libEnsemble User Manual

Release 0.5.0

Stephen Hudson, Jeffrey Larson, Stefan M. Wild, David Bindel

May 24, 2019
# What is libEnsemble?

1.1 Dependencies

1.2 Installation

1.3 Testing

1.4 Basic Usage

1.5 Documentation

1.6 Citing libEnsemble

1.7 Support

# Contributing

# Release Notes

3.1 Release 0.5.0

3.2 Release 0.4.1

3.3 Release 0.4.0

3.4 Release 0.3.0

3.5 Release 0.2.0

3.6 Release 0.1.0

# Introduction

4.1 libEnsemble overview

4.2 Expected use cases

4.3 The libEnsemble History Array

4.4 LibEnsemble Output

# libE Module

5.1 Main libEnsemble routine

# Data Structures

6.1 history array

6.2 worker array

6.3 work dictionary

6.4 libE_specs

6.5 sim_specs

6.6 gen_specs

6.7 exit_criteria

6.8 alloc_specs

6.9 persis_info

# User Functions

7.1 Sim, Gen, and Alloc functions API
A library for managing ensemble-like collections of computations.
WHAT IS LIBENSEMBLE?

libEnsemble is a Python library to coordinate the concurrent evaluation of ensembles of computations. Designed with flexibility in mind, libEnsemble can utilize massively parallel resources to accelerate the solution of design, decision, and inference problems.

A visual overview is given in the libEnsemble poster.

libEnsemble aims for:

• Extreme scaling
• Resilience/Fault tolerance
• Monitoring/killing jobs (and recovering resources)
• Portability and flexibility
• Exploitation of persistent data/control flow.

The user selects or supplies a generation function that produces simulation input as well as a simulation function that performs and monitors the simulations. The generation function may contain, for example, an optimization method to generate new simulation parameters on-the-fly and based on the results of previous simulations. Examples and templates of these functions are included in the library.

libEnsemble employs a manager-worker scheme that can run on various communication media (including MPI, multiprocessing, and TCP). Each worker can control and monitor any type of job from small sub-node jobs to huge many-node simulations. A job controller interface is provided to ensure scripts are portable, resilient and flexible; it also enables automatic detection of the nodes and cores in a system and can split up jobs automatically if nodes/cores are not supplied.

1.1 Dependencies

Required dependencies:

• Python 3.4 or above.
• NumPy

For libEnsemble running with the mpi4py parallelism:

• A functional MPI 1.x/2.x/3.x implementation such as MPICH built with shared/dynamic libraries.
• mpi4py v2.0.0 or above

Optional dependency:

• Balsam
From v0.2.0, libEnsemble has the option of using the Balsam job manager. This is required for running libEnsemble on the compute nodes of some supercomputing platforms (e.g., Cray XC40); platforms that do not support launching jobs from compute nodes. Note that as of v0.5.0, libEnsemble can also be run on the launch nodes using multiprocessing.

The example sim and gen functions and tests require the following dependencies:

- SciPy
- petsc4py
- PETSc - This can optionally be installed by pip along with petsc4py
- NLopt - Installed with shared libraries enabled.

PETSc and NLopt must be built with shared libraries enabled and present in sys.path (e.g., via setting the PYTHONPATH environment variable). NLopt should produce a file nlopt.py if Python is found on the system.

### 1.2 Installation

Use pip to install libEnsemble and its dependencies:

```bash
pip install libensemble
```

libEnsemble is also available in the Spack distribution.

The tests and examples can be accessed in the github repository. A tarball of the most recent release is also available.

### 1.3 Testing

The provided test suite includes both unit and regression tests and is run regularly on:

- Travis CI

The test suite requires the mock, pytest, pytest-cov and pytest-timeout packages to be installed and can be run from the libensemble/tests directory of the source distribution by running:

```
./run-tests.sh
```

To clean the test repositories run:

```
./run-tests.sh -c
```

Further options are available. To see a complete list of options run:

```
./run-tests.sh -h
```

Coverage reports are produced separately for unit tests and regression tests under the relevant directories. For parallel tests, the union of all processors is taken. Furthermore, a combined coverage report is created at the top level, which can be viewed after running the tests via the html file libensemble/tests/cov_merge/index.html. The Travis CI coverage results are given online at Coveralls.

Note: The job_controller tests can be run using the direct-launch or Balsam job controllers. However, currently only the direct-launch versions can be run on Travis CI, which reduces the test coverage results.
1.4 Basic Usage

The examples directory contains example libEnsemble calling scripts, sim functions, gen functions, alloc functions and job submission scripts.

The user will create a python script to call the libEnsemble libE function. This must supply the sim_specs and gen_specs, and optionally libE_specs, alloc_specs and persis_info.

The default manager/worker communications mode is MPI. The user script is launched as:

```
mpiexec -np N python myscript.py
```

where N is the number of processors. This will launch one manager and N-1 workers.

If running in local mode, which uses Python’s multiprocessing module, the ‘local’ comms option and the number of workers must be specified in libE_specs. The script can then be run as a regular python script:

```
python myscript.py
```

When specifying these options via command line options, one may use the parse_args function used in the regression tests, which can be found in libensemble/tests/regression_tests/common.py

See the user-guide for more information.

1.5 Documentation

- http://libensemble.readthedocs.org/

1.6 Citing libEnsemble

Please use the following to cite libEnsemble in a publication:

```
@techreport{libEnsemble,  
  author = {Stephen Hudson and Jeffrey Larson and Stefan M. Wild and David Bindel},  
  title = {{libEnsemble} Users Manual},  
  institution = {Argonne National Laboratory},  
  number = {Revision 0.5.0},  
  year = {2019},  
  url = {https://buildmedia.readthedocs.org/media/pdf/libensemble/latest/libensemble.pdf}
}
```

1.7 Support

Join the libEnsemble mailing list at:

- https://lists.mcs.anl.gov/mailman/listinfo/libensemble

or email questions to:

- libensemble@lists.mcs.anl.gov
or communicate (and establish a private channel, if desired) at:

- https://libensemble.slack.com
Contributions may be made via Github pull request to:

https://github.com/Libensemble/libensemble

libEnsemble uses the Gitflow model. Contributors should branch from, and make pull requests to, the develop branch. The master branch is used only for releases. Code should pass flake8 tests, allowing for the exceptions given in the “.flake8” configuration file in the project directory.

Issues can be raised at:

https://github.com/Libensemble/libensemble/issues

Issues may include reporting bugs or suggested features. Administrators will add issues, as appropriate, to the project board at:

https://github.com/Libensemble/libensemble/projects

By convention, user branch names should have a <type>/<name> format, where example types are feature, bugfix, testing, docs and experimental. Administrators may take a hotfix branch from the the master, which will be merged into master (as a patch) and develop. Administrators may also take a release branch off develop and merge into master and develop for a release. Most branches should relate to an issue.

When a branch closes a related issue, the pull request message should include the phrase “Closes #N” where N is the issue number. This will automatically close out the issues when they are pulled into the default branch (currently master).

libEnsemble is distributed under a 3-clause BSD license (see LICENSE). The act of submitting a pull request (with or without an explicit Signed-off-by tag) will be understood as an affirmation of the following:

Developer’s Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or

(b) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or

(c) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.

(d) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.
3.1 Release 0.5.0

Date May 22, 2019

• Added local (multiprocessing) and TCP options for manager/worker communications, in addition to mpi4py (#42).

• E.g., libEnsemble can be run on MOM/launch nodes (e.g., those of ALCF/Theta & OLCF/Summit) and can remotely detect compute resources.

• E.g., libEnsemble can be run on a system without MPI.

• E.g., libEnsemble can be run with a local manager and remote TCP workers.

• Added support for Summit/LSF scheduler in job controller.

• MPI job controller detects and re-tries launches on failure; adding resilience (#143).

• Job controller supports option to extract/print job times in libE_stats.txt (#136).

• Default logging level changed to INFO (#164).

• Logging interface added, which allows user to change logging level and file (#110).

