *str*, *extents:* *List*[sierra.core.utils.ArenaExtent]) → None

__module__ = 'sierra.plugins.platform.argos.variables.cameras'

__weakref__
list of weak references to the object (if defined)

_gen_camera_config(*ext:* sierra.core.utils.ArenaExtent, *index:* *int*, *n_cameras*) → tuple

_gen_keyframes(*adds:* sierra.core.experiment.xml.TagAddList, *n_cameras:* *int*, *cycle_length:* *int*) → None

gen_attr_changelist() → List[*sierra.core.experiment.xml.AttrChangeSet*]
No effect.

All tags/attributes are either deleted or added.

gen_files() → None

gen_tag_addlist() → List[*sierra.core.experiment.xml.TagAddList*]

gen_tag_rmlist() → List[*sierra.core.experiment.xml.TagRmList*]
Remove the <camera> tag if it exists.

Obviously you *must* call this function BEFORE adding new definitions.

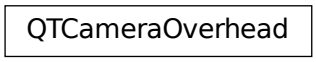
kARGOS_N_CAMERAS = 12

class sierra.plugins.platform.argos.variables.cameras.**QTCameraOverhead**(*extents: List[*sierra.core.utils.ArenaExtent*]*)

Defines a single overhead camera perspective within ARGoS.

extents
List of (X,Y,Z) tuple of dimensions of arena areas to generate camera definitions for.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.plugins.platform.argos.variables.cameras', '__doc__': 'Defines a single overhead camera perspective within ARGoS.\n\n Attributes:\n\n extents: List of (X,Y,Z) tuple of dimensions of arena areas to generate\n camera definitions for.\n\n ', '__init__': <function QTCameraOverhead.__init__>, 'gen_attr_changelist': <function QTCameraOverhead.gen_attr_changelist>, 'gen_tag_rmlist': <function QTCameraOverhead.gen_tag_rmlist>, 'gen_tag_addlist': <function QTCameraOverhead.gen_tag_addlist>, 'gen_files': <function QTCameraOverhead.gen_files>, '__dict__': <attribute '__dict__' of 'QTCameraOverhead' objects>, '__weakref__': <attribute '__weakref__' of 'QTCameraOverhead' objects>, '__annotations__': {'tag_adds': 'tp.List[xml.TagAddList]'}})

__doc__ = 'Defines a single overhead camera perspective within ARGoS.\n\n Attributes:\n\n extents: List of (X,Y,Z) tuple of dimensions of arena areas to generate\n camera definitions for.\n\n '

__init__(*extents: List[*sierra.core.utils.ArenaExtent*]*) → None

__module__ = 'sierra.plugins.platform.argos.variables.cameras'

__weakref__
list of weak references to the object (if defined)

gen_attr_changelist() → List[*sierra.core.experiment.xml.AttrChangeSet*]
No effect.

All tags/attributes are either deleted or added.

gen_files() → None

gen_tag_addlist() → List[*sierra.core.experiment.xml.TagAddList*]

gen_tag_rmlist() → List[*sierra.core.experiment.xml.TagRmList*]
Remove the <camera> tag if it exists.

Obviously you *must* call this function BEFORE adding new definitions.

sierra.plugins.platform.argos.variables.constant_density

- *ConstantDensity*: Defines common functionality for all constant-density classes.

class sierra.plugins.platform.argos.variables.constant_density.**ConstantDensity**(*cli_arg: str*, *main_config: Dict[str, Any]*, *batch_input_root: pathlib.Path*, *target_density: float*, *dimensions: List[*sierra.core.utils.ArenaExte*sce-*
nario_tag: str)

Defines common functionality for all constant-density classes.

Constant density = SOMETHING/arena size is held constant as arena size is increased. This class is a base class which should NEVER be used on its own.

target_density
The target density.

dimensions
List of (X,Y) dimensions to use (creates rectangular arenas).

scenario_tag
A scenario tag (presumably part of *–scenario*) to use to generate scenario names.

changes
List of sets of changes to apply to generate the specified arena sizes.

Inheritance

__doc__ = 'Defines common functionality for all constant-density classes.\n\n Constant density = SOMETHING/arena size is held constant as arena size is\n increased. This class is a base class which should NEVER be used on its own.\n\n Attributes:\n\n target_density: The target density.\n\n dimensions: List of (X,Y) dimensions to use (creates rectangular\n arenas).\n\n scenario_tag: A scenario tag (presumably part of `--scenario`) to use to\n generate scenario names.\n\n changes: List of sets of changes to apply to generate the specified\n arena sizes.\n\n '

__init__(*cli_arg: str*, *main_config: Dict[str, Any]*, *batch_input_root: pathlib.Path*, *target_density: float*, *dimensions: List[*sierra.core.utils.ArenaExtent*]*, *scenario_tag: str*) → None

__module__ = 'sierra.plugins.platform.argos.variables.constant_density'

exp_scenario_name(*exp_num: int*) → str

Given the exp number in the batch, compute a parsable scenario name.

It is necessary to query this criteria after generating the changelist in order to create generator classes for each experiment in the batch with the correct name and definition in some cases.

Normally controller+scenario are used to look up all necessary changes for the specified arena size, but for this criteria the specified scenario is the base scenario (i.e., the starting arena dimensions), and the correct arena dimensions for a given exp must be found via lookup with THIS function).

sierra.plugins.platform.argos.variables.exp_setup

Classes for the `--exp-setup` cmdline option.

See *Experiment Setup* for usage documentation.

