PyMOSO: Software for Multi-Objective Simulation Optimization with R-PERLE and R-MinRLE

Kyle Cooper
School of Industrial Engineering, Purdue University and Tata Consultancy Services, coope149@purdue.edu,

Susan R. Hunter
School of Industrial Engineering, Purdue University, susanhunter@purdue.edu,

We present the PyMOSO software package for (1) solving multi-objective simulation optimization (MOSO) problems on integer lattices, and (2) implementing and testing new simulation optimization (SO) algorithms. First, for solving MOSO problems on integer lattices, PyMOSO implements R-PERLE, a state-of-the-art algorithm for two objectives, and R-MinRLE, a competitive benchmark algorithm for three or more objectives. Both algorithms employ pseudo-gradients, are designed for sampling efficiency, and return solutions that, under appropriate regularity conditions, provably converge to a local efficient set with probability one as the simulation budget increases. PyMOSO can interface with existing simulation software and can obtain simulation replications in parallel. Second, for implementing and testing new SO algorithms, PyMOSO includes pseudo-random number stream management, implements algorithm testing with independent pseudo-random number streams run in parallel, and computes the performance of algorithms with user-defined metrics. For convenience, we also include an implementation of R-SPLINE for problems with one objective. The PyMOSO source code is available under a permissive open source license.

Key words: multi-objective simulation optimization, software
History: Authors' preprint compiled Saturday 8th June, 2019.

1. Introduction

We present the PyMOSO software package, written in Python, for using, implementing, and testing multi-objective simulation optimization (MOSO) algorithms. PyMOSO currently implements a state-of-the-art algorithm, R-PERLE, for solving MOSO problems on integer lattices with two objectives, and a competitive benchmark algorithm, R-MinRLE, for solving MOSO problems on integer lattices with many objectives. Since R-PERLE and R-MinRLE are both pseudo-gradient-based algorithms that rely on the single-objective SPLINE solver by Wang et al. (2013), for convenience, we also include an implementation of R-SPLINE (Wang et al. 2013) for solving single-objective simulation optimization problems on integer lattices. The initial version of the software, and this paper, serve as companions to Cooper et al. (2018), the paper that introduces and explains the R-PERLE and R-MinRLE algorithms.

MOSO problems are nonlinear optimization problems with more than one simultaneous objective function, each of which can only be observed with stochastic error. For example, at any feasible point, observations of the objective values may be obtained from a Monte Carlo simulation oracle.

By the nature of the algorithms currently included in PyMOSO, our focus is on MOSO problems in which the feasible set is a subset of an integer lattice. Such problems take the form

$$\text{Problem } M_d: \quad \underset{\mathbf{x}\in\mathcal{X}}{\text{minimize}} \; \{\mathbf{g}(\mathbf{x}) = (g_1(\mathbf{x}), ..., g_d(\mathbf{x})) := (\mathbb{E}[G_1(\mathbf{x}, \boldsymbol{\xi})], ..., \mathbb{E}[G_d(\mathbf{x}, \boldsymbol{\xi})])\},$$

where $\mathcal{X} \subseteq \mathbb{Z}^q$ is the feasible set and $\boldsymbol{\xi}$ is a random vector. R-PERLE is designed to solve Problem M_2 to local optimality, and R-MinRLE is designed to solve Problem M_d, $d \geq 2$, to local optimality. A local solution to Problem M_d is called a local efficient set, which we formally define in §1.2.

MOSO problems on integer lattices arise in many application domains including aviation, healthcare, transportation, and manufacturing (Hunter et al. 2019). For example, in aviation, Li et al. (2015) solve a bi-objective aircraft spare parts management problem, and Lee et al. (2008) solve a tri-objective inventory control problem for a network of airports. In healthcare, Chen and Wang (2016) solve a bi-objective capacity allocation problem for a hospital's emergency department. In transportation, Zhou et al. (2018) solve a bi-objective problem to reduce congestion in a lighterage terminal. Finally, in manufacturing, Andersson et al. (2007) solve a bi-objective problem to increase the throughput and maintain safety stocks for a camshaft production line. In these applications, the decision variables take on integer values, the objective functions can only be observed with stochastic error through a Monte Carlo simulation, and the goal is to retrieve the entire efficient set as input to the multi-criteria decision-making process.

Though MOSO problems arise in many application areas, few software packages exist to solve Problem M_d for $d \geq 2$ objectives. The software packages that do exist, such as OptQuest (OptTek Systems, Inc. 2018, Thengvall et al. 2016) and PaGMO/PyGMO (Biscani and Izzo 2018), tend to implement metaheuristics to solve MOSO problems, which do not provide performance guarantees (Hong et al. 2015). We are not aware of any available MOSO software that provides the sampling efficiency and convergence guarantees of R-PERLE and R-MinRLE, moreover, no software implementing these two algorithms currently exists.

Finally, we remark here that an implementation of an as-yet-unpublished algorithm, Multi-objective Partitioned Random Search, is available in a source code repository (Weizhi 2017).

1.1. Contributions

As in Schmeiser (2008), we adopt the terms 'practitioners' and 'researchers' to describe those who seek a solution to Problem M_d to aid in decision-making, and those who intend to create and compare MOSO algorithms, respectively. We discuss our contributions to each group.

PyMOSO provides practitioners with off-the-shelf access to the state-of-the-art, provably convergent, bi-objective solver R-PERLE, and to the competitive, provably convergent, multi-objective solver R-MinRLE. PyMOSO can accommodate any Monte Carlo simulation oracle that can be called from Python. Swain (2017) provides a list of proprietary simulation software packages and indicates whether each package can be invoked by an external program. Most simulation software packages that can be invoked by external programs are, with programming effort, compatible with PyMOSO. After the oracle has been implemented, PyMOSO can obtain simulation replications in parallel using common random numbers (CRN, see Law 2015) by exploiting the stream and substream capabilities of the pseudo-random number generator `mrg32k3a` (L'Ecuyer et al. 2002), which may reduce runtime. Finally, PyMOSO provides off-the-shelf access to R-SPLINE in the same software framework. While PyMOSO currently supports only integer decision variables, practitioners may wish to solve problems with continuous decision variables. We urge caution when discretizing continuous problems, as the choice of grid size is non-trivial. We hope that future versions of PyMOSO will include solvers for problems with continuous decision variables.

For researchers who intend to create and compare MOSO algorithms, PyMOSO offers two primary benefits, as follows.

1. Researchers designing new algorithms to solve MOSO problems on integer lattices should compare their algorithms with R-PERLE and R-MinRLE on a variety of test problems. PyMOSO enables researchers to compare algorithms by providing an interface for implementing test problems and calculating user-defined metrics. By default, PyMOSO includes all test problems and associated metrics from Cooper et al. (2018). To create additional test problems, we recommend taking inspiration from existing testbeds for deterministic multi-objective optimization (see, e.g., the references in Hunter et al. 2019, p. 25 – 26). Once the desired test problems are implemented, researchers may use PyMOSO to run many independent sample paths of the algorithms in parallel. PyMOSO

provides pseudo-random numbers using `mrg32k3a` (L'Ecuyer 1999) and random number stream management consistent with L'Ecuyer et al. (2002).

2. PyMOSO provides a framework that enables researchers to create and implement new algorithms. Although any new MOSO algorithm can be implemented in PyMOSO, it is especially easy to implement algorithms that rely on a version of sample average approximation called retrospective approximation (RA). (We provide a brief explanation of RA in §1.2; see Pasupathy and Ghosh (2013) for a more thorough explanation.) In particular, researchers can create new RA algorithms for MOSO by writing "accelerator" functions that provide starting points to the naïve search algorithm Relaxed Local Enumeration (RLE) in each RA iteration, as we describe in §3.3. Under appropriate regularity conditions, Cooper et al. (2018) prove that such algorithms converge to a local efficient set almost surely as the sample size increases.

In what follows, we discuss background concepts in §1.2, including optimality definitions, RA, and accelerators. Then, we provide introductions to the current version of PyMOSO for practitioners in §2 and for researchers in §3. Since all information provided for practitioners is relevant to researchers, we encourage researchers to read both sections. In addition to the online supplement, PyMOSO installation instructions, source code, and the user manual can be found at `https://github.com/pymoso/`.

1.2. Background

Before discussing PyMOSO in detail, for completeness, we provide a sense of the sets returned by R-PERLE and R-MinRLE, which we call estimated local efficient sets. Due to space constraints, we make this section compact. See Cooper et al. (2018) for a complete treatment.

To define a local efficient set (LES), we first define neighborhoods and dominated points. Let $d(\mathbf{x},\mathbf{x}') := ||\mathbf{x}-\mathbf{x}'||$ denote the Euclidean distance between two points $\mathbf{x},\mathbf{x}' \in \mathbb{R}^q$.

Definition 1. Given $a \in \mathbb{R}, a \geq 1$, the $\mathcal{N}_a$-neighborhood of a point $\mathbf{x} \in \mathbb{Z}^q$ is $\mathcal{N}_a(\mathbf{x}) := \{\mathbf{x}' \in \mathbb{Z}^q : d(\mathbf{x},\mathbf{x}') \leq a\}$, and the $\mathcal{N}_a$-neighborhood of a set $\mathcal{S} \subseteq \mathbb{Z}^q$ is $\mathcal{N}_a(\mathcal{S}) := \cup_{\mathbf{x}\in\mathcal{S}} \mathcal{N}_a(\mathbf{x})$.

Definition 2. A vector $\mathbf{g}(\mathbf{x}^*)$ dominates $\mathbf{g}(\mathbf{x})$, written as $\mathbf{g}(\mathbf{x}^*) \leq \mathbf{g}(\mathbf{x})$, if $g_k(\mathbf{x}^*) \leq g_k(\mathbf{x})$ for all $k \in \{1,\ldots,d\}$ and $g_{k^*}(\mathbf{x}^*) < g_{k^*}(\mathbf{x})$ for at least one $k^* \in \{1,\ldots,d\}$.