• All worker logging and calculation stats are routed through manager.

• libEnsemble can be run without a gen_func, for example, when using a previously computed random sample (#122).

• Aborts dump persis_info with the history.

Note

• This version no longer supports Python 2.

• Tested platforms include: Local Linux, Theta (Cray XC40/Cobalt), Summit (IBM Power9/LSF), Bebop (Cray CS400/Slurm).

Known issues

• OpenMPI does not work with direct MPI job launches in mpi4py comms mode, as it does not support nested MPI launches (Either use local mode or Balsam job controller).

• Local comms mode (multiprocessing) may fail if MPI is initialized before forking processors. This is thought to be responsible for issues combining with PETSc.

• Remote detection of logical cores via LSB_HOSTS (e.g., Summit) returns number of physical cores as SMT info not available.
TCP mode does not support: 1) more than one libEnsemble call in a given script or 2) the auto-resources option to the job controller.

### 3.2 Release 0.4.1

**Date** February 20, 2019

- Logging no longer uses root logger (Also added option to change libEnsemble log level) (#105)
- Added wait_on_run option for job controller launch to block until jobs have started (#111)
- persis_info can be passed to sim as well as gen functions (#112)
- Post-processing scripts added to create performance/utilization graphs (#102)
- New scaling test added (not part of current CI test suite) (#114)

### 3.3 Release 0.4.0

**Date** November 7, 2018

- Separate job controller classes into different modules including a base class (API change)
- Add central_mode run option to distributed type (MPI) job_controllers (API addition) (#93)
- Make poll and kill job methods (API change)
- In job_controller, set_kill_mode is removed and replaced by a wait argument for a hard kill (API change)
- Removed register module - incorporated into job_controller (API change)
- APOSMM has improved asynchronicity when batch mode is false (with new example). (#96)
- Manager errors (instead of hangs) when alloc_f or gen_f don’t return work when all workers are idle. (#95)

**Known issues**

- OpenMPI is not supported with direct MPI launches as nested MPI launches are not supported.

### 3.4 Release 0.3.0

**Date** September 7, 2018

- Issues with killing jobs have been fixed (#21)
- Fix to job_controller manager_poll to work with multiple jobs (#62)
- API change: persis_info now included as an argument to libE and is returned from libE instead of gen_info
- Gen funes: aposmm_logic module renamed to aposmm.
- New example gen and allocation functions.
- Updated Balsam launch script (with new Balsam workflow)
- History is dumped to file on manager or worker exception and MPI aborted (with exit code 1) (#46)
- Default logging level changed to DEBUG and redirected to file ensemble.log
- Added directory of standalone tests (comms, job kills, and nested MPI launches)
• Improved and speeded up unit tests (#68)
• Considerable documentation enhancements

Known issues
• OpenMPI is not supported with direct MPI launches as nested MPI launches are not supported.

3.5 Release 0.2.0

Date June 29, 2018
• Added job_controller interface (for portable user scripts).
• Added support for using the Balsam job manager. Enables portability and dynamic scheduling.
• Added auto-detection of system resources.
• Scalability testing: Ensemble performed with 1023 workers on Theta (Cray XC40) using Balsam.
• Tested MPI libraries: MPICH, Intel MPI.

Known issues
• Killing MPI jobs does not work correctly on some systems (including Cray XC40 and CS400). In these cases, libEnsemble continues, but processes remain running.
• OpenMPI does not work correctly with direct launches (and has not been tested with Balsam).

3.6 Release 0.1.0

Date November 30, 2017
• Initial Release.
4.1 libEnsemble overview

libEnsemble is a software library to coordinate the concurrent evaluation of ensembles of calculations. libEnsemble uses a manager to allocate work to various workers. (A libEnsemble worker is the smallest indivisible unit to perform some calculation.) The work performed by libEnsemble is governed by three routines:

- **gen_f**: Generates inputs to sim_f.
- **sim_f**: Evaluates a simulation or other evaluation at output from gen_f.
- **alloc_f**: Decides whether sim_f or gen_f should be called (and with what input/resources) as workers become available.

Example sim_f, gen_f, and alloc_f routines can be found in the examples/sim_funcs/, examples/gen_funcs/, and examples/alloc_funcs/ directories, respectively. Examples of scripts used for calling libEnsemble can be found in examples/calling_scripts/. To enable portability, a job_controller interface is supplied for users to launch and monitor jobs in their user-provided sim_f and gen_f routines.

The default alloc_f tells each available worker to call sim_f with the highest priority unit of work from gen_f. If a worker is idle and there is no gen_f output to give, the worker is told to call gen_f.

4.2 Expected use cases

Below are some expected libEnsemble use cases that we support (or are working to support) and plan to have examples of:

- A user is looking to optimize a simulation calculation. The simulation may already be using parallel resources, but not a large fraction of some computer. libEnsemble can coordinate the concurrent evaluation of the simulation sim_f at various parameter values and gen_f would return candidate parameter values (possibly after each sim_f output).

- A user has a gen_f that produces different meshes to be used within a sim_f. Given the sim_f output, gen_f will refine a mesh or produce a new mesh. libEnsemble can ensure that the calculated meshes can be used by multiple simulations without requiring movement of data.

- A user is attempting to sample a simulation sim_f at some parameter values, many of which will cause the simulation to fail. libEnsemble can stop unresponsive evaluations, and recover computational resources for future evaluations. gen_f can possibly update the sampling after discovering regions where evaluations of sim_f fail.

- A user has a simulation sim_f that requires calculating multiple expensive quantities, some of which depend on other quantities. sim_f can observe intermediate quantities in order to stop related calculations and preempt future calculations associated with poor parameter values.
• A user has a sim_f with multiple fidelities, with the higher-fidelity evaluations requiring more computational resources, and a gen_f/alloc_f that decides which parameters should be evaluated and at what fidelity level. libEnsemble can coordinate these evaluations without requiring the user know parallel programming.

• A user wishes to identify multiple local optima for a sim_f. Furthermore, sensitivity analysis is desired at each identified optimum. libEnsemble can use the points from the APOSMM gen_f to identify optima; and after a point is ruled to be an optimum, a different gen_f can produce a collection of parameters necessary for sensitivity analysis of sim_f.

Naturally, combinations of these use cases are supported as well. An example of such a combination is using libEnsemble to solve an optimization problem that relies on simulations that fail frequently.

### 4.3 The libEnsemble History Array

libEnsemble uses a numpy structured array H to store output from gen_f and corresponding sim_f output. Similarly, gen_f and sim_f are expected to return output in numpy structured arrays. The names of the fields to be given as input to gen_f and sim_f must be an output from gen_f or sim_f. In addition to the fields output from sim_f and gen_f, the final history returned from libEnsemble will include the following fields:

- **sim_id** [int]: Each unit of work output from gen_f must have an associated sim_id. The generator can assign this, but users must be careful to ensure points are added in order. For example, if alloc_f allows for two gen_f instances to be running simultaneously, alloc_f should ensure that both don’t generate points with the same sim_id.

- **given** [bool]: Has this gen_f output been given to a libEnsemble worker to be evaluated yet?

- **given_time** [float]: At what time (since the epoch) was this gen_f output given to a worker?

- **sim_worker** [int]: libEnsemble worker that it was given to be evaluated.

- **gen_worker** [int]: libEnsemble worker that generated this sim_id

- **gen_time** [float]: At what time (since the epoch) was this entry (or collection of entries) put into H by the manager

- **returned** [bool]: Has this worker completed the evaluation of this unit of work?

### 4.4 LibEnsemble Output

The history array is returned to the user by libEnsemble. In the case that libEnsemble aborts on an exception, the existing history array is dumped to a file libE_history_at_abort_<sim_count>.npy, where sim_count is the number of points evaluated.

Other libEnsemble files produced by default are:

- **libE_stats.txt**: This contains a one-line summary of all user calculations. Each calculation summary is sent by workers to the manager and printed as the run progresses.