- *ExpSetup*: Defines the simulation duration.

class sierra.plugins.platform.argos.variables.exp_setup.**ExpSetup**(*n_secs_per_run: int*, *n_datapoints: int*, *n_ticks_per_sec: int*)

Defines the simulation duration.

duration

The simulation duration in seconds, NOT timesteps.

Inheritance

__dict__ = mappingproxy({'__module__': 'sierra.plugins.platform.argos.variables.exp_setup', '__doc__': '\n Defines the simulation duration.\n\n Attributes:\n\n duration: The simulation duration in seconds, NOT timesteps.\n ', 'extract_time_params': <staticmethod object>, '__init__': <function ExpSetup.__init__>, 'gen_attr_changelist': <function ExpSetup.gen_attr_changelist>, 'gen_tag_rmlist': <function ExpSetup.gen_tag_rmlist>, 'gen_tag_addlist': <function ExpSetup.gen_tag_addlist>, 'gen_files': <function ExpSetup.gen_files>, '__dict__': <attribute '__dict__' of 'ExpSetup' objects>, '__weakref__': <attribute '__weakref__' of 'ExpSetup' objects>, '__annotations__': {'attr_changes': 'tp.List[xml.AttrChangeSet]'}})

__doc__ = '\n Defines the simulation duration.\n\n Attributes:\n\n duration: The simulation duration in seconds, NOT timesteps.\n '

__init__(*n_secs_per_run: int*, *n_datapoints: int*, *n_ticks_per_sec: int*) → None

__module__ = 'sierra.plugins.platform.argos.variables.exp_setup'

__weakref__
list of weak references to the object (if defined)

static extract_time_params(*exp_def:* sierra.core.experiment.xml.AttrChangeSet) → Dict[str, int]
Extract and return time parameters for the experiment.

Returns length (in seconds), ticks_per_second

gen_attr_changelist() → List[*sierra.core.experiment.xml.AttrChangeSet*]

gen_files() → None

gen_tag_addlist() → List[*sierra.core.experiment.xml.TagAddList*]

gen_tag_rmlist() → List[*sierra.core.experiment.xml.TagRmList*]

sierra.plugins.platform.argos.variables.physics_engines

Classes mapping an extent to a set of non-overlapping ARGoS physics engines.

Extent does not have to be the whole arena. 2D and 3D engines.

- *PhysicsEngines*: Defines 2D/3D physics engines within ARGoS and how they are laid out.
- *PhysicsEngines2D*: Specialization of *PhysicsEngines* for 2D.
- *PhysicsEngines3D*: Specialization of *PhysicsEngines* for 3D.

class sierra.plugins.platform.argos.variables.physics_engines.**PhysicsEngines**(*engine_type: str*, *n_engines: int*, *iter_per_tick: int*, *layout: str*, *extents: List[*sierra.core.utils.ArenaExtent*]*)

Defines 2D/3D physics engines within ARGoS and how they are laid out.

engine_type
The type of physics engine to use (one supported by ARGoS).

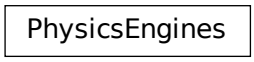
n_engines
# of engines. Can be one of [1,4,8,16,24].

iter_per_tick
: # of iterations physics engines should perform per tick. layout: Engine arrangement method. Can be one of:

  - `uniform_grid2D`: Arrange the engines in a uniform 2D grid that extends up to the maximum height in Z. For 2D engines, this is the maximum height of objects that can be present in the arena (I think).

extents
: List of (X,Y,Zs) tuple of dimensions of area to assign to engines of the specified type.

Inheritance




```
__dict__ = mappingproxy({'__module__':
'sierra.plugins.platform.argos.variables.physics_engines', '__doc__':  'Defines
2D/3D physics engines within ARGoS and how they are laid out.\n\n Attributes:\n\n
engine_type:  The type of physics engine to use (one supported by ARGoS).\n\n
n_engines:  # of engines. Can be one of [1,4,8,16,24].\n\n iter_per_tick:  # of
iterations physics engines should perform per tick.\n layout:  Engine arrangement
method. Can be one of:\n\n - ``uniform_grid2D``:  Arrange the engines in a uniform
2D\n grid that extends up to the maximum height in Z. For 2D\n engines, this is the
maximum height of objects that can\n be present in the arena (I think).\n\n extents:
List of (X,Y,Zs) tuple of dimensions of area to assign to\n engines of the specified
type.\n\n ', '__init__':  <function PhysicsEngines.__init__>, 'gen_attr_changelist':
<function PhysicsEngines.gen_attr_changelist>, 'gen_tag_rmlist':  <function
PhysicsEngines.gen_tag_rmlist>, 'gen_tag_addlist':  <function
PhysicsEngines.gen_tag_addlist>, 'gen_files':  <function PhysicsEngines.gen_files>,
'_gen_all_engines':  <function PhysicsEngines._gen_all_engines>,
'gen_single_engine':  <function PhysicsEngines.gen_single_engine>, '_gen1_engines':
<function PhysicsEngines._gen1_engines>, '_gen2_engines':  <function
PhysicsEngines._gen2_engines>, '_gen4_engines':  <function
PhysicsEngines._gen4_engines>, '_gen6_engines':  <function
PhysicsEngines._gen6_engines>, '_gen8_engines':  <function
PhysicsEngines._gen8_engines>, '_gen12_engines':  <function
PhysicsEngines._gen12_engines>, '_gen16_engines':  <function
PhysicsEngines._gen16_engines>, '_gen24_engines':  <function
PhysicsEngines._gen24_engines>, '_gen_engine_name':  <function
PhysicsEngines._gen_engine_name>, '_gen_engine_name_stem':  <function
PhysicsEngines._gen_engine_name_stem>, '__dict__':  <attribute '__dict__' of
'PhysicsEngines' objects>, '__weakref__':  <attribute '__weakref__' of
'PhysicsEngines' objects>, '__annotations__':  {'tag_adds':
'tp.List[xml.TagAddList]'}})
```