Definition 3 (Cooper et al. 2018). Given $a \in \mathbb{R}, a \geq 1$, a set $\mathcal{L}_a \subseteq \mathcal{X}, |\mathcal{L}_a| \geq 1$ is an $\mathcal{N}_a$-local efficient set ($\mathcal{N}_a$-LES) if (a) for each $\mathbf{x}^* \in \mathcal{L}_a$, $\nexists \mathbf{x} \in \mathcal{N}_a(\mathbf{x}^*) \cap \mathcal{X}$ such that $\mathbf{g}(\mathbf{x}) \leq \mathbf{g}(\mathbf{x}^*)$, (b) for each $\mathbf{x}^* \in \mathcal{L}_a, \nexists \mathbf{x}' \in \mathcal{L}_a$ such that $\mathbf{g}(\mathbf{x}') \leq \mathbf{g}(\mathbf{x}^*)$, (c) for each $\mathbf{x} \in (\mathcal{N}_a(\mathcal{L}_a) \setminus \mathcal{L}_a) \cap \mathcal{X}, \exists \mathbf{x}^* \in \mathcal{L}_a$ such that $\mathbf{g}(\mathbf{x}^*) \leq \mathbf{g}(\mathbf{x})$.

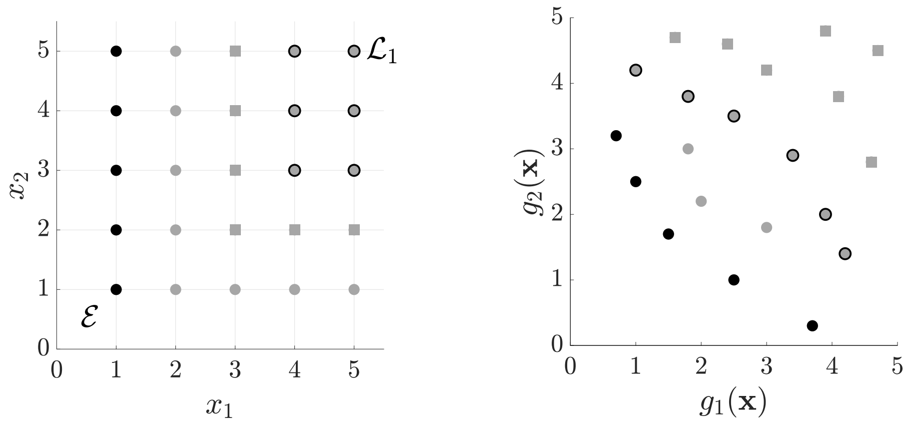

Figure 1 In this example, there are two LES's: the first, $\mathcal{L}_1$, is indicated by outlined points with its neighbors indicated by squares; the second, $\mathcal{E}$, is indicated by black points and is also the global efficient set.

Figure 1 shows two $\mathcal{N}_1$-LES's, $\mathcal{L}_1$ and $\mathcal{E}$. The set $\mathcal{E}$ is also the global efficient set, which we define as a local efficient set with $a=\infty$.

All algorithms currently implemented in PyMOSO solve Problem M_d using an algorithmic framework called RA. RA is a version of sample average approximation that is designed for algorithmic efficiency. Algorithms in an RA framework solve a sequence of sample-path problems at increasing sample sizes, using the solution from the previous RA iteration as a warm start for the current RA iteration. The sample-path problem is defined as

$$\text{minimize}_{\mathbf{x}\in\mathcal{X}} \left\{\bar{\mathbf{G}}_{m_\nu}(\mathbf{x}) = (\bar{G}_{1,m_\nu}(\mathbf{x}),\ldots,\bar{G}_{d,m_\nu}(\mathbf{x})) := \left(\tfrac{1}{m_\nu}\textstyle\sum_{i=1}^{m_\nu} G_1(\mathbf{x},\boldsymbol{\xi}_i),\ldots,\tfrac{1}{m_\nu}\sum_{i=1}^{m_\nu} G_d(\mathbf{x},\boldsymbol{\xi}_i)\right)\right\},$$

where $\bar{\mathbf{G}}_{m_\nu}(\mathbf{x})$ is an estimator of $\mathbf{g}(\mathbf{x})$ constucted in RA iteration ν with sample size m_ν. The local solution to the sample-path problem is a sample-path $\mathcal{N}_a$-LES, which we define by replacing all quantities in Definition 3 by their respective estimators. Since the estimated LES found in RA iteration $\nu-1$ is used as a warm start in RA iteration ν and the sequence of sample sizes $\{m_\nu, \nu=1,2,\ldots\}$ is increasing, RA algorithms are efficient because they reserve large sample sizes for points that are "close" to the solution.

In the context of R-PERLE and R-MinRLE, the algorithms that solve the sample-path problems, also called *sample-path solvers*, consist of two sub-routines: (1) an *accelerator* routine which generates a candidate estimated LES, and (2) RLE, which determines whether a set is an estimated LES and, if not, constructs an estimated LES by crawling through the feasible space. Thus, given a starting feasible point, the goal of the accelerator is to find candidate estimated LES members more efficiently than RLE alone would. As $\nu\to\infty$, the sequence of estimated LES's generated by RLE in an RA framework converge to a LES almost surely under appropriate regularity conditions. Loosely speaking, the regularity conditions include that all LES's are finite, the standard errors of the objective estimators go to zero fast enough as the sample size increases (e.g., the objective estimators are not heavy-tailed), and the objective values of all feasible points are separated from each other. (If this last condition does not hold, the algorithm guarantees a slightly weaker form of convergence.) Under the additional assumptions that the feasible set is finite, all local efficient points are global efficient points, and there is exactly one LES that equals the global efficient set, R-PERLE converges exponentially fast. Given a distance a and a total simulation budget that limits the number of simulation oracle calls, R-PERLE and R-MinRLE return an *approximate sample-path* $\mathcal{N}_a$*-LES* (ALES), which we refer to as an estimated LES. For an exact definition of an ALES, and for a thorough explanation of the regularity conditions required for each type of convergence, see Cooper et al. (2018).

2. Practitioners: Using PyMOSO to Solve a Problem

In this section, we discuss using PyMOSO in the practitioner context. Practitioners use PyMOSO in two steps: first, implement the simulation oracle in PyMOSO, which we discuss in §2.1, and second, use PyMOSO to solve the problem, which we discuss in §2.2.

2.1. Structuring an Oracle for Use in PyMOSO

To structure an oracle for use in PyMOSO, a practitioner should modify the Python source code template provided in Figure 2. Figure 2 implements an oracle named `MyProblem` in a Python file named `myproblem.py`. PyMOSO requires that the problem name match the file name, although the capitalization does not need to match.

A practitioner implements a MOSO problem by first modifying the number of objectives and the dimension of the feasible points in Lines 8 and 9 of Figure 2, respectively. Setting

```
1  # import the Oracle base class
2  from pymoso.chnbase import Oracle
3
4  class MyProblem(Oracle):
5      '''Example implementation of a user-defined MOSO problem.'''
6      def __init__(self, rng):
7          '''Specify the number of objectives and dimensionality of points.'''
8          self.num_obj = 2
9          self.dim = 1
10         super().__init__(rng)
11
12     def g(self, x, rng):
13         '''Check feasibility and simulate objective values.'''
14         #objective_values = (obj1, obj2), is_feasible = True
15         return is_feasible, objective_values
```

Figure 2 **The Python file `myproblem.py` is a template PyMOSO oracle. As shown, `g(self, x, rng)` is incomplete.**

the correct values for the practitioner's Problem M_d, and changing nothing else, is sufficient to implement the `__init__(self, rng)` method correctly. Then, the practitioner should replace the comment in Line 14 of Figure 2 with valid Python code that, given a Python tuple `x` representing a point $\mathbf{x} \in \mathbb{R}^q$, generates the following to return in Line 15: (a) a boolean indicator denoting whether $\mathbf{x}$ is feasible, and (b) a Python tuple, containing one observation of every objective function at $\mathbf{x}$ if $\mathbf{x}$ is feasible, and a Python tuple containing `None` for every objective if $\mathbf{x}$ is not feasible. In our notation, one observation of every objective function at $\mathbf{x}$ is represented by $(G_1(\mathbf{x}, \boldsymbol{\xi}), \ldots, G_d(\mathbf{x}, \boldsymbol{\xi}))$; alternatively, the practitioner may think of this quantity as one observation of $\bar{\mathbf{G}}_n(\mathbf{x})$ where $n = 1$. The function `g(self, x, rng)` may contain any number of lines and may be a wrapper for an external simulation oracle. Returning the feasibility indicator followed by a Python tuple of the objective values is sufficient to correctly implement the `g(self, x, rng)` method.

Optionally, a practitioner may use the PyMOSO object `rng` to generate pseudo-random numbers with `mrg32k3a`, or may use the `mrg32k3a` seed from `rng` as the seed in an external `mrg32k3a` generator. If `MyProblem` implements either approach, PyMOSO ensures that simulation replications obtained in parallel are independent by exploiting the stream and substream capabilities of `mrg32k3a` (L'Ecuyer et al. 2002). Further, the implemented PyMOSO oracle is compatible with PyMOSO's common random number (CRN) framework. Practitioners who ignore `rng` should take care when using PyMOSO's parallel computing and CRN capabilities. To determine when using CRN is appropriate, we refer the reader to Law (2015). More detailed information about `rng` is available in the PyMOSO user manual.