- **ensemble.log**: This is the logging output from libEnsemble. The default logging is at INFO level. To gain additional diagnostics logging level can be set to DEBUG. If this file is not removed, multiple runs will append output. For more info, see *Logging*.  


5.1 Main libEnsemble routine

This is the outer libEnsemble routine.

We dispatch to different types of worker teams depending on the contents of libE_specs. If ‘comm’ is a field, we use MPI; if ‘nthreads’ is a field, we use threads; if ‘nprocesses’ is a field, we use multiprocessing.

If an exception is encountered by the manager or workers, the history array is dumped to file and MPI abort is called.

Parameters

- **sim_specs** (dict) – Specifications for the simulation function *(example)*
- **gen_specs** (dict) – Specifications for the generator function *(example)*
- **exit_criteria** (dict) – Tell libEnsemble when to stop a run *(example)*
- **persis_info** (dict, optional) – Persistent information to be passed between user functions *(example)*
- **alloc_specs** (dict, optional) – Specifications for the allocation function *(example)*
- **libE_specs** (dict, optional) – Specifications for libEnsemble *(example)*
- **H0** (dict, optional) – A previous libEnsemble history to be prepended to the history in the current libEnsemble run *(example)*

Returns

- **H** (dict) – History array storing rows for each point. *(example) Dictionary containing persistent info
- **persis_info** (dict) – Final state of persistent information *(example)*
- **exit_flag** (int) – Flag containing job status: 0 = No errors, 1 = Exception occurred 2 = Manager timed out and ended simulation
libEnsemble has the following data structures.

### 6.1 history array

Stores the history of the output from gen_f, sim_f, and alloc_f:

\[
H: \text{numpy structured array} \\
\quad \text{History array storing rows for each point.}
\]

Fields in \( H \) include those specified in sim_specs['out'], gen_specs['out'], and alloc_specs['out']. All values are initiated to 0 for integers, 0.0 for floats, and False for booleans.

Below are the protected fields used in \( H \):

```python
libE_fields = [('sim_id', int),     # The number of the entry in H that was
               'given', bool),     # True if the entry has been given to a worker
               ('given_time', float), # Time (since epoch) that the entry was given
               ('sim_worker', int),  # Worker that did (or is doing) the sim eval
               ('gen_worker', int),  # Worker that generated the entry
               ('gen_time', float),  # Time (since epoch) that entry was entered
               ('returned', bool),   # True if the entry has been returned from the
               ('sim_eval', )]
```

**Examples**

See example sim_specs, gen_specs, and alloc_specs.

### 6.2 worker array

Stores information to inform the allocation function about the current state of the workers. Workers can be in a variety of states. We take the following convention:
<table>
<thead>
<tr>
<th>Worker state</th>
<th>active</th>
<th>persis_state</th>
<th>blocked</th>
</tr>
</thead>
<tbody>
<tr>
<td>idle worker</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>active, nonpersistent sim</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>active, nonpersistent gen</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>active, persistent sim</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>active, persistent gen</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>waiting, persistent sim</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>waiting, persistent gen</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>worker blocked by some other calculation</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Note

- libE only receives from workers with ‘active’ nonzero
- libE only calls the alloc_f if some worker has ‘active’ zero

6.3 work dictionary

Dictionary with integer keys $i$ and dictionary values to be given to worker $i$. $Work[i]$ has the following form:

```
Work[i]: [dict]:

   Required keys:
     'persis_info' [dict]: Any persistent info to be sent to worker 'i'
     'H_fields' [list]: The field names of the history 'H' to be sent to worker 'i'
     'tag' [int]: 'EVAL_SIM_TAG' (resp. 'EVAL_GEN_TAG') if worker 'i' is to call sim_func (resp. gen_func)
     'libE_info' [dict]: This information is sent to and returned from the worker to help libEnsemble quickly update the 'H' and 'W'. Available keys are:
       'H_rows' [list of ints]: History rows to send to worker 'i'
       blocking [list of ints]: Workers to be blocked by the calculation given to worker 'i'
       persistent' [bool]: True if worker 'i' will enter persistent mode
```

Examples

For allocation functions using persistent workers, see
libensemble/tests/regression_tests/
test_6-hump_camel_persistent_uniform_sampling.py
or
libensemble/tests/regression_tests/
test_6-hump_camel_uniform_sampling_with_persistent_localopt_gens.py

For allocation functions giving work that blocks other workers, see
libensemble/tests/regression_tests/
test_6-hump_camel_with_different_nodes_uniform_sample.py
6.4 libE_specs

Specifications for libEnsemble:

```python
libE_specs: [dict]:
    'comms' [string]:
        Manager/Worker communications mode. Default: mpi
        Options are 'mpi', 'local', 'tcp'
    'nprocesses' [int]:
        Number of worker processes to spawn (in local/tcp modes)
    'comm' [MPI communicator]:
        libEnsemble communicator. Default: MPI.COMM_WORLD
    'color' [int]:
        Communicator color. Default: 0
```

6.5 sim_specs

Simulation function specifications to be set in user calling script and passed to libE.libE():

```python
sim_specs: [dict]:

    Required keys:
    
    'sim_f' [func]:
        the simulation function being evaluated
    'in' [list]:
        field names (as strings) that will be given to sim_f
    'out' [list of tuples (field name, data type, [size])]:
        sim_f outputs that will be stored in the libEnsemble history

    Optional keys:
    
    'save_every_k' [int]:
        Save history array to file after every k simulated points.
    'sim_dir' [str]:
        Name of simulation directory which will be copied for each worker
    'sim_dir_prefix' [str]:
        A prefix path specifying where to create sim directories

    Additional entries in sim_specs will be given to sim_f
```

Notes

- The user may define other fields to be passed to the simulator function.
- The tuples defined in the 'out' list are entered into the master history array

Examples

From: libensemble/tests/regression_tests/test_6-hump_camel_uniform_sampling.py

```python
sim_specs = {'sim_f': six_hump_camel, # This is the function whose output is being minimized
             'in': ['x'], # These keys will be given to the above function 'out': [('f',float)], # This is the output from the function being minimized 'save_every_k': 400 }
```
Note that the dimensions and type of the 'in' field variable 'x' is specified by the corresponding generator 'out' field 'x' (see gen_specs example). Only the variable name is then required in sim_specs.

## 6.6 gen_specs

Generation function specifications to be set in user calling script and passed to libE.libE():

```python
gen_specs: [dict]:

    Required keys :

        'gen_f' [func] :
            generates inputs to sim_f
        'in' [list] :
            field names (as strings) that will be given to gen_f
        'out' [list of tuples (field name, data type, [size])]:
            gen_f outputs that will be stored in the libEnsemble history

    Optional keys :

        'save_every_k' [int] :
            Save history array to file after every k generated points.
```

### Notes

- The user may define other fields to be passed to the generator function.
- The tuples defined in the 'out' list are entered into the master history array
- The generator ‘out’ field will generally include a variable(s) which is used for the simulator ‘in’ field, in which case only the variable name is required for the simulator ‘in’ field. E.g. The test_6-hump_camel_uniform_sampling.py example below, matches the corresponding sim_specs example, where ‘x’ is defined in the gen_specs ‘out’ field to give two positional floats.

### Examples

From: libensemble/tests/regression_tests/test_6-hump_camel_uniform_sampling.py:

```python
gen_specs = {'gen_f': uniform_random_sample,  
        'in': ['sim_id'],  
        'out': [('x',float,2)],  
        'lb': np.array([-3,-2]),  
        'ub': np.array([ 3, 2]),  
        'gen_batch_size': 500,  
        'save_every_k': 300  
}
```

In this example, the generation function uniform_random_sample will generate 500 random points uniformly over the 2D domain defined by gen_specs['ub'] and gen_specs['lb']. The libEnsemble manager is set to dump the history array to file after every 300 generated points, though in this case it will only happen after 500 points due to the batch size.