__doc__ = 'Defines 2D/3D physics engines within ARGoS and how they are laid out.\n\n Attributes:\n\n engine_type: The type of physics engine to use (one supported by ARGoS).\n\n n_engines: # of engines. Can be one of [1,4,8,16,24].\n\n iter_per_tick: # of iterations physics engines should perform per tick.\n layout: Engine arrangement method. Can be one of:\n\n - ``uniform_grid2D``: Arrange the engines in a uniform 2D\n grid that extends up to the maximum height in Z. For 2D\n engines, this is the maximum height of objects that can\n be present in the arena (I think).\n\n extents: List of (X,Y,Zs) tuple of dimensions of area to assign to\n engines of the specified type.\n\n '

__init__(*engine_type: str*, *n_engines: int*, *iter_per_tick: int*, *layout: str*, *extents: List[*sierra.core.utils.ArenaExtent*]*) → None

__module__ = 'sierra.plugins.platform.argos.variables.physics_engines'

__weakref__
list of weak references to the object (if defined)

_gen12_engines(*extent:* sierra.core.utils.ArenaExtent) → *sierra.core.experiment.xml.TagAddList*
Generate definitions for 12 physics engines for the specified extent.

The 2D layout is:

8 9 10 11 7 6 5 4 0 1 2 3

Volume is *NOT* divided equally among 3D engines, but rather each of the engines is extended up to some maximum height in Z, forming a set of "silos".

_gen16_engines(*extent:* sierra.core.utils.ArenaExtent) → *sierra.core.experiment.xml.TagAddList*
Generate definitions for 16 physics engines for the specified extent.

The 2D layout is:

15 14 13 12 8 9 10 11 7 6 5 4 0 1 2 3

Volume is *NOT* divided equally among 3D engines, but rather each of the engines is extended up to some maximum height in Z, forming a set of "silos".

_gen1_engines() → *sierra.core.experiment.xml.TagAddList*
Generate definitions for 1 physics engine for the specified extent.

_gen24_engines(*extent:* sierra.core.utils.ArenaExtent) → *sierra.core.experiment.xml.TagAddList*
Generate definitions for 16 physics engines for the specified extent.

The 2D layout is:

23 22 21 20 19 18 12 13 14 15 16 17 11 10 9 8 7 6 0 1 2 3 4 5

Volume is *NOT* divided equally among 3D engines, but rather each of the engines is extended up to some maximum height in Z, forming a set of "silos".

_gen2_engines(*extent:* sierra.core.utils.ArenaExtent) → *sierra.core.experiment.xml.TagAddList*
Generate definitions for 2 physics engines for the specified extents.

Engines are layed out as follows in 2D, regardless if they are 2D or 3D engines:

0 1

Volume is *NOT* divided equally among 3D engines, but rather each of the engines is extended up to some maximum height in Z, forming a set of "silos".

_gen4_engines(*extent:* sierra.core.utils.ArenaExtent) → *sierra.core.experiment.xml.TagAddList*
Generate definitions for 4 physics engines for the specified extent.

Engines are layed out as follows in 2D, regardless if they are 2D or 3D engines:

3 2 0 1

Volume is *NOT* divided equally among 3D engines, but rather each of the engines is extended up to some maximum height in Z, forming a set of "silos".

_gen6_engines(*extent:* sierra.core.utils.ArenaExtent) → *sierra.core.experiment.xml.TagAddList*
Generate definitions for 6 physics engines for the specified extent.

Engines are layed out as follows in 2D, regardless if they are 2D or 3D engines:

5 4 3 0 1 2

Volume is *NOT* divided equally among 3D engines, but rather each of the engines is extended up to some maximum height in Z, forming a set of "silos".

_gen8_engines(*extent:* sierra.core.utils.ArenaExtent) → *sierra.core.experiment.xml.TagAddList*
Generate definitions for 8 physics engines for the specified extent.

The 2D layout is:

7 6 5 4 0 1 2 3

Volume is *NOT* divided equally among 3D engines, but rather each of the engines is extended up to some maximum height in Z, forming a set of "silos".

_gen_all_engines(*extent:* sierra.core.utils.ArenaExtent, *n_engines_x: int*, *n_engines_y: int*, *forward_engines: List[int]*) → *sierra.core.experiment.xml.TagAddList*
Generate definitions for the specified # of engines for the extent.

_gen_engine_name(*engine_id: int*) → str
Generate the unique ID for an engine.

ID is comprised of a type + numeric identifier of the engine.

_gen_engine_name_stem() → str
Generate the name stem for the specified engine type.

gen_attr_changelist() → List[*sierra.core.experiment.xml.AttrChangeSet*]
No effect.

All tags/attributes are either deleted or added.

gen_files() → None

gen_single_engine(*engine_id: int*, *extent:* sierra.core.utils.ArenaExtent, *n_engines_x: int*, *n_engines_y: int*, *forward_engines: List[int]*) → *sierra.core.experiment.xml.TagAddList*
Generate definitions for a specific 2D/3D engine.

Volume is *NOT* divided equally among engines, but rather each of the engines is extended up to some maximum height in Z, forming a set of "silos".