In the remainder of the paper, we assume the practitioner has implemented a PyMOSO oracle called `MyProblem`. So the reader can run our examples, we provide a simple example of `MyProblem` in Figure 3, where Problem M_2 is $g_1(x) = x^2 + z_0$ and $g_2(x) = (x-2)^2 + z_1$, $x \in \{-100, -99, ..., 100\}$, and z_0, z_1 are standard normal random variables. The solution is $\{0, 1, 2\}$. We remark here that Figure 3 contains an example of using `rng`.

2.2. Solving a MOSO Problem in PyMOSO

Having implemented a PyMOSO oracle called `MyProblem` in §2.1, we now discuss using PyMOSO to solve `MyProblem`. Practitioners may use PyMOSO in two modes: as a standalone solver invoked from the command line, or as a subroutine in a Python program. The former creates a file containing the output of one run of the selected algorithm, which is an estimated LES. The latter returns the estimated LES as a Python set. In either case,

```
def g(self, x, rng):
    '''Check feasibility and simulate objective values for MyProblem.'''
    feas_range = range(-100, 101)
    obj = []
    is_feas = False
    # check that dimensions of x match self.dim
    if len(x) == self.dim:
        is_feas = True
        for i in x:
            if not i in feas_range:
                is_feas = False
    if is_feas:
        z_vec = [rng.normalvariate(0, 1) for i in [0, 1]]
        obj1 = x[0]**2 + z_vec[0]
        obj2 = (x[0] - 2)**2 + z_vec[1]
    return is_feas, (obj1, obj2)
```

Figure 3 **This figure provides an example `g` function, which we use in `MyProblem`.**

PyMOSO requires the practitioner to specify, using a method we describe, at least the following information: the problem, the algorithm, and an initial feasible point. The method is slightly different depending on the chosen mode. In this section, we only consider the command line mode to solve `MyProblem`. See the user manual for the subroutine mode.

The simplest viable command to solve a problem in command line mode follows the structure `program command problem solver x0`, where `x0` denotes the initial feasible point. The solve command takes options, including the total simulation budget and the number of parallel processors. Practitioners may view the full set of available options by entering `pymoso --help` and the full list of PyMOSO solver names by entering `pymoso listitems`. As an example, the command to solve `MyProblem` using R-PERLE, starting from the feasible point 97, with a total simulation budget of 10,000 and 4 processors, is below.

```
pymoso solve --budget=10000 --simpar=4 myproblem.py RPERLE 97
```

This invocation requires that `myproblem.py` is in the working directory. Since the feasible points of `MyProblem` are one-dimensional, `x0` is a scalar. For feasible points in higher dimensions, separate each component with a space, e.g., a three-dimensional point $(97, 23, 18)$ is written as `97 23 18`. After issuing the above command, PyMOSO creates a new subdirectory named `testrun` in the working directory. This subdirectory typically contains two files: one file containing the metadata and one file containing the estimated LES. We exhibit the file containing the estimated LES in Figure 4. If PyMOSO detects an error, it may also write an error file.

```
(2,)
(0,)
(1,)
```

Figure 4 **The output of the `solve` invocation given in §2.2 gives the correct answer to `MyProblem` as a list of points. The unusual syntax is because the feasible points have only one dimension and are each represented as a Python tuple of length one.**

3. Researchers: Testing and Comparing MOSO Algorithms with PyMOSO

In this section, we discuss using PyMOSO in the researcher context. Researchers can use PyMOSO to compare algorithms and to create new algorithms. To compare algorithms, researchers first implement a PyMOSO oracle as in §2.1. Then, they create a PyMOSO tester, which we discuss in §3.1, and run the tester, which we discuss in §3.2. We briefly discuss creating new algorithms in §3.3.

3.1. Structuring a Test Problem for Use in PyMOSO

After implementing a PyMOSO oracle called `MyProblem` in §2.1, researchers must implement a PyMOSO tester for `MyProblem`. We provide an example tester called `MyTester` in Figure 5, where the oracle to be tested in Line 27 is specified as `MyProblem`.

Technically, a valid PyMOSO tester may consist of only Lines 24–27 in Figure 5. However, Figure 5 illustrates two optional features that researchers may find useful. First, researchers can implement a PyMOSO function that generates feasible starting points by setting `self.get_ranx0` to an appropriate function in Line 30. We provide an example function, called `get_ranx0`, that randomly generates a feasible point for `MyProblem` in Lines 9–12 of Figure 5. The function must take `rng` as a parameter and return a Python tuple representing a feasible point. The second feature enables researchers to implement a metric for comparing an estimated solution to the known, true solution.

We provide an example metric that calculates the Hausdorff distance from the expected objective values of the points in the estimated LES, `eles`, to the image of the known solution, `self.answer`, in Lines 32–36. To calculate the expected objective values of the points in `eles`, we implement the function `true_g` in Lines 15-19. We also specify `myanswer` in Line 22 and set `self.answer` and `self.true_g` as members of `MyTester` in Lines 28–29. Researchers may replace Lines 34–36 with a metric of their choosing.

3.2. Testing a MOSO Algorithm in PyMOSO

Having implemented both `MyProblem`, in §2.1, and its tester, `MyTester` in §3.1, in PyMOSO, we now discuss using PyMOSO to test algorithms on `MyProblem`. As with practitioners

```
import sys, os
sys.path.insert(0,os.path.dirname(__file__))
# use hausdorff distance (dh) as an example metric
from pymoso.chnutils import dh
# import the MyProblem oracle
from myproblem import MyProblem

# optionally, define a function to randomly choose a MyProblem feasible x0
def get_ranx0(rng):
    val = rng.choice(range(-100, 101))
    x0 = (val, )
    return x0

# compute the true values of x, for computing the metric
def true_g(x):
    '''Compute the objective values.'''
    obj1 = x[0]**2
    obj2 = (x[0] - 2)**2
    return obj1, obj2

# define an answer as appropriate for the metric
myanswer = {(0, 4), (4, 0), (1, 1)}

class MyTester(object):
    '''Example tester implementation for MyProblem.'''
    def __init__(self):
        self.ranorc = MyProblem
        self.answer = myanswer
        self.true_g = true_g
        self.get_ranx0 = get_ranx0

    def metric(self, eles):
        '''Metric to be computed per retrospective iteration.'''
        epareto = [self.true_g(point) for point in eles]
        haus = dh(epareto, self.answer)
        return haus
```

Figure 5 **The file `mytester.py` implements `MyTester`, a tester for `MyProblem`.**

solving MOSO problems, researchers can use PyMOSO in two modes for testing algorithms on problems: as a stand-alone solver invoked from the command line, or as a subroutine in a Python program. In this section, we only consider the command line mode to test PyMOSO algorithms. See the user manual for the subroutine mode.

The simplest viable command to test an algorithm in command line mode follows the structure `program command tester solver`. (Researchers may also specify a feasible starting point if the tester is not programmed to generate them.) The `testsolve` command takes options, including the number of independent sample paths of the test problem, the number of processors to use, and whether to compute a metric. As an example, the command to test R-PERLE by running 16 independent sample paths of `MyProblem` using `MyTester`, on 4 processors and computing a metric, is below.

```
pymoso --isp=16 --proc=4 --metric testsolve mytester.py RPERLE
```

This invocation requires that `myproblem.py` and `mytester.py` are in the working directory. After issuing the above command, PyMOSO creates a new subdirectory named `testrun` in the working directory. This subdirectory contains (a) one metadata file; (b) 16 data files, each containing a list of estimated LES's, one for every algorithm iteration; and (c) 16 files containing metric calculations, which are included only when using the `--metric` option. The files containing metric calculations each have data of the form $(\nu,\, w_\nu,\, h_\nu)$, where ν is the RA algorithm iteration number; w_ν is the cumulative work done, measured as the total number of simulations used at the end of iteration ν; and h_ν is the metric computed on the estimated LES at iteration ν. We exhibit sample data and metric files in Figures 6 and 7. If PyMOSO detects an error, it may also write an error file.

```
1  {(3,)}
2  {(2,), (1,)}
3  {(0,), (1,)}
4  {(2,), (0,), (1,)}
5  {(2,), (0,), (1,)}
6  {(2,), (1,)}
7  {(0,), (1,)}
8  {(2,), (0,), (1,)}
9  {(2,), (0,), (1,)}
10 {(2,), (0,), (1,)}
```

Figure 6 **One of the 16 solution output files from the `testsolve` invocation given in §3.2. Each line contains a set, which is the solution of the iteration corresponding to the line number.**

```
(0, 0, 9.486832980505138)
(1, 15, 3.1622776601683795)
(2, 33, 3.1622776601683795)
(3, 51, 0.0)
(4, 69, 0.0)
(5, 93, 3.1622776601683795)
(6, 117, 3.1622776601683795)
(7, 141, 0.0)
(8, 171, 0.0)
(9, 201, 0.0)
```

Figure 7 **One of 16 metric output files from the `testsolve` invocation given in §3.2. Each line contains indicates the iteration number, the number of simulations used at the end of the iteration, and the performance metric of the iteration solution. In this example, the best possible performance is zero.**

```
1  from pymoso.chnbase import RLESolver
2
3  # create a subclass of RLESolver
4  class MyAccel(RLESolver):
5      '''Example implementation of an RLE accelerator.'''
6
7      def accel(self, warm_start):
8          '''Return a collection of points to send to RLE.'''
9          # implement algorithm logic here and return a set
10         return warm_start
```

Figure 8 **The file `myaccel.py` implements a provably convergent MOSO algorithm by relying on RLE in a RA framework. We encourage MOSO researchers to improve it.**

3.3. Creating New Accelerators in PyMOSO

Though researchers may implement any simulation optimization algorithm in PyMOSO, we discuss how to implement RA algorithms that invoke an "accelerator" followed by RLE in every RA iteration (see Cooper et al. 2018). For example, in R-PERLE, "Pε" is the accelerator, and in R-MinRLE, "Min" is the accelerator. Users can create new accelerators. We provide an accelerator template in Figure 8. Researchers should replace the comment in Line 9 of Figure 8 with their own code. The function signature must be `accel(self, warm_start)` and the function must return a Python set. After implementing a PyMOSO algorithm, researchers can test it as in §3.2.

```
pymoso --isp=20 --proc=4 --metric testsolve mytester.py myaccel.py
```

For implementing algorithm logic, PyMOSO also provides support for, e.g., obtaining simulation replications from the oracle, computing all points in a set that are non-dominated, and generating neighborhoods of points and sets. For detailed descriptions of the support functions, including working examples, we refer the reader to the user manual.