## 6.7 exit_criteria

Exit criteria for libEnsemble:
exit_criteria: [dict]:

Optional keys (At least one must be given):

- 'sim_max' [int]: Stop after this many points have been evaluated (by sim_f) in current run
- 'gen_max' [int]: Stop after this many points have been generated by gen_f in current run
- 'elapsed_wallclock_time' [float]: Stop after this amount of seconds have elapsed (since the libEnsemble manager has been initialized)
- 'stop_val' [(str, float)]: Stop when H[str] (for some field str returned from sim_out or gen_out) has been observed with a value less than the float given

### 6.8 alloc_specs

Allocation function specifications to be set in user calling script and passed to libE.libE():

<table>
<thead>
<tr>
<th>alloc_specs: [dict, optional]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Required keys:</strong></td>
</tr>
<tr>
<td>- 'alloc_f' [func]: Default: give_sim_work_first</td>
</tr>
<tr>
<td><strong>Optional keys:</strong></td>
</tr>
<tr>
<td>- 'out' [list of tuples]: Default: [('allocated', bool)]</td>
</tr>
</tbody>
</table>

**Notes**
- The alloc_specs has the default keys as given above, but may be overridden by the user.
- The tuples defined in the ‘out’ list are entered into the master history array

**Examples**

### 6.9 persis_info

Supply persistent information to libEnsemble:

<table>
<thead>
<tr>
<th>persis_info: [dict]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dictionary containing persistent info</td>
</tr>
</tbody>
</table>

Holds data that is passed to and from workers updating some state information. A typical example is a random number generator to be used in consecutive calls to a generator.

If worker i sends back persis_info, it is stored in persis_info[i]. This functionality can be used to, for example, pass a random stream back to the manager to be included in future work from the allocation function.

**Examples**

From: libEnsemble/tests/regression_tests/test_6-hump_camel_aposmm_LD_MAA.py:
persis_info = {'next_to_give':0}  # used in alloc_funcs/fast_alloc_to_aposmm.py to store the next entry in H to give
persis_info['total_gen_calls'] = 0  # used in alloc_funcs/fast_alloc_to_aposmm.py to count total gen calls

for i in range(MPI.COMM_WORLD.Get_size()):
    persis_info[i] = {'rand_stream': np.random.RandomState(i)}  # used as a random number stream for each worker
libEnsemble requires functions for generation, simulation and allocation.
While libEnsemble provides a default allocation function, the sim and gen functions must be provided. The required API and examples are given here.

7.1 Sim, Gen, and Alloc functions API

7.1.1 sim_f API

The sim_f function will be called by libEnsemble with the following API:

```python
out = sim_f(H[sim_specs['in']][sim_ids_from_allocf], persis_info, sim_specs, libE_info)
```

Parameters:

- **H**: numpy structured array *(example)*
- **persis_info**: dict *(example)*
- **sim_specs**: dict *(example)*
- **libE_info**: dict *(example)*

Returns:

- **H**: numpy structured array with keys/value-sizes matching those in sim_specs['out']. *(example)*
- **persis_info**: dict *(example)*
- **calc_tag**: int, optional Used to tell manager why a persistent worker is stopping.

--- Tags

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNSET_TAG</td>
<td>0  # sh temp - this is a libe feature that is to be reviewed for best solution</td>
</tr>
<tr>
<td>EVAL_SIM_TAG</td>
<td>1</td>
</tr>
<tr>
<td>EVAL_GEN_TAG</td>
<td>2</td>
</tr>
<tr>
<td>STOP_TAG</td>
<td>3</td>
</tr>
<tr>
<td>PERSIS_STOP</td>
<td>4  # manager tells persistent worker to desist</td>
</tr>
<tr>
<td>FINISHED_PERSISTENT_SIM_TAG</td>
<td>11  # tells manager sim_f done persistent mode</td>
</tr>
</tbody>
</table>
7.1.2 gen_f API

The gen_f calculations will be called by libEnsemble with the following API:

```python
out = gen_f(H[gen_specs['in']][sim_ids_from_allocf], persis_info, gen_specs, libE_info)
```

**Parameters:**

- `H`: numpy structured array *(example)*
- `persis_info`: dict *(example)*
- `gen_specs`: dict *(example)*
- `libE_info`: dict *(example)*

**Returns:**

- `H`: numpy structured array with keys/value-sizes matching those in `gen_specs['out']`. *(example)*
- `persis_info`: dict *(example)*
- `calc_tag`: int, optional Used to tell manager why a persistent worker is stopping.

```python
# --- Tags
UNSET_TAG = 0  # sh temp - this is a libe feature that is to be reviewed for best_solution
EVAL_SIM_TAG = 1
EVAL_GEN_TAG = 2
STOP_TAG = 3
PERSIS_STOP = 4  # manager tells persistent worker to desist
FINISHED_PERSISTENT_SIM_TAG = 11  # tells manager sim_f done persistent mode
```

7.1.3 alloc_f API

The alloc_f calculations will be called by libEnsemble with the following API:

```python
Work, persis_info = alloc_f(W, H, sim_specs, gen_specs, alloc_specs, persis_info)
```

**Parameters:**

- `W`: numpy structured array *(example)*
- `H`: numpy structured array *(example)*
- `sim_specs`: dict *(example)*
- `gen_specs`: dict *(example)*
- `alloc_specs`: dict *(example)*
- `persis_info`: dict *(example)*
Returns:

- **Work**: `dict` Dictionary with integer keys `$i$` for work to be send to worker `$i$`. *(example)*
- **persis_info**: `dict` *(example)*

## 7.2 Example User Funcs

Example gen, sim and alloc functions for `libEnsemble`.

### 7.2.1 Generation Functions

Below are example generation functions available in `libEnsemble`.

**Important**: See the API for generation functions [here](#).

#### uniform_sampling

This module contains multiple generation functions for sampling a domain. All use (and return) a random stream in `persis_info`, given by the allocation function.

```python
uniform_sampling.uniform_random_sample(H, persis_info, gen_specs, _)
```

Generates `$gen_specs['gen_batch_size']$` points uniformly over the domain defined by `$gen_specs['ub']$` and `$gen_specs['lb']$`. See `libensemble/tests/regression_tests/test_6-hump_camel_uniform_sampling.py`.

```python
uniform_sampling.uniform_random_sample_obj_components(H, persis_info, gen_specs, _)
```

Generates points uniformly over the domain defined by `$gen_specs['ub']$` and `$gen_specs['lb']$` but requests each `$obj_component$` be evaluated separately. See `libensemble/tests/regression_tests/test_chwirut_uniform_sampling_one_residual_at_a_time.py`.

```python
uniform_sampling.uniform_random_sample_with_different_nodes_and_ranks(H, persis_info, gen_specs, _)
```

Generates points uniformly over the domain defined by `$gen_specs['ub']$` and `$gen_specs['lb']$`. Also randomly requests a different `$number_of_nodes$` and `$ranks_per_node$` to be used in the evaluation of the generated point. See `libensemble/tests/regression_tests/test_6-hump_camel_with_different_nodes_uniform_sample.py`.