Parameters

- **engine_id** – Numerical UUID for the engine.
- **extent** – The mapped extent for ALL physics engines.
- **exceptions** – List of lists of points defining polygons which should NOT be managed by any of the engines currently being processed.
- **n_engines_x** – # engines in the x direction.
- **n_engines_y** – # engines in the y direction.
- **forward_engines** – IDs of engines that are placed in increasing order in X when layed out L->R.

gen_tag_addlist() → List[*sierra.core.experiment.xml.TagAddList*]

gen_tag_rmlist() → List[*sierra.core.experiment.xml.TagRmList*]

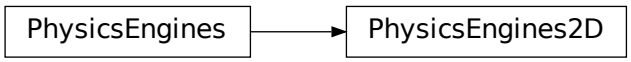
Remove the <physics_engines> tag if it exists may be desirable.

Obviously you *must* call this function BEFORE adding new definitions.

class sierra.plugins.platform.argos.variables.physics_engines.**PhysicsEngines2D**(*engine_type: str*, *n_engines: int*, *spatial_hash_info: Optional[Dict[str, Any]]*, *iter_per_tick: int*, *layout: str*, *extents: List[*sierra.core.utils.ArenaExtent*]*)

Specialization of *PhysicsEngines* for 2D.

Inheritance



__doc__ = '\n Specialization of :class:`PhysicsEngines` for 2D.\n '

__init__(*engine_type: str*, *n_engines: int*, *spatial_hash_info: Optional[Dict[str, Any]]*, *iter_per_tick: int*, *layout: str*, *extents: List[*sierra.core.utils.ArenaExtent*]*) → None

__module__ = 'sierra.plugins.platform.argos.variables.physics_engines'

gen_single_engine(*engine_id: int*, *extent:* sierra.core.utils.ArenaExtent, *n_engines_x: int*, *n_engines_y: int*, *forward_engines: List[int]*) → *sierra.core.experiment.xml.TagAddList*

Generate definitions for a specific 2D/3D engine.

Volume is *NOT* divided equally among engines, but rather each of the engines is extended up to some maximum height in Z, forming a set of "silos".

Parameters

- **engine_id** – Numerical UUID for the engine.
- **extent** – The mapped extent for ALL physics engines.
- **exceptions** – List of lists of points defining polygons which should NOT be managed by any of the engines currently being processed.
- **n_engines_x** – # engines in the x direction.
- **n_engines_y** – # engines in the y direction.

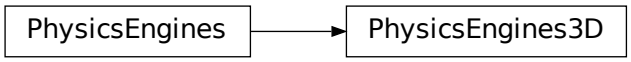
- **forward_engines** – IDs of engines that are placed in increasing order in X when layed out L->R.

tag_adds: List[*sierra.core.experiment.xml.TagAddList*]

class sierra.plugins.platform.argos.variables.physics_engines.**PhysicsEngines3D**(*engine_type: str*, *n_engines: int*, *iter_per_tick: int*, *layout: str*, *extents: List[*sierra.core.utils.ArenaExte

Specialization of *PhysicsEngines* for 3D.

Inheritance



__doc__ = '\n Specialization of :class:`PhysicsEngines` for 3D.\n '

__init__(*engine_type: str*, *n_engines: int*, *iter_per_tick: int*, *layout: str*, *extents: List[*sierra.core.utils.ArenaExtent*]*) → None

__module__ = 'sierra.plugins.platform.argos.variables.physics_engines'

tag_adds: List[*sierra.core.experiment.xml.TagAddList*]

sierra.plugins.platform.argos.variables.population_constant_density

Classes for the constant population density batch criteria.

See *Constant Population Density* for usage documentation.

- *PopulationConstantDensity*: Defines XML changes for maintain population density across arena sizes.

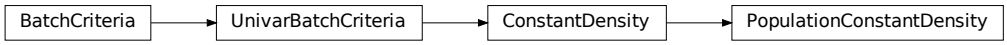
class sierra.plugins.platform.argos.variables.population_constant_density.**PopulationConstantDensity**(*arg **kw

Defines XML changes for maintain population density across arena sizes.

This class is a base class which should (almost) never be used on its own. Instead, the `factory()` function should be used to dynamically create derived classes expressing the user's desired density.

Does not change the # blocks/block manifest.

Inheritance



__doc__ = "Defines XML changes for maintain population density across arena sizes.\n\n This class is a base class which should (almost) never be used on its\n own. Instead, the ``factory()`` function should be used to dynamically\n create derived classes expressing the user's desired density.\n\n Does not change the # blocks/block manifest.\n\n "

__init__(**args*, ***kwargs*) → None

__module__ = 'sierra.plugins.platform.argos.variables.population_constant_density'

gen_attr_changelist() → List[*sierra.core.experiment.xml.AttrChangeSet*]

Generate XML modifications to to maintain constant population density.

Robots are approximated as point masses.

gen_exp_names(*cmdopts: Dict[str, Any]*) → List[str]

Generate list of experiment names from the criteria.

Used for creating unique directory names for each experiment in the batch.

Returns List of experiments names for current experiment.

graph_xlabel(*cmdopts: Dict[str, Any]*) → str

graph_xticklabels(*cmdopts: Dict[str, Any]*, *exp_names: Optional[List[str]] = None*) → List[str]

graph_xticks(*cmdopts: Dict[str, Any]*, *exp_names: Optional[List[str]] = None*) → List[float]

n_robots(*exp_num: int*) → int

sierra.plugins.platform.argos.variables.population_size

Classes for the population size batch criteria.

See *Population Size* for usage documentation.