4. Conclusion

The PyMOSO software package provides open-source, off-the-shelf access to state-of-the-art solvers for simulation optimization on integer lattices in an accessible and popular programming language. PyMOSO also provides a framework and useful tools for researchers who wish to compare and create new algorithms.

Acknowledgments

S. R. Hunter was supported by the National Science Foundation through grant CMMI-1554144. We thank Kalyani Nagaraj for her work on R-PERLE and Devadatta Kulkarni for reviewing draft versions of this paper.

References

Andersson M, Grimm H, Persson A, Ng A (2007) A web-based simulation optimization system for industrial scheduling. Henderson SG, Biller B, Hsieh MH, Shortle J, Tew JD, Barton RR, eds., *Proceedings of the 2007 Winter Simulation Conference*, 1844–1852 (Piscataway, NJ: IEEE).

Biscani F, Izzo D (2018) esa/pagmo2: pagmo 2.9. URL `http://dx.doi.org/10.5281/zenodo.1406840`.

Chen T, Wang C (2016) Multi-objective simulation optimization for medical capacity allocation in emergency department. *Journal of Simulation* 10(1):50–68, URL `http://dx.doi.org/10.1057/jos.2014.39`.

Cooper K, Hunter SR, Nagaraj K (2018) Bi-objective simulation optimization on integer lattices using the epsilon-constraint method in a retrospective approximation framework. *Optimization Online* URL `http://www.optimization-online.org/DB_HTML/2018/06/6649.html`.

Hong LJ, Nelson BL, Xu J (2015) Discrete optimization via simulation. Fu MC, ed., *Handbook of Simulation Optimization*, volume 216 of *International Series in Operations Research & Management Science*, 9–44 (New York: Springer), URL `http://dx.doi.org/10.1007/978-1-4939-1384-8_1`.

Hunter SR, Applegate EA, Arora V, Chong B, Cooper K, Rincón-Guevara O, Vivas-Valencia C (2019) An introduction to multi-objective simulation optimization. *ACM Transactions on Modeling and Computer Simulation* 29(1):7:1–7:36, URL `http://dx.doi.org/10.1145/3299872`.

Law AM (2015) *Simulation Modeling and Analysis* (New York: McGraw Hill Education), 5 edition.

L'Ecuyer P (1999) Good parameters and implementations for combined multiple recursive random number generators. *Operations Research* 47(1):159 – 164, URL `http://dx.doi.org/https://doi.org/10.1287/opre.47.1.159`.

L'Ecuyer P, Simard R, Chen EJ, Kelton WD (2002) An object-oriented random-number package with many long streams and substreams. *Operations Research* 50(6):1073–1075, URL `http://dx.doi.org/10.1287/opre.50.6.1073.358`.

Lee LH, Chew EP, Teng S, Chen Y (2008) Multi-objective simulation-based evolutionary algorithm for an aircraft spare parts allocation problem. *European Journal of Operational Research* 189(2):476–491.

Li H, Zhu Y, Chen Y, Pedrielli G, Pujowidianto NA, Chen Y (2015) The object-oriented discrete event simulation modeling: a case study on aircraft spare part management. Yilmaz L, Chan WKV, Roeder TMK, Macal C, Rosetti M, eds., *Proceedings of the 2015 Winter Simulation Conference*, 3514–3525 (Piscataway, NJ: IEEE).

OptTek Systems, Inc (2018) OptQuest. URL `http://www.opttek.com/products/optquest/`.

Pasupathy R, Ghosh S (2013) Simulation optimization: a concise overview and implementation guide. Topaloglu H, ed., *TutORials in Operations Research*, chapter 7, 122–150 (Catonsville, MD: INFORMS), URL `http://dx.doi.org/10.1287/educ.2013.0118`.

Schmeiser B (2008) A practitioner, a vender, and a researcher walk into a bar: trying to explain what researchers do. Mason SJ, Hill RR, Mönch L, Rose O, Jefferson T, Fowler JW, eds., *Proceedings of* *the 2008 Winter Simulation Conference*, 2–9 (Piscataway, NJ: IEEE), URL http://dx.doi.org/10.1109/WSC.2008.4736049.

Swain J (2017) Simulation software survey - simulation: new and improved reality show. *OR/MS Today* 44(5):38–49.

Thengvall B, Glover F, Davino D (2016) Coupling optimization and statistical analysis with simulation models. Roeder TMK, Frazier PI, Szechtman R, Zhou E, Huschka T, Chick SE, eds., *Proceedings of the 2016 Winter Simulation Conference* (Piscataway, NJ: IEEE), URL http://dx.doi.org/10.1109/WSC.2016.7822120.

Wang H, Pasupathy R, Schmeiser BW (2013) Integer-ordered simulation optimization using R-SPLINE: Retrospective Search using Piecewise-Linear Interpolation and Neighborhood Enumeration. *ACM Transactions on Modeling and Computer Simulation* 23(3), URL http://dx.doi.org/10.1145/2499913.2499916.

Weizhi L (2017) PyPRS. GitHub repository, URL https://github.com/Greenwicher/PyPRS.

Zhou C, Li H, Lee BK, Qiu Z (2018) A simulation-based vessel-truck coordination strategy for lighterage terminals. *Transportation Research Part C: Emerging Technologies* 95:149–164, URL http://dx.doi.org/10.1016/j.trc.2018.07.015.

Online Supplement for PyMOSO: Software for Multi-Objective Simulation Optimization with R-PERLE and R-MinRLE

Kyle Cooper
School of Industrial Engineering, Purdue University and Tata Consultancy Services, coope149@purdue.edu

Susan R. Hunter
School of Industrial Engineering, Purdue University, susanhunter@purdue.edu

---

Contents

A. PyMOSO User Manual

We provide this user manual to accompany the initial release of PyMOSO.

A.1. Additional Reading

The initial release of PyMOSO contains solvers that implement four total algorithms, in alphabetical order: R-MinRLE, R-PE, R-PERLE, and R-SPLINE. The algorithms R-MinRLE, R-PE, and R-PERLE were introduced in the following paper:

Cooper, K., Hunter, S. R., and Nagaraj, K. 2018. Bi-objective simulation optimization on integer lattices using the epsilon-constraint method in a retrospective approximation framework. Optimization Online, `http://www.optimization-online.org/DB_HTML/2018/06/6649.html`.

The algorithm R-SPLINE was introduced in the following paper:

Wang, H., Pasupathy, R., and Schmeiser, B. W. 2013. Integer-ordered simulation optimization using R-SPLINE: Retrospective Search with Piecewise-Linear Interpolation and Neighborhood Enumeration. ACM Transactions on Modeling and Computer Simulation, Vol. 23, No. 3, Article 17 (July 2013), 24 pages. `http://dx.doi.org/10.1145/2499913.2499916`

We recommend reading these papers to understand the algorithms, what they return, and the algorithm parameter options that we describe in the user manual.

A.2. Installation

Since PyMOSO is programmed in Python, every PyMOSO user must first install Python, which can be downloaded from `https://www.python.org/downloads/`. PyMOSO is compatible with Python versions 3.6 and higher. In the remainder of this section, we assume an appropriate Python version is installed. We discuss three different methods to install PyMOSO: first, from the Python Packaging Index; second, directly from our source code using git; and third, manually installing PyMOSO from our source code.

A.2.1. Install PyMOSO from the Python Packaging Index using `pip` For ease of distribution, we keep stable, recent releases of PyMOSO on the Python Packaging Index (PyPI). Since the program `pip` is included in Python versions 3.6 and higher, we recommend using `pip` to install PyMOSO. To do so, open a terminal, type the following command, and press enter.

```
pip install pymoso
```

Depending on how users configure their Python installation and how many versions of Python they install, they may need to replace `pip` with `pip3`, or other variants of `pip`.

A.2.2. Install PyMOSO from `git` using `pip` Users with `git` installed can use `pip` to install the most current version of PyMOSO directly from our source code:

```
pip install git+https://github.com/pymoso/PyMOSO.git
```

We consider the latest source to be less stable than the fixed releases we upload to PyPI, and thus we recommend most users install PyMOSO as in §A.2.1.

A.2.3. Install PyMOSO Manually from Source Code Users may follow the steps below to manually install PyMOSO from any version of the source code.

1. Acquire the PyMOSO source code, for example, by downloading it from the repository `https://github.com/pymoso/PyMOSO`.
2. Install the `wheel` package, e.g. using the `pip install wheel` command.
3. Open a terminal and navigate into the main project directory which contains the file `setup.py`
4. Build the installable PyMOSO package, called a wheel, using the command `python setup.py bdist_wheel`. As with `pip`, some users may need to replace `python` with `python3` or something similar. The command should create a directory named `dist` containing the PyMOSO wheel.
5. Install the PyMOSO wheel using `pip install dist/pymoso-x.x.x-py3-none-any.whl`, where users replace `x.x.x` with the appropriate PyMOSO version.

A.3. Command Line Interface (CLI)

PyMOSO users solving MOSO problems and testing MOSO algorithms may do so using the command line interface. First, we show how to access the included help file. Then, we show how to view the lists of solvers, testers, and oracles installed by default with PyMOSO. Finally, we discuss the `solve` and `testsolve` commands.