#### APOSMM

This module contains methods used our implementation of the Asynchronously Parallel Optimization Solver for finding Multiple Minima (APOSMM) method described in detail in the paper [https://doi.org/10.1007/s12532-017-0131-4](https://doi.org/10.1007/s12532-017-0131-4).
APOSMM coordinates multiple local optimization runs, starting from points which do not have a better point nearby (within a distance $r_k$). This generation function produces/requires the following fields in $H$:

- 'x' [n floats]: Parameters being optimized over
- 'x_on_cube' [n floats]: Parameters scaled to the unit cube
- 'f' [float]: Objective function being minimized
- 'local_pt' [bool]: True if point from a local optimization run
- 'dist_to_unit_bounds' [float]: Distance to domain boundary
- 'dist_to_better_l' [float]: Dist to closest better local opt point
- 'dist_to_better_s' [float]: Dist to closest better sample point
- 'ind_of_better_l' [int]: Index of point 'dist_to_better_l' away
- 'ind_of_better_s' [int]: Index of point 'dist_to_better_s' away
- 'started_run' [bool]: True if point has started a local opt run
- 'num_active_runs' [int]: Number of active local runs point is in
- 'local_min' [float]: True if point has been ruled a local minima
- 'sim_id' [int]: Row number of entry in history

and optionally

- 'priority' [float]: Value quantifying a point’s desirability
- 'f_i' [float]: Value of ith objective component (if single_component)
- 'fvec' [m floats]: All objective components (if calculated together)
- 'obj_component' [int]: Index corresponding to value in 'f_i'
- 'pt_id' [int]: Identify the point (useful when evaluating different objective components for a given 'x')

When using libEnsemble to do individual objective component evaluations, APOSMM will return `gen_specs['components']` copies of each point, but the component=0 entry of each point will only be considered when

- deciding where to start a run,
- best nearby point,
- storing the order of the points is the run
- storing the combined objective function value
- etc

Necessary quantities in `gen_specs` are:

- 'lb' [n floats]: Lower bound on search domain
- 'ub' [n floats]: Upper bound on search domain
- 'initial_sample_size' [int]: Number of uniformly sampled points must be returned (non-nan value) before a local opt run is started
- 'localopt_method' [str]: Name of an NLopt, PETSc/TAO, or SciPy method (see ‘advance_local_run’ below for supported methods)
Optional `gen_specs` entries are:

- `'sample_points'` [numpy array]: Points to be sampled (original domain)
- `'combine_component_func'` [func]: Function to combine obj components
- `'components'` [int]: Number of objective components
- `'dist_to_bound_multiple'` [float in (0,1]]: What fraction of the distance to the nearest boundary should the initial step size be in localopt runs
- `'high_priority_to_best_localopt_runs'`: [bool]: True if localopt runs with smallest observed function value are given priority
- `'lhs_divisions'` [int]: Number of Latin hypercube sampling partitions (0 or 1 results in uniform sampling)
- `'min_batch_size'` [int]: Lower bound on the number of points given every time APOSMM is called
- `'mu'` [float]: Distance from the boundary that all localopt starting points must satisfy
- `'nu'` [float]: Distance from identified minima that all starting points must satisfy
- `'single_component_at_a_time'` [bool]: True if single objective components will be evaluated at a time
- `'rk_const'` [float]: Multiplier in front of the r_k value
- `'max_active_runs'` [int]: Bound on number of runs APOSMM is advancing

And `gen_specs` convergence tolerances for NLopt, PETSc/TAO, SciPy

- `'fatol'` [float]:
- `'ftol_abs'` [float]:
- `'ftol_rel'` [float]:
- `'gatol'` [float]:
- `'grtol'` [float]:
- `'xtol_abs'` [float]:
- `'xtol_rel'` [float]:
- `'tol'` [float]:

As a default, APOSMM starts a local optimization runs from a point that:

- is not in an active local optimization run,
- is more than `mu` from the boundary (in the unit-cube domain),
- is more than `nu` from identified minima (in the unit-cube domain),
- does not have a better point within a distance `r_k` of it.

If the above results in more than `'max_active_runs'` being advanced, the best point in each run is determined and the dist_to_better is computed (with inf being the value for the best run). Then those `'max_active_runs'` runs with largest dist_to_better are advanced (breaking ties arbitrarily).

**Note** `gen_specs['combine_component_func']` must be defined when there are multiple objective components.

**Note** APOSMM critically uses `persis_info` to store information about active runs, order of points in each run, etc. The allocation function must ensure it’s always given.
See `libensemble/tests/regression_tests/test_branin_aposmm.py` for basic APOSMM usage.

See `libensemble/tests/regression_tests/test_chwirut_aposmm_one_residual_at_a_time.py` for an example of APOSMM coordinating multiple local optimization runs for an objective with more than one component.

**aposmm.initialize_APOSMM** \((H, \text{gen_specs})\)

Computes common values every time that APOSMM is reinvoked

See `/libensemble/alloc_funcs/start_persistent_local_opt_gens.py`

**aposmm.decide_where_to_start_localopt** \((H, r_k, \mu=0, \nu=0, \gamma_{\text{quantile}}=1)\)

Finds points in the history that satisfy the conditions (S1-S5 and L1-L8) in Table 1 of the APOSMM paper. This method first identifies sample points satisfying S2-S5, and then identifies all localopt points that satisfy L1-L7. We then start from any sample point also satisfying S1. We do not check condition L8 currently.

We don’t consider points in the history that have not returned from computation, or that have a `nan` value. Also, note that \(\mu\) and \(\nu\) implicitly depend on the scaling that is happening with the domain. That is, adjusting the initial domain can make a run start (or not start) at a point that didn’t (or did) previously.

**Parameters**

- **H** *(numpy structured array)* – History array storing rows for each point.
- **r_k_const** *(float)* – Radius for deciding when to start runs
- **lhs_divisions** *(integer)* – Number of Latin hypercube sampling divisions (0 or 1 means uniform random sampling over the domain)
- **mu** *(nonnegative float)* – Distance from the boundary that all starting points must satisfy
- **nu** *(nonnegative float)* – Distance from identified minima that all starting points must satisfy
- **gamma_quantile** *(float in \((0,1]\))* – Only sample points whose function values are in the lower gamma_quantile can start localopt runs

**Returns**

- **start_inds** – Indices where a local opt run should be started

**Return type** list

See `/libensemble/alloc_funcs/start_persistent_local_opt_gens.py`

**aposmm.update_history_dist** \((H, n, \text{gen_specs}, c\_flag)\)

Updates distances/indices after new points that have been evaluated.

See `/libensemble/alloc_funcs/start_persistent_local_opt_gens.py`

**uniform_or_localopt**

**uniform_or_localopt.uniform_or_localopt** \((H, \text{persis_info}, \text{gen_specs}, \text{libE_info})\)

This generation function returns `gen_specs['gen_batch_size']` uniformly sampled points when called in nonpersistent mode (i.e., when `libE_info['persistent']` isn’t True). Otherwise, the generation function a persistent nlopt local optimization run.

See `libensemble/tests/regression_tests/test_6-hump_camel_uniform_sampling_with_persistent_localopt_gens.py`
persistent_uniform_sampling

```
persistent_uniform_sampling.persistent_uniform(H, persis_info, gen_specs, libE_info)
```

This generation function always enters into persistent mode and returns `gen_specs['gen_batch_size']`
uniformly sampled points.

See libensemble/libensemble/tests/regression_tests/
test_6-hump_camel_persistent_uniform_sampling.py

### 7.2.2 Simulation Functions

Below are example simulation functions available in libEnsemble.

**Important:** See the API for simulation functions [here](#)

**six_hump_camel**

This module contains various versions that evaluate the six hump camel function.

```
six_hump_camel.six_hump_camel_with_different_ranks_and_nodes(H, persis_info, 
              sim_specs, libE_info)
```

Evaluates the six hump camel for a collection of points given in `H['x']`, but also performs a system call with
a given number of nodes and ranks per node using a machinefile (to show one way of evaluating a compiled
simulation).

See /libensemble/tests/regression_tests/test_6-hump_camel_with_different_nodes_uniform_sample.py

```
six_hump_camel.six_hump_camel(H, persis_info, sim_specs, _)
```

Evaluates the six hump camel function for a collection of points given in `H['x']`. Additionally evaluates the
gradient if `grad` is a field in `sim_specs['out']` and pauses for `sim_specs['pause_time']` if defined.

See /libensemble/libensemble/tests/regression_tests/
test_6-hump_camel_aposmm_LD_MMA.py

```
six_hump_camel.six_hump_camel_simple(x, persis_info, sim_specs, _)
```

Evaluates the six hump camel function for a single point `x`.

See /libensemble/libensemble/tests/regression_tests/
test_fast_alloc.py

**chwirut**

```
chwirut1.chwirut_eval(H, persis_info, sim_specs, _)
```

Evaluates the chwirut objective function at a given set of points in `H['x']`. If `obj_component` is a field in `sim_specs['out']`, only that component of the objective will be evaluated. Otherwise, all 214 components are evaluated and returned in the `fvec` field.