- `PopulationSize`: A univariate range of swarm sizes used to define batch experiments.

class sierra.plugins.platform.argos.variables.population_size.**PopulationSize**(*cli_arg: str*, *main_config: Dict[str, Any]*, *batch_input_root: pathlib.Path*, *size_list: List[int]*)

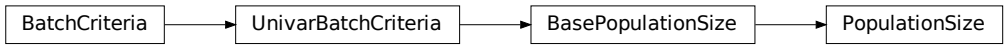
A univariate range of swarm sizes used to define batch experiments.

This class is a base class which should (almost) never be used on its own. Instead, the `factory()` function should be used to dynamically create derived classes expressing the user's desired size distribution.

Note: Usage of this class assumes homogeneous swarms.

size_list
List of integer swarm sizes defining the range of the variable for the batch experiment.

Inheritance



__doc__ = "A univariate range of swarm sizes used to define batch experiments.\n\n This class is a base class which should (almost) never be used on its\n own. Instead, the ``factory()`` function should be used to dynamically\n create derived classes expressing the user's desired size distribution.\n\n Note: Usage of this class assumes homogeneous swarms.\n\n Attributes:\n\n size_list: List of integer swarm sizes defining the range of the\n variable for the batch experiment.\n\n "

__init__(*cli_arg: str*, *main_config: Dict[str, Any]*, *batch_input_root: pathlib.Path*, *size_list: List[int]*) → None

__module__ = 'sierra.plugins.platform.argos.variables.population_size'

gen_attr_changelist() → List[*sierra.core.experiment.xml.AttrChangeSet*]
Generate list of sets of changes for swarm sizes to define a batch experiment.

static gen_attr_changelist_from_list(*sizes: List[int]*) → List[*sierra.core.experiment.xml.AttrChangeSet*]

gen_exp_names(*cmdopts: Dict[str, Any]*) → List[str]
Generate list of experiment names from the criteria.

Used for creating unique directory names for each experiment in the batch.

Returns List of experiments names for current experiment.

n_robots(*exp_num: int*) → int

sierra.plugins.platform.argos.variables.population_variable_density

Classes for the variable population density batch criteria.

See *Variable Population Density* for usage documentation.

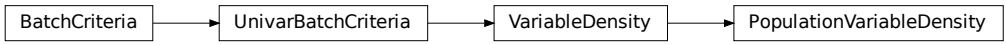
- `PopulationVariableDensity`: Defines XML changes for variable population density within a single arena.

class sierra.plugins.platform.argos.variables.population_variable_density.**PopulationVariableDensity**(**arg*
***kw*

Defines XML changes for variable population density within a single arena.

This class is a base class which should (almost) never be used on its own. Instead, the `factory()` function should be used to dynamically create derived classes expressing the user's desired density ranges.

Inheritance



__doc__ = "Defines XML changes for variable population density within a single arena.\n\n This class is a base class which should (almost) never be used on its\n own. Instead, the ``factory()`` function should be used to dynamically\n create derived classes expressing the user's desired density ranges.\n\n "

__init__(**args*, ***kwargs*) → None

__module__ = 'sierra.plugins.platform.argos.variables.population_variable_density'

attr_changes: List[*sierra.core.experiment.xml.AttrChangeSet*]

gen_attr_changelist() → List[*sierra.core.experiment.xml.AttrChangeSet*]
Generate XML modifications to achieve the desired population densities.

Robots are approximated as point masses.

gen_exp_names(*cmdopts: Dict[str, Any]*) → List[str]
Generate list of experiment names from the criteria.

Used for creating unique directory names for each experiment in the batch.

Returns List of experiments names for current experiment.

graph_xlabel(*cmdopts: Dict[str, Any]*) → str

graph_xticklabels(*cmdopts: Dict[str, Any]*, *exp_names: Optional[List[str]] = None*) → List[str]

graph_xticks(*cmdopts: Dict[str, Any]*, *exp_names: Optional[List[str]] = None*) → List[float]

n_robots(*exp_num: int*) → int

sierra.plugins.platform.argos.variables.rendering

ARGoS headless QT rendering configuration.

- *ARGoSQTHeadlessRendering*: Sets up ARGoS headless rendering with QT.

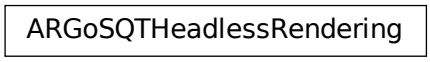
class sierra.plugins.platform.argos.variables.rendering.**ARGoSQTHeadlessRendering**(*setup:* sierra.plugins.platform.argos.

Sets up ARGoS headless rendering with QT.

tsetup
Simulation time definitions.

extents
List of (X,Y,Zs) tuple of dimensions of area to assign to engines of the specified type.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.plugins.platform.argos.variables.rendering', '__doc__': '\n Sets up ARGoS headless rendering with QT.\n\n Attributes:\n\n tsetup: Simulation time definitions.\n\n extents: List of (X,Y,Zs) tuple of dimensions of area to assign to\n engines of the specified type.\n ', 'kFrameSize': '1600x1200', 'kQUALITY': 100, 'kFRAME_RATE': 10, '__init__': <function ARGoSQTHeadlessRendering.__init__>, 'gen_attr_changelist': <function ARGoSQTHeadlessRendering.gen_attr_changelist>, 'gen_tag_rmlist': <function ARGoSQTHeadlessRendering.gen_tag_rmlist>, 'gen_tag_addlist': <function ARGoSQTHeadlessRendering.gen_tag_addlist>, 'gen_files': <function ARGoSQTHeadlessRendering.gen_files>, '__dict__': <attribute '__dict__' of 'ARGoSQTHeadlessRendering' objects>, '__weakref__': <attribute '__weakref__' of 'ARGoSQTHeadlessRendering' objects>, '__annotations__': {'tag_adds': 'tp.List[xml.TagAddList]'}})

__doc__ = '\n Sets up ARGoS headless rendering with QT.\n\n Attributes:\n\n tsetup: Simulation time definitions.\n\n extents: List of (X,Y,Zs) tuple of dimensions of area to assign to\n engines of the specified type.\n '

__init__(*setup:* sierra.plugins.platform.argos.variables.exp_setup.ExpSetup) → None

__module__ = 'sierra.plugins.platform.argos.variables.rendering'

__weakref__
list of weak references to the object (if defined)

gen_attr_changelist() → List[*sierra.core.experiment.xml.AttrChangeSet*]
No effect.