A.3.1. CLI Help PyMOSO includes a command line help file. The help file shows syntax templates for every PyMOSO command, the available options, and a selection of example invocations. The `pymoso --help` invocation prints the file to the terminal. The file is also printed when PyMOSO cannot parse an invocation that begins with `pymoso`. We show the current help file in Figure 9.

A.3.2. The `listitems` Command for Viewing Solvers, Testers, and Oracles Included in PyMOSO The default installation of PyMOSO includes a selection of solvers, testers, and oracles. Users can view the complete lists of included solvers, testers, and oracles using

```
Usage:
  pymoso listitems
  pymoso solve [--budget=B] [--odir=D] [--crn] [--simpar=P]
    [(--seed <s> <s> <s> <s> <s> <s>)] [(--param <param> <val>)]...
    <problem> <solver> <x>...
  pymoso testsolve [--budget=B] [--odir=D] [--crn] [--isp=T] [--proc=Q]
    [--metric] [(--seed <s> <s> <s> <s> <s> <s>)] [(--param <param> <val>)]...
    <tester> <solver> [<x>...]
  pymoso -h | --help
  pymoso -v | --version

Options:
  --budget=B      Set the simulation budget [default: 200]
  --odir=D        Set the output file directory name. [default: testrun]
  --crn           Set if common random numbers are desired.
  --simpar=P      Set number of parallel processes for simulation replications. [default: 1]
  --isp=T         Set number of algorithm instances to solve. [default: 1]
  --proc=Q        Set number of parallel processes for the algorithm instances. [default: 1]
  --metric        Set if metric computation is desired.
  --seed          Set the random number seed with 6 spaced integers.
  --param         Specify a solver-specific parameter <param> <val>.
  -h --help       Show this screen.
  -v --version    Show version.

Examples:
  pymoso listitems
  pymoso solve ProbTPA RPERLE 4 14
  pymoso solve --budget=100000 --odir=test1  ProbTPB RMINRLE 3 12
  pymoso solve --seed 12345 32123 5322 2 9543 666666666 ProbTPC RPERLE 31 21 11
  pymoso solve --simpar=4 --param betaeps 0.4 ProbTPA RPERLE 30 30
  pymoso solve --param radius 3 ProbTPA RPERLE 45 45
  pymoso testsolve --isp=16 --proc=4 TPATester RPERLE
  pymoso testsolve --isp=20 --proc=10 --metric --crn TPBTester RMINRLE 9 9
```

Figure 9 **PyMOSO displays help when users enter the `pymoso --help` invocation.**

the `pymoso listitems` command. We show the current listing in Figure 10. Test problems A, B, and C refer to those in Cooper et al. (2018).

A.3.3. The `solve` Command The PyMOSO `solve` command is for solving MOSO problems. Users can solve the built-in problems (use the `listitems` command to view the built-in problems), however, PyMOSO `solve` users typically will have their own MOSO problem they wish to solve. Thus, we assume users have implemented a PyMOSO oracle named `MyProblem`

```
Solver                     Description
***********************    ***********************
RMINRLE                    A solver using R-MinRLE for integer-ordered MOSO.
RPE                        A solver using R-Pe for integer-ordered bi-objective MOSO.
RPERLE                     A solver using R-PERLE for integer-ordered bi-objective MOSO.
RSPLINE                    A solver using R-SPLINE for single objective SO.

Problems                   Description                Test Name (if available)
***********************    ***********************    ***********************
ProbSimpleSO               x^2 + noise.               SimpleSOTester
ProbTPA                    Test Problem A             TPATester
ProbTPB                    Test Problem B             TPBTester
ProbTPC                    Test Problem C             TPCTester
BSProb                     Bus Scheduling problem     BSTester
```

Figure 10 **The `pymoso listitems` invocation shows the lists of built-in solvers, testers, and oracles.**

in `myproblem.py`. In the examples that follow, we assume the `MyProblem` implementation in Figure 11, which is a bi-objective oracle with one-dimensional feasible points. See §A.4.1 for instructions on implementing a MOSO problem as a PyMOSO oracle.

The template `solve` command is `pymoso solve oracle solver x0`, where `oracle` is a built-in or user-defined oracle, `solver` is a built-in or user-defined algorithm, and `x0` is a feasible starting point for the solver, with a space between each component. As a first example, we solve the user-defined `MyProblem` using the built-in R-PERLE starting at the feasible point 97.

```
pymoso solve myproblem.py RPERLE 97
```

Similarly, we can solve built-in problems, such as `ProbTPA` which has two-dimensional feasible points.

```
pymoso solve ProbTPA RPERLE 40 40
```

```python
# import the Oracle base class
from pymoso.chnbase import Oracle

class MyProblem(Oracle):
    '''Example implementation of a user-defined MOSO problem.'''
    def __init__(self, rng):
        '''Specify the number of objectives and dimensionality of points.'''
        self.num_obj = 2
        self.dim = 1
        super().__init__(rng)

    def g(self, x, rng):
        '''Check feasibility and simulate objective values.'''
        # feasible values for x in this example
        feas_range = range(-100, 101)
        # initialize obj to empty and is_feas to False
        obj = []
        is_feas = False
        # check that dimensions of x match self.dim
        if len(x) == self.dim:
            is_feas = True
            # then check that each component of x is in the range above
            for i in x:
                if not i in feas_range:
                    is_feas = False
        # if x is feasible, simulate the objectives
        if is_feas:
            #use rng to generate random numbers
            z0 = rng.normalvariate(0, 1)
            z1 = rng.normalvariate(0, 1)
            obj1 = x[0]**2 + z0
            obj2 = (x[0] - 2)**2 + z1
            obj = (obj1, obj2)
        return is_feas, obj
```

Figure 11 **The file `myproblem.py` implements the example `MyProblem`.**

Henceforth, we present `solve` examples only for solving `MyProblem`. Since `MyProblem` is bi-objective, we recommend using the R-PERLE solver. However, for two or more objectives, PyMOSO has R-MinRLE.

```
pymoso solve myproblem.py RMINRLE 97
```

For a single objective problem, PyMOSO has R-SPLINE. We remark that if given a multi-objective problem, R-SPLINE will simply minimize the first objective. We do not necessarily prohibit such use, but urge that users take care when using R-SPLINE to minimize one objective of a many-objective problem.

```
pymoso solve myproblem.py RSPLNE 97
```

Regardless of the chosen solver, PyMOSO creates a new sub-directory of the working directory containing output. There will be a metadata file, indicating the date, time, solver, problem, and any other specified options. In addition, PyMOSO creates a file containing the solver-generated solution. PyMOSO provides additional options for users solving MOSO problems. We present examples of each option below. First, users can specify the name of the output directory.

```
pymoso solve --odir=OutDirectory myproblem.py RPERLE 45
```

Users can specify the simulation budget, which is currently set to a default of 200.

```
pymoso solve --budget=100000 myproblem.py RPERLE 12
```

Users may specify to take simulation replications in parallel. We only recommend doing so if the user has thought through appropriate pseudo-random number stream control issues (see §A.4.1). Furthermore, due to the overhead of parallelization, we only recommend using the parallel simulation replications feature if observations are sufficiently "expensive" to compute, e.g. the simulation takes a half second or more to generate a single observation. We remark that the run-time complexity of the simulation oracle may not perfectly indicate when it is appropriate to use parallelization; other factors include, e.g., the total simulation budget.

```
pymoso solve --simpar=4 myproblem.py RPERLE 44
```

Currently, all PyMOSO solvers support using common random numbers. Users may enable the functionality using the `crn` option.

```
pymoso solve --crn myproblem.py RMINRLE 62
```

We do not recommend this option unless the oracle is implemented to be compatible, that is, the oracle uses PyMOSO's pseudo-random number generator to generate pseudo-random numbers or to provide a seed to an external `mrg32k3a` generator (see §A.4.1).

Users may specify an initial seed to PyMOSO's `mrg32k3a` pseudo-random number generator. Seeds must be 6 positive integers with spaces. The default is 12345 for each of the 6 components.

```
pymoso solve --seed 1111 2222 3333 4444 5555 6666 myproblem.py RPERLE 23
```

Users may specify algorithm-specific parameters (see the papers in which the algorithms were introduced for detailed explanations of the parameters). All parameters are specified in the form `--param name value`. For example, the RLE relaxation parameter can be specified and set as `betadel` to a real number. We refer the reader to Table 1 for the full list of currently available algorithm-specific parameters.

```
pymoso solve --param betadel 0.2 myproblem.py RPERLE 34
```

Table 1 The table contains the current list of algorithm-specific parameters.

Parameter Name	Default Value	Affected Solvers	Description
mconst	2	R-PERLE, R-MinRLE, R-PE, R-SPLINE	Initialize the sample size and subsequent schedule of sample sizes.
bconst	8	R-PERLE, R-MinRLE, R-PE, R-SPLINE	Initialize the search sampling limit and subsequent schedule of limits.
radius	1	R-PERLE, R-MinRLE, R-PE, R-SPLINE	Set the radius a that determines a point's neighborhood, $\mathcal{N}_a$ (Wang et al. 2013).
betadel	0.5	R-PERLE, R-MinRLE	An error tolerance parameter for RLE. See Cooper et al. (2018).
betaeps	0.5	R-PERLE, R-PE	An error tolerance parameter for Pε. See Cooper et al. (2018).