See /libensemble/tests/regression_tests/test_chwirut_pounders.py for an example where the entire fvec is computed.

See /libensemble/tests/regression_tests/test_chwirut_aposmm_one_residual_at_a_time.py
job_control_hworld

job_control_hworld

Test of launching and polling job and exiting on job finish

7.2.3 Allocation Functions

Below are example allocation functions available in libEnsemble.

Important: See the API for allocation functions [here](#).

Note: The default alloc_func is give_sim_work_first.

give_sim_work_first

give_sim_work_first

Decide what should be given to workers. This allocation function gives any available simulation work first, and only when all simulations are completed or running does it start (at most gen_specs['num_active_gens']) generator instances.

Allows for a gen_specs['batch_mode'] where no generation work is given out unless all entries in H are returned.

Allows for blocking of workers that are not active, for example, so their resources can be used for a different simulation evaluation.

Can give points in highest priority, if 'priority' is a field in H.

This is the default allocation function if one is not defined.

See /libensemble/tests/regression_tests/test_6-hump_camell_uniform_sampling.py

fast_alloc

fast_alloc

This allocation function gives (in order) entries in H to idle workers to evaluate in the simulation function. The fields in sim_specs['in'] are given. If all entries in H have been given a be evaluated, a worker is told to call the generator function, provided this wouldn’t result in more than gen_specs['num_active_gen'] active generators.

See /libensemble/tests/regression_tests/test_fast_alloc.py

fast_alloc_to_aposmm

fast_alloc_to_aposmm

This allocation function gives (in order) entries in H to idle workers to evaluate in the simulation function. The fields in sim_specs['in'] are given. If all entries in H have been given a be evaluated, a worker is told to
call the generator function, provided this wouldn’t result in more than `gen_specs['num_active_gen']` active generators. Also allows for a ‘batch_mode’.

See /libensemble/tests/regression_tests/test_6-hump_camel_aosmm_LD_MMA.py

**start_only_persistent**

`start_only_persistent.only_persistent_gens(W, H, sim_specs, gen_specs, alloc_specs, persis_info)`

This allocation function will give simulation work if possible, but otherwise start up to 1 persistent generator. If all points requested by the persistent generator have been returned from the simulation evaluation, then this information is given back to the persistent generator.

See /libensemble/tests/regression_tests/test_6-hump_camel_persistent_uniform_sampling.py

**start_persistent_local_opt_gens**

`start_persistent_local_opt_gens.start_persistent_local_opt_gens(W, H, sim_specs, gen_specs, alloc_specs, persis_info)`

This allocation function will:

- Start up a persistent generator that is a local opt run at the first point identified by APOSMM’s `decide_where_to_start_localopt`.
- It will only do this if at least one worker will be left to perform simulation evaluations.
- If multiple starting points are available, the one with smallest function value is chosen.
- If no candidate starting points exist, points from existing runs will be evaluated (oldest first).
- If no points are left, call the generation function.

See /libensemble/tests/regression_tests/test_6-hump_camel_uniform_sampling_with_persistent_localopt_gens.py
CHAPTER EIGHT

JOB CONTROLLER

The job controller can be used with simulation functions to provide a simple, portable interface for running and managing user jobs.

8.1 Job Controller Overview

The Job Controller module can be used by the worker or user-side code to issue and manage jobs using a portable interface. Various back-end mechanisms may be used to implement this interface on the system, either specified by the user at the top-level, or auto-detected. The job_controller manages jobs using the launch, poll and kill functions. Job attributes can then be queried to determine status. Functions are also provided to access and interrogate files in the job’s working directory.

At the top-level calling script, a job_controller is created and the executable gen or sim applications are registered to it (these are applications that will be runnable jobs). If an alternative job_controller, such as Balsam, is to be used, then these can be created as in the example. Once in the user-side worker code (sim/gen func), an MPI based job_controller can be retrieved without any need to specify the specific type.

**Example usage (code runnable with or without a Balsam backend):**

In calling function:

```python
sim_app = '/path/to/my/exe'
USE_BALSAM = False

if USE_BALSAM:
    from libensemble.balsam_controller import BalsamJobController
    jobctrl = BalsamJobController()
else:
    from libensemble.mpi_controller import MPIJobController
    jobctrl = MPIJobController()

jobctrl.register_calc(full_path=sim_app, calc_type='sim')
```

In user sim func:

```python
jobctl = MPIJobController.controller # This will work for inherited controllers also
       # (e.g., Balsam)
import time

jobctl = MPIJobController.controller # Will return controller (whether Balsam or
       # standard MPI).
job = jobctl1.launch(calc_type='sim', num_procs=8, app_args='input.txt', stdout='out.
       # txt', stderr='err.txt')
```

(continues on next page)
while time.time() - start < timeout_sec:
    time.sleep(delay)

    # Has manager sent a finish signal
    jobctl.manager_poll()
    if jobctl.manager_signal == 'finish':
        job.kill()

    # Poll job to see if completed
    job.poll()
    if job.finished:
        print(job.state)
        break

    # Check output file for error and kill job
    if job.stdout_exists():
        if 'Error' in job.read_stdout():
            job.kill()
        break

See the job_controller interface for API.

For a more detailed working example see:

- libensemble/tests/regression_tests/test_jobcontroller_hworld.py
  which uses sim function:
  - libensemble/sim_funcs/job_control_hworld.py

## 8.2 Job Controller Module

Module to launch and control running jobs.

Contains job_controller and job. The class JobController is a base class and not intended for direct use. Instead one of the inherited classes should be used. Inherited classes include MPI and Balsam variants. A job_controller can create and manage multiple jobs. The worker or user-side code can issue and manage jobs using the launch, poll and kill functions. Job attributes are queried to determine status. Functions are also provided to access and interrogate files in the job’s working directory.

See example for usage.

See the controller APIs for optional arguments.

### 8.2.1 MPI Job Controller

To create an MPI job controller, the calling script should contain:

```python
jobctl = MPIJobController()
```

See the controller API below for optional arguments.

Module to launch and control running MPI jobs.
class mpi_controller.MPIJobController(auto_resources=True, central_node=False, nodelist_env_slurm=None, nodelist_env_cobalt=None, nodelist_env_lsf=None)

Bases: libensemble.controller.JobController

The MPI job_controller can create, poll and kill runnable MPI jobs

default_app(calc_type)
    Get the default app for a given calc type.

gen_default_app
    Return the default generator app.

get_job(jobid)
    Returns the job object for the supplied job ID

kill(job)
    Kill a job

launch(calc_type, num_procs=None, num_nodes=None, ranks_per_node=None, machinefile=None, app_args=None, stdout=None, stderr=None, stage_inout=None, hyperthreads=False, test=False, wait_on_run=False)
    Creates a new job, and either launches or schedules launch.
The created job object is returned.

Parameters

- calc_type (String) – The calculation type: ‘sim’ or ‘gen’
- num_procs (int, optional) – The total number of MPI tasks on which to launch the job.
- num_nodes (int, optional) – The number of nodes on which to launch the job.
- ranks_per_node (int, optional) – The ranks per node for this job.
- machinefile (string, optional) – Name of a machinefile for this job to use.
- app_args (string, optional) – A string of the application arguments to be added to job launch command line.
- stdout (string, optional) – A standard output filename.
- stderr (string, optional) – A standard error filename.
- stage_inout (string, optional) – A directory to copy files from. Default will take from current directory.
- hyperthreads (boolean, optional) – Whether to launch MPI tasks to hyperthreads
- test (boolean, optional) – Whether this is a test - No job will be launched. Instead runline is printed to logger (At INFO level).
- wait_on_run (boolean, optional) – Whether to wait for job to be polled as RUNNING (or other active/end state) before continuing.

Returns job – The launched job object.