All tags/attributes are either deleted or added.

gen_files() → None

gen_tag_addlist() → List[*sierra.core.experiment.xml.TagAddList*]

gen_tag_rmlist() → List[*sierra.core.experiment.xml.TagRmList*]
Remove the <qt_opengl> tag if it exists.

Obviously you *must* call this function BEFORE adding new definitions.

kFRAME_RATE = 10

kFrameSize = '1600x1200'

kQUALITY = 100

sierra.plugins.platform.ros1gazebo

sierra.plugins.platform.ros1gazebo.cmdline

Command line parsing and validation for the *ROS1+Gazebo* platform.

sierra.plugins.platform.ros1gazebo.generators

sierra.plugins.platform.ros1gazebo.generators.platform_generators

Classes for generating common XML modifications for *ROS1+Gazebo*.

I.e., changes which are platform-specific, but applicable to all projects using the platform.

- *PlatformExpDefGenerator*: Init the object.
- *PlatformExpRunDefUniqueGenerator*: Generate XML changes unique to a experimental runs for ROS experiments.

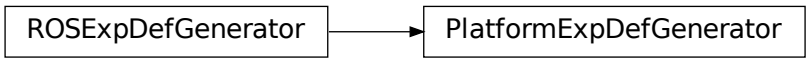
class sierra.plugins.platform.ros1gazebo.generators.platform_generators.**PlatformExpDefGenerator**(*exp_spec*: sierra.core.experiment.spec.ExperimentSpec, *controller: str*, *cmdopts: Dict[str, Any]*, ***kwargs*)

Init the object.

controller
The controller used for the experiment.

cmdopts
Dictionary of parsed cmdline parameters.

Inheritance



__doc__ = '\n Init the object.\n\n Attributes:\n\n controller: The controller used for the experiment.\n\n cmdopts: Dictionary of parsed cmdline parameters.\n '

__init__(*exp_spec: sierra.core.experiment.spec.ExperimentSpec*, *controller: str*, *cmdopts: Dict[str, Any]*, ***kwargs*) → None

__module__ = 'sierra.plugins.platform.ros1gazebo.generators.platform_generators'

_generate_gazebo_core(*exp_def:* sierra.core.experiment.definition.XMLExpDef) → None
Generate XML tag changes to setup Gazebo core experiment parameters.

Does not write generated changes to the run definition pickle file.

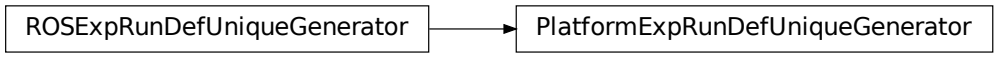
_generate_gazebo_vis(*exp_def:* sierra.core.experiment.definition.XMLExpDef) → None
Generate XML changes to configure Gazebo visualizations.

Does not write generated changes to the simulation definition pickle file.

generate() → *sierra.core.experiment.definition.XMLExpDef*

class sierra.plugins.platform.ros1gazebo.generators.platform_generators.**PlatformExpRunDefUniqueGenerato**

Inheritance



__doc__ = None

__init__(**args*, ***kwargs*) → None

__module__ = 'sierra.plugins.platform.ros1gazebo.generators.platform_generators'

generate(*exp_def:* sierra.core.experiment.definition.XMLExpDef)

sierra.plugins.platform.ros1gazebo.plugin

Provides platform-specific callbacks for the *ROS1+Gazebo* platform.

sierra.plugins.platform.ros1gazebo.variables

sierra.plugins.platform.ros1gazebo.variables.population_size

Classes for the population size batch criteria.

See *System Population Size* for usage documentation.

- *PopulationSize*: A univariate range of system sizes used to define batch experiments.

class sierra.plugins.platform.ros1gazebo.variables.population_size.**PopulationSize**(*cli_arg: str*, *main_config: Dict[str, Any]*, *batch_input_root: pathlib.Path*, *robot: str*, *sizes: List[int]*, *positions: List[*sierra.core.vector.Vecto

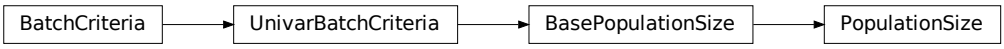
A univariate range of system sizes used to define batch experiments.

This class is a base class which should (almost) never be used on its own. Instead, the `factory()` function should be used to dynamically create derived classes expressing the user's desired size distribution.

Note: Usage of this class assumes homogeneous systems.

size_list
List of integer system sizes defining the range of the variable for the batch experiment.

Inheritance



__doc__ = "A univariate range of system sizes used to define batch experiments.\n\n This class is a base class which should (almost) never be used on its\n own. Instead, the ``factory()`` function should be used to dynamically\n create derived classes expressing the user's desired size distribution.\n\n Note: Usage of this class assumes homogeneous systems.\n\n Attributes:\n\n size_list: List of integer system sizes defining the range of the\n variable for the batch experiment.\n\n "

__init__(*cli_arg: str*, *main_config: Dict[str, Any]*, *batch_input_root: pathlib.Path*, *robot: str*, *sizes: List[int]*, *positions: List[*sierra.core.vector.Vector3D*]*) → None

__module__ = 'sierra.plugins.platform.ros1gazebo.variables.population_size'

gen_exp_names(*cmdopts: Dict[str, Any]*) → List[str]
Generate list of experiment names from the criteria.