Finally, users may specify any number of options in one invocation. However, all options must be specified after the `solve` command and before the `myproblem.py` argument. Furthermore, any `--param` options must be last. (Note that the \ at the end of the first line continues the command to the second line.)

```
pymoso solve --crn --simpar=4 --budget=10000 --seed 1 2 3 4 5 6 \
  --odir=Exp1 --param mconst 4 --param betadel 0.7 myproblem.py RPERLE 97
```

A.3.4. The `testsolve` Command The PyMOSO `testsolve` command tests algorithms on problems using a PyMOSO tester. Users can test built-in or user-defined solvers with built-in or user-defined testers. In the examples that follow, we assume users have implemented `MyProblem` as in Figure 11 and the corresponding tester named `MyTester` in `mytester.py`, shown in Figure 12. See §A.4.2 for instructions on implementing a user-defined tester, including a metric for comparing algorithms, in PyMOSO.

```python
import sys, os
sys.path.insert(0,os.path.dirname(__file__))
# use hausdorff distance (dh) as an example metric
from pymoso.chnutils import dh
# import the MyProblem oracle
from myproblem import MyProblem

# optionally, define a function to randomly choose a MyProblem feasible x0
def get_ranx0(rng):
    val = rng.choice(range(-100, 101))
    x0 = (val, )
    return x0

# compute the true values of x, for computing the metric
def true_g(x):
    '''Compute the objective values.'''
    obj1 = x[0]**2
    obj2 = (x[0] - 2)**2
    return obj1, obj2

# define an answer as appropriate for the metric
myanswer = {(0, 4), (4, 0), (1, 1)}

class MyTester(object):
    '''Example tester implementation for MyProblem.'''
    def __init__(self):
        self.ranorc = MyProblem
        self.answer = myanswer
        self.true_g = true_g
        self.get_ranx0 = get_ranx0

    def metric(self, eles):
        '''Metric to be computed per retrospective iteration.'''
        epareto = [self.true_g(point) for point in eles]
        haus = dh(epareto, self.answer)
        return haus
```

Figure 12 **The file `mytester.py` implements the example `MyTester`.**

The template `testsolve` command is `pymoso testsolve tester solver` where `tester` is a built-in or user-defined tester, and `solver` is a built-in or user-defined solver. Users may also specify an `x0`, as in the `solve` command, if the `tester` does not implement the function to generate feasible points. As a first example, we test R-PERLE on `MyProblem` using `MyTester`. Since some options are compatible with both `solve` and `testsolve`, we include those options in this example.

```
pymoso testsolve --budget=999 --odir=exp1 \
  --crn --seed 1 2 3 4 5 6 mytester.py RPERLE
```

Users may want to compute some metric on the algorithm-generated solutions. If a metric is defined as part of the tester, such as in `MyTester`, the `testsolve` command can compute the metric on every algorithm iteration using the `--metric` option.

```
pymoso testsolve --metric mytester.py RPERLE
```

The `testsolve` command cannot perform simulation replications in parallel. However, testers can apply the solvers to independent sample paths of the problems. For example, to test R-PERLE on 100 independent sample paths of `MyProblem`, compute the metrics for each sample path, and use common random numbers in each sample path, use the following command.

```
pymoso testsolve --crn --metric --isp=100 mytester.py RPERLE
```

PyMOSO can perform independent algorithm runs in parallel. Use the `proc` option to specify the number of processes available to PyMOSO.

```
pymoso testsolve --crn --metric --isp=100 --proc=20 mytester.py RPERLE
```

We remark here that, to ensure the algorithm runs remain independent using PyMOSO's pseudo-random number generator (see §A.4.1), researchers should set the total simulation budget so that the included algorithms do not surpass 200 retrospective approximation (RA) iterations. For reference, using the default settings, the sample size at every point in the 200th RA iteration is almost 380 million.

The `testsolve` command creates a results file for each independent sample path. The file contains the solutions generated at every algorithm iteration, such that the solution of iteration 2 is on line 2, iteration 10 on line 10, and so forth. If `--metric` is specified, PyMOSO generates a second file for each independent sample path containing the collection of triples (iteration number, simulations used at end of iteration, metric).

A.4. Implementing Oracles, Testers, and Solvers in PyMOSO

To use PyMOSO, users solving MOSO problems must implement a PyMOSO oracle, and users testing MOSO algorithms should implement, at least, a PyMOSO oracle and tester. In this section, we provide template Python code to help users quickly implement oracles, testers, and perhaps solvers in PyMOSO.

A.4.1. Implementing PyMOSO Oracles Usually, implementing a PyMOSO oracle implies implementing a Monte Carlo simulation oracle as a black box function while following the PyMOSO rules put forth in this section. For reference, we discuss the example PyMOSO oracle `MyProblem` in Figure 11. Users may copy the code in Figure 11 and re-implement the function `g` as needed. We now list the basic requirements of every `g` implementation.

1. The function `g` must be an instance method of an `Oracle` sub-class, and thus take `self` as its first parameter.

2. The function `g` must take an arbitrarily-named second parameter which is a tuple of length `self.dim` and represents a point. Stylistically, PyMOSO consistently names this parameter `x`.
3. The function `g` must take an arbitrarily-named third parameter which is a modified Python `random.Random` object. Stylistically, PyMOSO consistently names this parameter `rng`.
4. The function `g` must return a boolean first and a tuple of length `self.num_obj` second.
   - The boolean is `True` if `x` is feasible, and `False` otherwise.
   - If `x` is feasible, the tuple contains a single observation of every objective. If `x` is not feasible, each element in the tuple is `None`.

If users already have an implemented simulation oracle, they may find it convenient to implement `g` as wrapper which calls that simulation from Python. As an example, suppose a user has implemented a simulation in C which is compiled to a C library called `mysim.so` and placed in the working directory. Suppose further that the simulation function takes the following as parameters: an array of integers representing a point $\mathtt{x} \in \mathbb{R}$ and an unsigned integer representing the number of observations to take at `x`. The function output is defined as `struct Simout` with members `feas` set to 0 or 1, `obj` a double array set to the mean of the observed objective values, and `var` a double array set to the sample variance of the observed objective values. Then users can modify the template to wrap the C function `struct Simout c_func(int x, int n)` as in Figure 13.

Figure 13 is a valid PyMOSO oracle which wraps a C function. However, PyMOSO algorithms cannot enable common random numbers on this oracle. Furthermore, PyMOSO cannot guarantee that observations are independent when taken in parallel. To enable these properties, the external simulation must use `mrg32k3a` as the generator and must accept a user-specified seed.

Suppose the library `mysim.so` also implements the function `set_simseed` which accepts a long array representing an `mrg32k3a` seed. We modify the wrapper in Figure 14 for compatibility with common random numbers and to guarantee independence of parallel observations. Figure 14 demonstrates using `rng.get_seed()` to return the current `mrg32k3a` seed.

Alternatively, if the number of required pseudo-random numbers is known, users can use `rng.random()` to generate pseudo-random numbers and then pass them to an external simulation if such functionality is supported.

```
from ctypes import CDLL, c_double, c_uint, c_int, Structure
import os.path
libname = 'mysim.so'
libabspath = os.path.dirname(os.path.abspath(__file__)) + os.path.sep + dll_name
libobj = CDLL(libabspath)

class Simout(Structure):
    _fields_ = [("feas", c_int), ("obj", c_double*2), ("var", c_double*2)]
csimout = libobj.c_func
csimout.restype = Simout

from pymoso.chnbase import Oracle

class MyProblem(Oracle):
    '''Example implementation of a user-defined MOSO problem.'''
    def __init__(self, rng):
        '''Specify the number of objectives and dimensionality of points.'''
        self.num_obj = 2
        self.dim = 1
        super().__init__(rng)

    def g(self, x, rng):
        '''Check feasibility and simulate objective values.'''
        is_feasible = True
        objective_values = (None, None)
        # g takes only one observation so set the c_func parameter to 1
        c_n = c_uint(1)
        # c_func requires is an integer so convert it -- this is a 1D example
        c_x = c_int(x[0])
        # call the C function
        mysimout = csimout(c_x, c_n)
        if not mysimout.feas:
            is_feasible = False
        else:
            is_feasible = True
        if is_feasible:
            objective_values = tuple(mysimout.obj)
        return is_feasible, objective_values
```

Figure 13 **The g function wraps an external simulation written in C.**

The `rng` object is implemented as a sub-class of Python's `random.Random` class, thus the official Python documentation for `random` applies to `rng` and is found at `https://docs.python.org/3/library/random.html`. In addition to `rng` using `mrg32k3a` as its generator, we also implement `rng.normalvariate` such that it uses the Beasley-Springer-Moro algorithm (Law 2015, p. 458) to approximate the inverse of the standard normal cumulative distribution function.

When using `rng`, to ensure independent sampling of observations, PyMOSO "jumps" forward in the pseudo-random number stream after obtaining every simulation replication. Each jump is of fixed size 2^{76} pseudo-random numbers. Thus, we require that every simulation replication use fewer than 2^{76} pseudo-random numbers. We ensure independence among parallel replications by "giving" each processor a stream (an `rng`), each of which is 2^{127} pseudo-random numbers apart. When using the current PyMOSO algorithms that rely on

```python
from ctypes import CDLL, c_double, c_uint, c_int, Structure, c_long
import os.path
libname = 'mysim.so'
libabspath = os.path.dirname(os.path.abspath(__file__)) + os.path.sep + dll_name
libobj = CDLL(libabspath)

class Simout(Structure):
    _fields_ = [("feas", c_int), ("obj", c_double*2), ("var", c_double*2)]
csimout = libobj.c_func
csetseed = libobj.set_simseed
csimout.restype = Simout

from pymoso.chnbase import Oracle

class MyProblem(Oracle):
    '''Example implementation of a user-defined MOSO problem.'''
    def __init__(self, rng):
        '''Specify the number of objectives and dimensionality of points.'''
        self.num_obj = 2
        self.dim = 1
        super().__init__(rng)

    def g(self, x, rng):
        '''Check feasibility and simulate objective values.'''
        is_feasible = True
        objective_values = (None, None)
        # get the PyMOSO seed from rng
        seed = rng.get_seed()
        # convert the seed to c_long array
        c_longarr = c_long*6
        c_seed = c_longarr(seed[0], seed[1], seed[2], seed[3], seed[4], seed[5])
        # use the library function to set the sim seed
        csetseed(c_seed)
        # g takes only one observation so set the c_func parameter to 1
        c_n = c_uint(1)
        # c_func requires is an integer so convert it -- this is a 1D example
        c_x = c_int(x[0])
        # call the C function
        mysimout = csimout(c_x, c_n)
        if not mysimout.feas:
            is_feasible = False
        else:
            is_feasible = True
        if is_feasible:
            objective_values = tuple(mysimout.obj)
        return is_feasible, objective_values
```

Figure 14 **The g function wraps an external simulation written in C, and maintains compatibility with common random numbers and taking simulation replications in parallel.**

RA, each RA iteration begins the next available independent stream 2^{127}, where PyMOSO accounts for the possibility of parallel computation within an RA iteration. Thus, in a given RA iteration, a user may simulate 100 million points at a sample size of 1 million, without common random numbers, and easily not reach the limit.