Return type obj: Job

Note that if some combination of num_procs, num_nodes and ranks_per_node are provided, these will be honored if possible. If resource detection is on and these are omitted, then the available resources will be divided amongst workers.
manager_poll (comm)
Polls for a manager signal

The job controller manager_signal attribute will be updated.

poll (job)
Polls a job

register_calc (full_path, calc_type='sim', desc=None)
Registers a user application to libEnsemble

Parameters
- **full_path (String)** – The full path of the user application to be registered.
- **calc_type (String)** – Calculation type: Is this application part of a ‘sim’ or ‘gen’
  function.
- **desc (String, optional)** – Description of this application.

set_workerID (workerid)
Sets the worker ID for this job_controller

set_worker_info (comm, workerid=None)
Sets info for this job_controller

sim_default_app
Return the default simulation app.

8.2.2 Balsam Job Controller

To create a Balsam job controller, the calling script should contain:

```python
jobctr = BalsamJobController()
```

The Balsam job controller inherits from the MPI job controller. See the `MPIJobController` for shared API. Any
differences are shown below.

Module to launch and control running jobs with Balsam.

class balsam_controller.BalsamJobController (auto_resources=True,  
  central_mode=True, nodelist_env_slurm=None,  
  nodelist_env_cobalt=None)

Bases: libensemble.mpi_controller.MPIJobController

Inherits from MPIJobController and wraps the Balsam job management service

**Note:** Job kills are not configurable in the Balsam job_controller.

class balsam_controller.BalsamJob (app=None, app_args=None, workdir=None, stdout=None,  
  stderr=None, workerid=None)

Bases: libensemble.controller.Job
Wraps a Balsam Job from the Balsam service.
The same attributes and query routines are implemented.
8.2.3 Job Class

Jobs are created and returned through the job_controller launch function. Jobs can be polled and killed with the respective poll and kill functions. Job information can be queried through the job attributes below and the query functions. Note that the job attributes are only updated when they are polled/killed (or through other job or job controller functions).

```python
class controller.Job(app=None, app_args=None, workdir=None, stdout=None, stderr=None, workerid=None):
    Manage the creation, configuration and status of a launchable job.

    file_exists_in_workdir(filename)
        Returns True if the named file exists in the job’s workdir

    kill(wait_time=60)
        Kills or cancels the supplied job

        Sends SIGTERM, waits for a period of <wait_time> for graceful termination, then sends a hard kill with SIGKILL. If <wait_time> is 0, we go immediately to SIGKILL; if <wait_time> is None, we never do a SIGKILL.

    poll()
        Polls and updates the status attributes of the job

    read_file_in_workdir(filename)
        Open and reads the named file in the job’s workdir

    read_stderr()
        Open and reads the job’s stderr file in the job’s workdir

    read_stdout()
        Open and reads the job’s stdout file in the job’s workdir

    stderr_exists()
        Returns True if the job’s stderr file exists in the workdir

    stdout_exists()
        Returns True if the job’s stdout file exists in the workdir

    workdir_exists()
        Returns True if the job’s workdir exists
```

8.2.4 Job Attributes

Following is a list of job status and configuration attributes that can be retrieved from a job.

NOTE These should not be set directly. Jobs are launched by the job controller and job information can be queried through the job attributes below and the query functions.

Job Status attributes include:

- **job.state** (string) The job status. One of: (`UNKNOWN`|`CREATED`|`WAITING`|`RUNNING`|`FINISHED`|`USER_KILLED`)
- **job.process** (process obj) The process object used by the underlying process manager (e.g., return value of subprocess.Popen)
- **job.errcode** (int) The errorcode/return code used by the underlying process manager
- **job.finished** (Boolean) True means job has finished running - not whether was successful
- **job.success** (Boolean) Did job complete successfully (e.g., returncode is zero)
- **job.runtime** (int) Time in seconds that job has been running.
job.total_time (int) Total time from job submission to completion (only available when job is finished).

Run configuration attributes - Some will be auto-generated:

job.workdir (string) Work directory for the job

job.name (string) Name of job - auto-generated

job.app (app obj) Use application/executable, registered using jobctl.register_calc

job.app_args (string) Application arguments as a string

job.stdout (string) Name of file where the standard output of the job is written (in job.workdir)

job.stderr (string) Name of file where the standard error of the job is written (in job.workdir)

A list of job_controller and job functions can be found under the Job Controller Module.
The libEnsemble logger uses the standard Python logging levels (DEBUG, INFO, WARNING, ERROR, CRITICAL). The default level is INFO, which includes information such as how jobs are launched and when jobs are killed. To gain additional diagnostics, logging level can be set to DEBUG. libEnsemble produces logging to the file ensemble.log by default. A log file name can also be supplied.

E.g. To change the logging level to DEBUG, provide the following in your the calling scripts:

```python
from libensemble import libE_logger
libE_logger.set_level('DEBUG')
```

### 9.1 Logging API

- `libE_logger.get_level()`  
  Return libEnsemble logging level

- `libE_logger.set_filename(filename)`  
  Sets logger filename if loggers not yet created, else None

- `libE_logger.set_level(level)`  
  Set libEnsemble logging level
LIBENSEMBLE INTERNAL MODULES

Internal modules of libEnsemble.

10.1 Manager Module

10.1.1 libEnsemble manager routines

class libE_manager.Manager(hist, libE_specs, alloc_specs, sim_specs, gen_specs, exit_criteria, wcomms=[])

Manager class for libensemble.

run(persis_info)
Run the manager.

term_test(logged=True)
Check termination criteria

term_test_gen_max(gen_max)
Check against max generator calls.

term_test_sim_max(sim_max)
Check against max simulations

term_test_stop_val(stop_val)
Check against stop value criterion.

term_test_wallclock(max_elapsed)
Check against wallclock timeout

worker_dtype = [('worker_id', <class 'int'>), ('active', <class 'int'>), ('persis_state', <class 'int'>), ('blocked', <class 'bool'>)]

exception libE_manager.ManagerException
Exception at manager, raised on abort signal from worker

libE_manager.filter_nans(array)
Filter out NaNs from a numpy array.

libE_manager.manager_main(hist, libE_specs, alloc_specs, sim_specs, gen_specs, exit_criteria, persis_info, wcomms=[])
Manager routine to coordinate the generation and simulation evaluations
10.2 Worker Module

10.2.1 libEnsemble worker class

libE_worker.worker_main(comm, sim_specs, gen_specs, workerID=None, log_comm=True)

Evaluate calculations given to it by the manager.

Creates a worker object, receives work from manager, runs worker, and communicates results. This routine also creates and writes to the workers summary file.

Parameters

- **comm** (comm object for manager communications) –
- **sim_specs** (dict with parameters/information for simulation calculations) –
- **gen_specs** (dict with parameters/information for generation calculations) –
- **workerID** (manager assigned worker ID (if None, default is comm.rank)) –

class libE_worker.Worker(comm, dtypes, workerID, sim_specs, gen_specs)

The Worker Class provides methods for controlling sim and gen funcs

run()
Run the main worker loop.

10.3 Resources Module

Module for detecting and returning system resources

class resources.Resources(top_level_dir=None, central_mode=False, launcher=None, nodelist_env_slurm=None, nodelist_env_cobalt=None, nodelist_env_lsf=None)

Provide system resources to libEnsemble and job controller.

This is initialised when the job_controller is created with auto_resources set to True.

Class Attributes:

Variables

- **default_nodelist_env_slurm** (string) – Default SLRUM nodelist environment variable
- **default_nodelist_env_cobalt** (string) – Default Cobal nodelist environment variable

Object Attributes:

These are set on initialisation.

Variables

- **top_level_dir** (string) – Directory where searches for worker_list file.
- **central_mode** (boolean) – If true, then running in central mode, else distributed.
- **nodelist_env_slurm** (string) – Slurm environment variable giving node-list.
• `nodelist_env_cobalt` *(string)* – Cobalt environment variable giving node-list.

• `global_nodelist` *(list)* – A list of all nodes available for running user applications.