Used for creating unique directory names for each experiment in the batch.

Returns List of experiments names for current experiment.

gen_tag_addlist() → List[*sierra.core.experiment.xml.TagAddList*]
Generate XML modifications to set system sizes.

n_robots(*exp_num: int*) → int

sierra.plugins.platform.ros1robot

sierra.plugins.platform.ros1robot.cmdline

Command line parsing and validation for the *ROS1+robot* platform.

sierra.plugins.platform.ros1robot.generators

sierra.plugins.platform.ros1robot.generators.platform_generators

Classes for generating common XML modifications to *ROS1* input files.

I.e., changes which are platform-specific, but applicable to all projects using ROS with a real robot execution environment.

- *PlatformExpDefGenerator*: Init the object.
- *PlatformExpRunDefUniqueGenerator*: Generate XML changes unique to a experimental runs for ROS experiments.

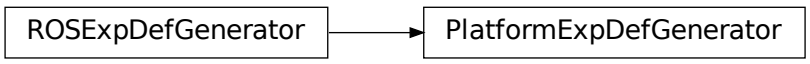
class sierra.plugins.platform.ros1robot.generators.platform_generators.**PlatformExpDefGenerator**(*exp_spec:* sierra.core.experiment.spec.ExperimentSpec, *controller: str*, *cmdopts: Dict[str, Any]*, ***kwargs*)

Init the object.

controller
The controller used for the experiment.

cmdopts
Dictionary of parsed cmdline parameters.

Inheritance



__doc__ = '\n Init the object.\n\n Attributes:\n\n controller: The controller used for the experiment.\n cmdopts: Dictionary of parsed cmdline parameters.\n '

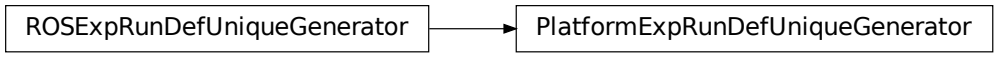
__init__(*exp_spec:* sierra.core.experiment.spec.ExperimentSpec, *controller: str*, *cmdopts: Dict[str, Any]*, ***kwargs*) → None

__module__ = 'sierra.plugins.platform.ros1robot.generators.platform_generators'

generate() → *sierra.core.experiment.definition.XMLExpDef*

class sierra.plugins.platform.ros1robot.generators.platform_generators.**PlatformExpRunDefUniqueGenerator**

Inheritance



__doc__ = None

__init__(**args*, ***kwargs*) → None

__module__ = 'sierra.plugins.platform.ros1robot.generators.platform_generators'

generate(*exp_def:* sierra.core.experiment.definition.XMLExpDef)

sierra.plugins.platform.ros1robot.plugin

sierra.plugins.platform.ros1robot.variables

sierra.plugins.platform.ros1robot.variables.population_size

Classes for the population size batch criteria.

See *System Population Size* for usage documentation.

- `PopulationSize`: A univariate range of system sizes used to define batch experiments.

class sierra.plugins.platform.ros1robot.variables.population_size.**PopulationSize**(*cli_arg: str*, *main_config: Dict[str, Any]*, *batch_input_root: path-lib.Path*, *robot: str*, *sizes: List[int]*)

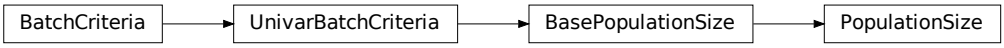
A univariate range of system sizes used to define batch experiments.

This class is a base class which should (almost) never be used on its own. Instead, the `factory()` function should be used to dynamically create derived classes expressing the user's desired size distribution.

Note: Usage of this class assumes homogeneous systems.

size_list
List of integer system sizes defining the range of the variable for the batch experiment.

Inheritance



__doc__ = "A univariate range of system sizes used to define batch experiments.\n\n This class is a base class which should (almost) never be used on its\n own. Instead, the ``factory()`` function should be used to dynamically\n create derived classes expressing the user's desired size distribution.\n\n Note: Usage of this class assumes homogeneous systems.\n\n Attributes:\n\n size_list: List of integer system sizes defining the range of the\n variable for the batch experiment.\n\n "

__init__(*cli_arg: str*, *main_config: Dict[str, Any]*, *batch_input_root: pathlib.Path*, *robot: str*, *sizes: List[int]*) → None

__module__ = 'sierra.plugins.platform.ros1robot.variables.population_size'

gen_exp_names(*cmdopts: Dict[str, Any]*) → List[str]
Generate list of experiment names from the criteria.

Used for creating unique directory names for each experiment in the batch.

Returns List of experiments names for current experiment.

gen_tag_addlist() → List[*sierra.core.experiment.xml.TagAddList*]
Generate XML modifications to set system sizes.

n_robots(*exp_num: int*) → int

sierra.plugins.robot

sierra.plugins.robot.turtlebot3

sierra.plugins.robot.turtlebot3.plugin

Robot plugin for running SIERRA with a set of Turtlebot3 robots.

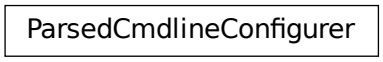
- *ParsedCmdlineConfigurer*: Configure SIERRA for the turtlebot3 execution environment.
- *ExpShellCmdsGenerator*: Generate the cmds to invoke GNU Parallel to launch ROS on the turtlebots.
- *ExecEnvChecker*: Base class for verifying execution environments before running experiments.

class sierra.plugins.robot.turtlebot3.plugin.**ParsedCmdlineConfigurer**(*exec_env: str*)
Configure SIERRA for the turtlebot3 execution environment.