A.4.2. Implementing PyMOSO Testers Consider again the example tester in Figure 12. As a minimal valid PyMOSO tester, users may do nothing but assign the `MyTester` member `self.ranorc` to a PyMOSO oracle, such as `MyProblem`, in Line 27. However, we expect most users to leverage PyMOSO features by implementing metrics and feasible point generators.

The function `get_ranx0` allows the tester to generate feasible points to `MyProblem` and `metric` allows the tester to compute a metric on sets returned by a solver. Researchers may implement any number of additional supporting functions, including members and methods of the tester class. The `true_g` function is an example of such a supporting function, which is used to compute the example metric.

First, we list the rules for implementing a feasible point generator.

1. The function is arbitrarily named but must be set to the `self.get_ranx0` member of a tester.
2. The function must take a single parameter, an arbitrarily named `random.Random` object we suggest naming `rng`.
3. The function must return a tuple with length corresponding to the `self.dim` member of the `self.ranorc` member of the tester.

Since a researcher's desired metric depends on the algorithm capabilities and problem complexity, PyMOSO allows researchers to implement any metric they choose. We provide three example metrics, but first, we list the implementation rules of the `metric` function.

1. The `metric` function must be an instance method of a tester, and thus take `self` as its first parameter.
2. The second parameter of `metric` is arbitrarily named and is a Python set of tuples.
3. PyMOSO does not enforce the return value of `metric`, but we recommend a scalar real number.

The metric implemented in Figure 12 is the Hausdorff distance from (a) the true image of an estimated solution returned by an algorithm, to (b) the true solution hard-coded as `myanswer`.

For an example of a different metric, consider a MOSO problem that has more than one local efficient set (LES) and such that each LES contains no members of another LES. Since an algorithm that converges to a LES is may find only one LES, we may define the metric to compute the Hausdorff distance between the true image of the estimated solution and the "closest" true LES, as follows. Let `self.answer` be implemented as a list of sets, and assume a `self.true_g` implementation. Then Figure 15 implements the described metric.

For single-objective problems with one correct solution $\mathbf{x}^*$, a simple metric that takes an estimated solution $\mathbf{X}$ is $|g(\mathbf{X}) - g(\mathbf{x}^*)|$, which we implement in Figure 16 assuming an appropriate implementation of `self.answer` and `self.true_g`.

```
def metric(self, eles):
    # use the distance to the closest set.
    epareto = [self.true_g(point) for point in eles]
    # self.soln is a list of sets
    dist_list = [dh(epareto, les) for les in self.answer]
    return min(dist_list)
```

Figure 15 **We provide a potentially useful metric for testing MOSO algorithms that converge to a LES on problems with more than one LES, such that none of the LES's have members in common.**

```
def metric(self, singleton_set):
    # single objective algorithms still return a set
    point, = singleton_set
    # let self.soln be a real number
    dist = abs(self.true_g(point) - self.answer)
    return dist
```

Figure 16 **We provide a potentially useful metric for testing single objective algorithms.**

A.4.3. Implementing PyMOSO Algorithms Researchers can implement simulation optimization algorithms in the PyMOSO framework. PyMOSO provides support for algorithms in three categories:

1. PyMOSO provides strong support for implementing new MOSO algorithms that rely on RLE in an RA framework.
2. PyMOSO provides strong support for implementing general RA algorithms.
3. PyMOSO provides basic support, such as pseudo-random number control, for implementing other simulation optimization algorithms.

We provide templates of algorithms implemented in each of these three categories, along with example code snippets.

In the first category, programmers can use PyMOSO to create new RA algorithms that use RLE for convergence. The novel part of these algorithms, created by the user, will be the `accel` function which should collect points to send to RLE for certification. Here, we list the rules for `accel`.

1. The `accel` function must be an instance method of an `RLESolver` object, and thus its first parameter must be `self`.
2. The second parameter is arbitrarily named and is a set of tuples. We recommend naming the parameter `warm_start`, as it represents the sample-path solution of the previous RA iteration.
3. The return value must be a set of tuples representing feasible points; we do not recommend any particular name.

```
from pymoso.chnbase import RLESolver

# create a subclass of RLESolver
class MyAccel(RLESolver):
    '''Template implementation of an RLE accelerator.'''

    def accel(self, warm_start):
        '''Return a collection of points to send to RLE.'''
        # implement algorithm logic here and return a set
        return warm_start
```

Figure 17 We provide a template for implementing MOSO algorithms that use RLE for convergence.

In every RA iteration, PyMOSO will first call `accel(self, warm_start)` and send the returned set to `rle(self, candidate_les)`. The return value must be a set of tuples. The implementer does not need to implement or call RLE, as in Figure 17.

In the second category, algorithm designers can quickly implement any RA algorithm by sub-classing `RASolver` and implementing the `spsolve` function, as shown in Figure 18. The algorithm can be a single-objective algorithm. PyMOSO cannot guarantee the convergence of such algorithms. Figure 18 is technically valid in PyMOSO but is probably not effective. Though analogous to those of an `RLESolver.accel` method, for completeness, we list the requirements for an `RASolver.spsolve` method.

1. The `spsolve` function must be an instance method of an `RASolver` object, and thus its first parameter must be `self`.
2. The second parameter is arbitrarily named and is a set of tuples. We recommend naming the parameter `warm_start` as it represents the sample-path solution of the previous RA iteration.
3. The return value must be a set of tuples representing feasible points; we do not recommend any particular name.

In the third category, PyMOSO can accommodate any simulation optimization algorithm by implementing the `solve` function of a `MOSOSolver` sub-class as shown in Figure 19. It does not have to be a multi-objective algorithm. PyMOSO will require users to send an

```
from pymoso.chnbase import RASolver

class MyRAAlg(RASolver):
    '''Template implementation of an RA solver.'''

    def spsolve(self, warm_start):
        '''Return the sample path solution.'''
        # implement algorithm logic here and return a set
        return warm_start
```

Figure 18 We provide a template for implementing RA algorithms.

```
from pymoso.chnbase import MOSOSolver

class MyMOSOAlg(MOSOSolver):
    '''Template implementation of a MOSO solver.'''

    def solve(self, budget):
        while self.num_calls <= budget:
            # implement algorithm logic and return the results
        return results
```

Figure 19 We provide a template to implement a simulation optimization algorithm.

initial feasible point `x0` whether or not the algorithm needs it. The initial feasible point `x0` is accessed through `self.x0` which is a tuple. We now list the rules for implementing any `MOSOSolver.solve` function.

1. The `solve` function must be an instance method of `MOSOSolver`, and thus take `self` as its first parameter.
2. The second parameter is the simulation budget, a natural number.
3. The `solve` function must return a dictionary (we name it `results` in our example) with at least 3 keys: `'itersoln'`, `'simcalls'`, `'endseed'`. Researchers may track additional data and add it to `results` as desired.
   - The `'itersoln'` key itself corresponds to a dictionary with a key for each algorithm iteration labeled $\{0, 1, \ldots\}$. The value at each iteration is a set containing the estimated solution at the end of the iteration.
   - The `'simcalls'` key itself corresponds to a dictionary with a key for each algorithm iteration labeled $\{0, 1, \ldots\}$. The value at each iteration is a natural number containing the cumulative number of simulation replications taken at the end of the iteration.
   - The `'endseed'` key corresponds to a tuple of length 6, representing an `mrg32k3a` seed. The algorithm programmer should ensure the stream generated by `results['endseed']` is independent of all streams used by the algorithm.

Researchers may use Figure 19 to implement new simulation optimization algorithms.

For convenience, in the list below, we also provide some example code snippets that we find useful when implementing algorithms in PyMOSO. They work without modification when using the templates above that inherit `RLESolver` or `RASolver`, but some functions may require implementation or modification for use in a `MOSOSolver`. For reference, §B contains a list of most objects accessible to PyMOSO programmers.