• `num_workers` *(int)* – Total number of workers.

• `logical_cores_avail_per_node` *(int)* – Logical cores (including SMT threads) available on a node.

• `physical_cores_avail_per_node` *(int)* – Physical cores available on a node.

• `workerID` *(int)* – workerID.

• `local_nodelist` *(list)* – A list of all nodes assigned to this worker.

• `local_node_count` *(int)* – The number of nodes available to this worker (rounded up to whole number).

• `workers_per_node` *(int)* – The number of workers per node (if using sub-node workers).

```python
__init__(top_level_dir=None, central_mode=False, launcher=None, nodelist_env_slurm=None, nodelist_env_cobalt=None, nodelist_env_lsf=None)
```

Initialise new Resources instance.

Works out the compute resources available for current allocation, including node list and cores/hardware threads available within nodes.

**Parameters**

• `top_level_dir` *(string, optional:)* – Directory libEnsemble runs in (default is current working directory).

• `optional` *(central_mode,)* – If true, then running in central mode, else distributed. Central mode means libE processes (manager and workers) are grouped together and do not share nodes with applications. Distributed mode means Workers share nodes with applications.

• `launcher` *(String, optional)* – The name of the job launcher such as mpirun or aprun. This may be used to obtain intra-node information by launching a probing job onto the compute nodes. If not present, the local node will be used to obtain this information.

• `nodelist_env_slurm` *(String, optional)* – The environment variable giving a node list in Slurm format (Default: Uses SLURM_NODELIST) Note: This is only queried if a worker_list file is not provided and auto_resources=True.

• `nodelist_env_cobalt` *(String, optional)* – The environment variable giving a node list in Cobalt format (Default: Uses COBALT_PARTNAME) Note: This is only queried if a worker_list file is not provided and auto_resources=True.

• `nodelist_env_lsf` *(String, optional)* – The environment variable giving a node list in LSF format (Default: Uses LSB_HOSTS) Note: This is only queried if a worker_list file is not provided and auto_resources=True.

```python
static get_libE_nodes()
```

Returns a list of nodes running libE workers.

```python
static get_MPI_variant()
```

Returns MPI base implementation.

• Returns `mpi_variant` – MPI variant ‘aprun’ or ‘jsrun’ or ‘mpich’ or ‘openmpi’.

• Return type  `string:`


**static get_slurm_nodelist** *(node_list_env)*

Get global libEnsemble nodelist from the Slurm environment

**static get_cobalt_nodelist** *(node_list_env)*

Get global libEnsemble nodelist from the Cobalt environment

**static get_lsf_nodelist** *(node_list_env)*

Get global libEnsemble nodelist from the LSF environment

**static remove_nodes** *(global_nodelist_in, remove_list)*

Any nodes in remove_list are removed from the global nodelist

**static best_split** *(a, n)*

Create the most even split of list a into n parts and return list of lists

**static get_global_nodelist** *(rundir=None, node_list_env_slurm=None, node_list_env_cobalt=None, node_list_env_lsf=None)*

Return the list of nodes available to all libEnsemble workers

If a worker_list file exists this is used, otherwise the environment is interrogated for a node list. If a
dedicated manager node is used, then a worker_list file is recommended.

In central mode, any node with a libE worker is removed from the list.
CHAPTER

ELEVEN

INDICES AND TABLES

• genindex
• modindex
• search
a
aposmm, 25

b
balsam_controller, 36

c
chwirut1, 29
ccontroller, 34

f
fast_alloc, 30
fast_alloc_to_aposmm, 30

g
give_sim_work_first, 30

j
job_control_hworld, 30

l
libE, 15
libE_logger, 39
libE_manager, 41
libE_worker, 42

m
mpi_controller, 34

p
persistent_uniform_sampling, 29

r
resources, 42

s
six_hump_camel, 29
start_only_persistent, 31
start_persistent_local_opt_gens, 31

u
uniform_or_localopt, 28
uniform_sampling, 25
Symbols

__init__() (resources.Resources method), 43

A
aposmm (module), 25
aposmm_logic() (in module aposmm), 25

B
balsam_controller (module), 36
BalsamJob (class in balsam_controller), 36
BalsamJobController (class in balsam_controller), 36
best_split() (resources.Resources static method), 44

c
chwirut1 (module), 29
chwirut_eval() (in module chwirut1), 29
controller (module), 34

D
decide_where_to_start_localopt() (in module aposmm), 28
default_app() (mpi_controller.MPIJobController method), 35

F
fast_alloc (module), 30
fast_alloc_to_aposmm (module), 30
file_exists_in_workdir() (controller.Job method), 37
filter_nans() (in module libE_manager), 41

G
gen_default_app (mpi_controller.MPIJobController attribute), 35
get_cobalt_nodelist() (resources.Resources static method), 44
get_global_nodelist() (resources.Resources static method), 44
get_job() (mpi_controller.MPIJobController method), 35
get_level() (in module libE_logger), 39
give_sim_work_first (module), 30
give_sim_work_first() (in module fast_alloc), 30
give_sim_work_first() (in module fast_alloc_to_aposmm), 30
give_sim_work_first() (in module give_sim_work_first), 30

I
initialize_APOSMM() (in module aposmm), 28

J
Job (class in controller), 37
job_control_hworld (module), 30
job_control_hworld() (in module job_control_hworld), 30

K
kill() (controller.Job method), 37
kill() (mpi_controller.MPIJobController method), 35

L
launch() (mpi_controller.MPIJobController method), 35

M
Manager (class in libE_manager), 41
manager_main() (in module libE_manager), 41
manager_poll() (mpi_controller.MPIJobController method), 35
ManagerException, 41
mpi_controller (module), 34
MPIJobController (class in mpi_controller), 34

O
only_persistent_gens() (in module start_only_persistent), 31

P
persistent_uniform() (in module persistent_uniform_sampling), 29
poll() (controller.Job method), 37
poll() (mpi_controller.MPIJobController method), 36

R
read_file_in_workdir() (controller.Job method), 37
read_stderr() (controller.Job method), 37
read_stdout() (controller.Job method), 37
register_calc() (mpi_controller.MPIJobController method), 36
remove_nodes() (resources.Resources static method), 44
Resources (class in resources), 42
resources (module), 42
run() (libE_manager.Manager method), 41
run() (libE_worker.Worker method), 42

S
set_filename() (in module libE_logger), 39
set_level() (in module libE_logger), 39
set_worker_info() (mpi_controller.MPIJobController method), 36
set_workerID() (mpi_controller.MPIJobController method), 36
sim_default_app (mpi_controller.MPIJobController attribute), 36
six_hump_camel (module), 29
six_hump_camel() (in module six_hump_camel), 29
six_hump_camel_simple() (in module six_hump_camel), 29
six_hump_camel_with_different_ranks_and_nodes() (in module six_hump_camel), 29
start_only_persistent (module), 31
start_persistent_local_opt_gens (module), 31
start_persistent_local_opt_gens() (in module start_persistent_local_opt_gens), 31
stderr_exists() (controller.Job method), 37
stdout_exists() (controller.Job method), 37

T
term_test() (libE_manager.Manager method), 41
term_test_gen_max() (libE_manager.Manager method), 41
term_test_sim_max() (libE_manager.Manager method), 41
term_test_stop_val() (libE_manager.Manager method), 41
term_test_wallclock() (libE_manager.Manager method), 41

U
uniform_or_localopt (module), 28
uniform_or_localopt() (in module uniform_or_localopt), 28
uniform_random_sample() (in module uniform_sampling), 25
uniform_random_sample_obj_components() (in module uniform_sampling), 25
uniform_random_sample_with_different_nodes_and_ranks() (in module uniform_sampling), 25
uniform_sampling (module), 25
update_history_dist() (in module aposmm), 28

W
workdir_exists() (controller.Job method), 37
Worker (class in libE_worker), 42
worker_dtype (libE_manager.Manager attribute), 41
worker_main() (in module libE_worker), 42