May use the following environment variables:

- `SIERRA_NODEFILE` - If this is not defined `--nodefile` must be passed.

Inheritance



__call__(*args: argparse.Namespace*) → None
Call self as a function.

__dict__ = mappingproxy({'__module__': 'sierra.plugins.robot.turtlebot3.plugin', '__doc__': 'Configure SIERRA for the turtlebot3 execution environment.\n\n May use the following environment variables:\n\n - ``SIERRA_NODEFILE`` - If this is not defined ``--nodefile`` must be\n passed.\n\n ', '__init__': <function ParsedCmdlineConfigurer.__init__>, '__call__': <function ParsedCmdlineConfigurer.__call__>, '__dict__': <attribute '__dict__' of 'ParsedCmdlineConfigurer' objects>, '__weakref__': <attribute '__weakref__' of 'ParsedCmdlineConfigurer' objects>, '__annotations__': {}})

__doc__ = 'Configure SIERRA for the turtlebot3 execution environment.\n\n May use the following environment variables:\n\n - ``SIERRA_NODEFILE`` - If this is not defined ``--nodefile`` must be\n passed.\n\n '

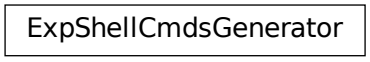
__init__(*exec_env: str*) → None

__module__ = 'sierra.plugins.robot.turtlebot3.plugin'

__weakref__
list of weak references to the object (if defined)

class sierra.plugins.robot.turtlebot3.plugin.**ExpShellCmdsGenerator**(*cmdopts: Dict[str, Any]*, *exp_num: int*)
Generate the cmds to invoke GNU Parallel to launch ROS on the turtlebots.

Inheritance



__dict__ = mappingproxy({'__module__': 'sierra.plugins.robot.turtlebot3.plugin', '__doc__': 'Generate the cmds to invoke GNU Parallel to launch ROS on the turtlebots.\n\n ', '__init__': <function ExpShellCmdsGenerator.__init__>, 'pre_exp_cmds': <function ExpShellCmdsGenerator.pre_exp_cmds>, 'post_exp_cmds': <function ExpShellCmdsGenerator.post_exp_cmds>, 'exec_exp_cmds': <function ExpShellCmdsGenerator.exec_exp_cmds>, '__dict__': <attribute '__dict__' of 'ExpShellCmdsGenerator' objects>, '__weakref__': <attribute '__weakref__' of 'ExpShellCmdsGenerator' objects>, '__annotations__': {}})

__doc__ = 'Generate the cmds to invoke GNU Parallel to launch ROS on the turtlebots.\n\n '

__init__(*cmdopts: Dict[str, Any]*, *exp_num: int*) → None

__module__ = 'sierra.plugins.robot.turtlebot3.plugin'

__weakref__
list of weak references to the object (if defined)

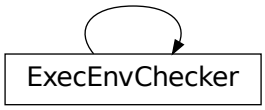
exec_exp_cmds(*exec_opts: Dict[str, str]*) → List[*sierra.core.types.ShellCmdSpec*]

post_exp_cmds() → List[*sierra.core.types.ShellCmdSpec*]

pre_exp_cmds() → List[*sierra.core.types.ShellCmdSpec*]

class sierra.plugins.robot.turtlebot3.plugin.**ExecEnvChecker**(*cmdopts: Dict[str, Any]*)

Inheritance



__call__() → None
Call self as a function.

__doc__ = None

__init__(*cmdopts: Dict[str, Any]*) → None

__module__ = 'sierra.plugins.robot.turtlebot3.plugin'

sierra.plugins.storage

sierra.plugins.storage.csv

sierra.plugins.storage.csv.plugin

Plugin for reading from CSV files when processing experimental run results.

CHAPTER

SEVENTEEN

CITING SIERRA

If you use SIERRA and find it helpful, please cite the following paper:

```
@inproceedings{Harwell2022a-SIERRA,
 author = {Harwell, John and Lowmanstone, London and Gini, Maria},
 title = {SIERRA: A Modular Framework for Research Automation},
 year = {2022},
 isbn = {9781450392136},
 publisher = {International Foundation for Autonomous Agents and Multiagent Systems},
 address = {Richland, SC},
 booktitle = {Proceedings of the 21st International Conference on Autonomous Agents and␣
↪Multiagent Systems},
 pages = {1905-1907},
 numpages = {3},
 keywords = {simulation, real robots, research automation, scientific method},
 location = {Virtual Event, New Zealand},
 series = {AAMAS '22}
 }
```

You can also cite the following DOI for the specific version of SIERRA used, to help facilitate reproducibility:

CHAPTER

EIGHTEEN

SIERRA IN THE WILD

Here is a non-exhaustive list of some of the different ways SIERRA has been used.

18.1 Papers

- Improved Swarm Engineering: Aligning Intuition and Analysis
- Characterizing The Limits of Linear Modeling of Non-Linear Swarm Behaviors
- Demystifying Emergent Intelligence and Its Effect on Performance in Large Swarms
- Swarm Engineering Through Quantitative Measurement in a 10,000 Robot Swarm
- Socially Inspired Communication in Swarm Robotics
- Maximizing Energy Efficiency in Swarm Robotics

18.2 Projects

- FORDYCA
- PRISM

18.3 Demos

- 2022 AAMAS Demo

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

F

G

H

I

J

K

L

M

N

S

T

U

V

W

X

Y

Z