- Example code to take simulation replications of a point at some sample size:

```
# pretend x has not yet been visited in this RA iteration and is feasible
x = (1, 1, 1)

# self.m is the sample size of the current RA iteration
m = self.m
# self.num_calls is the cumulative number of simulations used till now
start_num_calls = self.num_calls
# use estimate to sample x and put results in self.gbar and self.sehat
isfeas, fx, se = self.estimate(x)
calls_used = self.num_calls - start_num_calls
print(m == calls_used) # True
print(fx == self.gbar[x]) # True
print(se == self.sehat[x]) # True

# estimate will not simulate again in subsequent visits to a point
start_num_calls = self.num_calls
isfeas, fx, se = self.estimate(x)
calls_used = self.num_calls - start_num_calls
print(calls_used == 0) # True
```

- Example code to retrieve a point's neighbors and take simulation replications:

```
from pymoso.chnutils import get_nbors
r = self.nbor_rad
nbors = get_nbors(x0, r)
self.upsample(nbors)
for n in nbors:
  print(n in self.gbar) # True if n feasible else False
# upsample also returns the feasible subset
nbors = self.upsample(nbors)
```

- Example code to sort points by their observed objective values:

```
# 0 index for first objective
sorted_feas = sorted(nbors | {x}, key=lambda t: self.gbar[t][0])
xmin = sorted_feas[0]
fxmin = self.gbar[x]
```

- Example code to use the built-in SPLINE implementation:

```
# unconstrained minimize the 2nd objective
x0 = (2, 2, 2)
isfeas, fx, sex = self.estimate(x0)
# the suppressed value is the set visited along SPLINE's trajectory
_, xmin, fxmin, sexmin = self.spline(x0, float('inf'), 1, 0)
print(self.gbar[xmin] == fxmin) # True
```

- Example code to find the non-dominated points in a dictionary:

```
from pymoso.chnutils import get_nondom
nondom = get_nondom(self.gbar)
```

- Example code to randomly choose points from a set:

```
solver_rng = self.sprn
# pick 5 points -- returns a list, not a set.
ran_pts = solver_rng.sample(list(nondom), 5)
one_in_five = solver_rng.choice(ran_pts)
```

A.5. Using `solve` and `testsolve` in Python Programs

Users may invoke the `solve` and `testsolve` functions within a Python program.

- Using `solve` in a Python program is similar to using the CLI `solve`. We provide the minimal example here.

```
# import the solve function
from pymoso.chnutils import solve
# import the module containing the RPERLE implementation
import pymoso.solvers.rperle as rp
# import MyProblem - myproblem.py should usually be in the script directory
import myproblem as mp

# specify an x0. In MyProblem, it is a tuple of length 1
x0 = (97,)
soln = solve(mp.MyProblem, rp.RPERLE, x0)
print(soln)
```

- Users can specify options, including algorithm-specific parameters, as shown below.

```
# example for specifying budget and seed
budget=10000
seed = (111, 222, 333, 444, 555, 666)
soln1 = solve(mp.MyProblem, rp.RPERLE, x0, budget=budget, seed=seed)

# specify crn and simpar
soln2 = solve(mp.MyProblem, rp.RPERLE, x0, crn=True, simpar=4)

# specify algorithm specific parameters
soln3 = solve(mp.MyProblem, rp.RPERLE, x0, radius=2, betaeps=0.3, betadel=0.4)

# mix them
soln4 = solve(mp.MyProblem, rp.RPERLE, x0, crn=True, seed=seed, radius=5)
```

- Using `testsolve` in a Python program is also similar to using the CLI `testsolve`. Here, we provide an example with options.

```
# import the testsolve functions
from pymoso.chnutils import testsolve
# import the module containing RPERLE
import pymoso.solvers.rperle as rp
# import the MyTester class
from mytester import MyTester

# testsolve needs a "dummy" x0 even if MyTester will generate them
x0 = (1, )
run_data = testsolve(MyTester, rp.RPERLE, x0, isp=100, crn=True, radius=2)
```

- When using `testsolve` in a Python program, users must compute their metric. Here, `run_data` is a dictionary of the form described in §A.4.3, in the description of Figure 19. In the snippet below, we compute the metric on the 5th algorithm iteration of the 12th independent sample path.

```
iter5_soln = run_data[11]['itersoln'][4]
isp12_iter5_metric = MyTester.metric(iter5_soln)
```

B. PyMOSO Programming Object List

We describe the object names inside each of the following `pymoso` modules.

`prng.mrg32k3a` The module exposes the pseudo-random number generator and functions to manipulate it.

`MRG32k3a` Sub-class of random.Random, defines all `rng` objects.

`get_next_prnstream(seed)` Return an `rng` object seeded 2^{127} steps from the input seed.

`jump_substream(rng)` Seed the input `rng` object 2^{76} steps forward.

`chnbase` The module implements the base classes for programming oracles and solvers.

`Oracle` Base class for implementing oracles.

`RLESolver` Base class for implementing solvers using RLE.

`RASolver` Base class for implementing RA solvers.

`MOSOSolver` Base class for all solvers.

`chnutils` The module contains generally useful functions for programming or testing algorithms.

`solve(oracle, solver, x0, **kwargs)` See §A.5.

`testsolve(tester, solver, x0, **kwargs)` See §A.5.

`does_weak_dominate(g, h, relg, relh)` All inputs are tuples of equal length. Returns True if `g` weakly dominates `h` with relaxations.

`does_dominate(g, h, relg, relh)` Returns `True` if `g` dominates `h` with relaxations.

`does_strict_dominate(g, h, relg, relh)` Returns `True` if `g` strictly dominates `h` with relaxations.

`get_nondom(obj_dict)` Input: a dictionary with tuples for keys and values. The keys are feasible points; the values are their objective values. Return: a set of tuples representing non-dominated points.

`get_nbors(`$\mathbf{x}$, r`)` Input: a tuple $\mathbf{x}$, a positive real scalar r indicating the neighborhood radius. Return: Set of tuples, the neighbors.

`get_setnbors(`$\mathcal{S}$, r`)` Input: a set of tuples, and the neighborhood radius. Return: $\cup_{\mathbf{x}\in\mathcal{S}}$ `get_nbors(`$\mathbf{x}$, r`)`.

`dh(A, B)` Returns the Hausdorff distance between set `A` and set `B`.

`edist(x1, x2)` Returns the Euclidean distance between `x1` and `x2`.

`gen_metric(results, tester)` Input: `results` is a dictionary, the output of each sample path of `testsolve`. `tester` must implement `metric`. Returns: The set of triples (iteration, simulation count, metric) for an algorithm run.

`Oracle` When implementing `RAsolver` algorithms, programmers may not need to access `Oracle` objects directly at all. When implementing `MOSOSolver` algorithms, programmers will use (or wrap) `hit` and `crn_advance()`.

`Oracle.num_obj` A positive integer, the number of objectives.

`Oracle.dim` A positive integer, the dimensionality of feasible points.

`Oracle.rng` An instance of `MRG32k3a` internal to the oracle.

`Oracle.hit(x, n)` Take `n` observations of `x`. Return: `True`, and a tuple containing the mean of the observations for each objective `se`, and a tuple containing the standard error for each objective if `x` is feasible. The function handles CRN internally.

`Oracle.set_crnflag(bool)` Turn CRN on (`True`) or off.

`Oracle.set_crnold(state)` Save the `rng` state as the CRN baseline, e.g. for an algorithm iteration.

`Oracle.crn_reset()` Back the oracle `rng` to the CRN baseline.

`Oracle.crn_advance()` If CRN is on, reset, and then jump to the next independent pseudo-random stream and save the new baseline, e.g. before starting a new algorithm iteration.

`Oracle.crn_setobs()` Set an intermediate CRN for individual oracle observations.

`Oracle.crn_nextobs()` Jump the `rng` forward, e.g. after taking an observation, and `set_obs` the seed.

`Oracle.crn_check()` If CRN is on, return to the baseline. Otherwise, use `nextobs` before taking the next observation.

`MOSOSolver` The base class provides a basic structure for implementing new MOSO algorithms in PyMOSO.

`MOSOSolver.orc` The oracle object for the solver to solve.

`MOSOSolver.dim` Number of dimensions of points in the `self.orc`'s feasible points.

`MOSOSolver.num_obj` Similarly, the number of objectives in `self.orc`.

`MOSOSolver.num_calls` A running count of the number of observations taken of `self.orc`.

`MOSOSolver.x0` A feasible starting point. This point is additionally supplied to algorithms that don't need one.

`RASolver` Implements a common structure for all RA algorithms, including: caching of simulation replications, scheduling and updating of sample sizes and limits, and a wrapper to `Oracle.hit`.

`RASolver.sprn` An instance of `MRG32k3a` for the solver to use.

`RASolver.nbor_rad` The neighborhood radius used by solvers seeking local optimality.

`RASolver.gbar` A dictionary where every key and value is a tuple. The keys are feasible points, values are their objective values. `gbar` is "wiped" every retrospective iteration.

`RASolver.sehat` Exactly like `gbar` except the values are standard errors.

`RASolver.m` The sample size of the current iteration.

`RASolver.calc_m(nu)` Compute the sample size of the current iteration. RA algorithms automatically do this every iteration and assign the value to `self.m`.

`RASolver.b` The searching sample limit of the current iteration.

`RASolver.calc_b(nu)` Exactly as `calc_m` but for the searching sample limit.

`RASolver.estimate(x, c, obj)` The `estimate` function is essentially a smart wrapper for `self.orc.hit`. Inputs: tuple `x` to sample, `c` a feasibility constraint, `obj` the objective to constrain. Return: same as `Oracle.hit`. Retrieves or saves the results from/to `gbar` and `sehat` as appropriate. Returns not feasible if the otherwise feasible result is not less than the constraint.

`RASolver.upsample(mcS)` A version of `estimate` for sets. Returns the feasible subset of `mcS`.

`RASolver.spline(x, c, obmin, obcon)` Return a sample path local minimizer. Input: a feasible start, constraint, objective to minimize, objective to constrain. Return: a set of tuples of the trajectory, the minimizer tuple, the minimum tuple, the standard error tuple.

`RLESolver` Builds on `RASolver` to add RLE and its relaxation.

`RLESolver.betadel` Affects the relaxation values computed in RLE.

`RLESolver.calc_delta(se)` Computes the RLE relaxation given a standard error, using `self.m` and `self.betadel`

`RLESolver.rle(candidate_les)` Input: set of tuples, Returns: set of tuples. Finds the LES at sample size `self.m`.