SimOpt: A Testbed for Simulation-Optimization Experiments

David J. Eckman
Wm Barnes '64 Department of Industrial and Systems Engineering
College Station, Texas 77843

Shane G. Henderson
School of Operations Research and Information Engineering
Cornell University
Ithaca, New York 14853

Sara Shashaani
Edward P. Fitts Department of Industrial and Systems Engineering
North Carolina State University
Raleigh, North Carolina 27695

Abstract

This paper introduces a major redesign of SimOpt, a testbed of simulation-optimization (SO) problems and solvers. The testbed promotes the empirical evaluation and comparison of solvers and aims to accelerate their development. Relative to previous versions of SimOpt, the redesign ports the code to an object-oriented architecture in Python; includes an implementation of the MRG32k3a random-number generator that supports streams, substreams and subsubstreams; supports the automated use of common random numbers for efficiency; includes a powerful suite of plotting tools for visualizing experiment results; uses bootstrapping to obtain error estimates; accommodates the use of data farming to explore simulation models and optimization solvers as their input parameters vary; and provides a graphical user interface. The SimOpt source code is available on a GitHub repository under a permissive open-source license.

1 Introduction and Motivation

A simulation-optimization (SO) problem is an optimization problem where the objective function and/or constraints are evaluated (approximately) through a stochastic simulation. A wide variety of problems can be formulated as SO problems, and while quite a few SO solvers exist, there is tremendous room for more development. SimOpt [Eckman et al., 2021a] is a suite of SO problems and solvers that supports the testing and development of SO solvers, among many other uses. SimOpt has evolved over quite some time [Pasupathy and Henderson, 2006], with the most recent version prior to this release built in MATLAB with a standardized interface and automated generation of a small number of plots [Eckman et al., 2019].

This paper introduces a comprehensive re-imagining and redesign of SimOpt that significantly enhances its functionality while increasing its ease of use. In addition to SimOpt's original purpose of providing a testbed for benchmarking solvers and spurring their development, the latest incarnation permits additional uses. These include

1. sensitivity analysis of simulation models through data farming;
2. assisting with the tuning of input parameters of SO solvers;
3. educational uses, offering a set of models for students to explore. Writing new problems and/or solvers for the library could be a suitable final project for a graduate course in simulation or simulation optimization.

We highlight the following innovations in SimOpt:

- An object-oriented architecture in Python that enables both open access and extensibility to new domains, e.g., data farming.
- An implementation of the MRG32k3a random number generator that supports streams, substreams and subsubstreams.
- A schema for controlling random numbers that allows the use of common random numbers in a variety of ways with almost no effort on the part of the user.
- An expanded suite of plotting tools for visualizing experiment results.
- A bootstrapping approach to error estimation that permits a broad range of analyses.
- A data farming capability that allows one to test how changes to parameters of simulation models or simulation optimization solvers affect their outputs.
- A GUI that increases ease of use.

The result of these innovations is a powerful platform that we hope will become a standard medium for the study of SO problems and solvers.

Our work is partly inspired by the recent development of PyMOSO [Cooper and Hunter, 2020, 2021] for multi-objective SO problems. Neither SimOpt nor PyMOSO dominates the other in scope. While PyMOSO has the ability to work with the single-objective SO problems that are our focus, it is primarily intended for multi-objective problems. On the other hand, SimOpt latently supports multi-objective SO problems but will require further development before this capability is fully implemented. We believe that many of the innovations we develop here will be of interest to users of PyMOSO, e.g., the use of simulation *models* that exist separately from simulation *problems*. In any case, we view both PyMOSO and SimOpt to be important tools for the SO research community.

The remainder of this paper is organized as follows. Section 2 defines single-objective SO problems and delineates the classes of SO problems and solvers supported by SimOpt. Section 3 discusses the infrastructure of SimOpt including how experiments may be designed and evaluated. Section 4 explores several use cases. Section 5 reviews the central aspects of the SimOpt code, which will be helpful to those wishing to contribute problems or solvers. It also provides a deeper understanding of the code for those interested in advanced experiments. Section 6 describes SimOpt's implementation of MRG32k3a and how it enables the use of common random numbers in a variety of ways. Section 7 describes in detail how one can access and work with SimOpt, and Section 8 concludes.

2 SO Problems and Solvers

The prototypical SO problem we consider is

$$\begin{aligned} \min_x\, & f(x, w) = \mathbb{E} f(x, w, \xi) \\ \text{s/t}\ & g(x, w) = \mathbb{E} g(x, w, \xi) \geq 0 \\ & h(x, w) \geq 0 \\ & x \in \mathcal{D}(w). \end{aligned}$$

In this formulation, x is a vector of decision variables that takes values in a domain $\mathcal{D}(w)$ that could, for example, restrict x to be integer-valued and could depend on w. The vector w consists of input parameters that are not decision variables, but provide additional "settings" for the problem that the user may wish to modify. Collectively we refer to (x, w) as *factors* to align with the design-of-experiments literature. Factors can be continuous or integer-ordered scalars or vectors, or even categorical in nature. In a single problem the factors w can be viewed as fixed constants, but in, e.g., data-farming, they could be varied. In the data-farming setting, we refer to x and w as decision factors and noise factors, respectively [Sanchez, 2020].

The explicit dependence on the noise factors, w, is not typical in the notation of SO problems. We adopt this notation because SimOpt has been structured to allow almost any parameter of a model to be varied. This design choice is intended to make the code as flexible as possible for user (re)specification. For example, this allows the definition of multiple problems that all rely on the same underlying simulation model and code. Varying the values of the factors w will usually lead to only modest changes in the structure of the problem, but could lead to more substantial changes in properties like continuity, convexity, and smoothness. In addition, varying w can change the optimal solution of the problem or, if the problem is constrained, even make the problem infeasible in that no x is feasible for the given w.

The random object ξ represents all random variables required to generate a single replication. The expectation operator $\mathbb{E}$ potentially depends on x and w, in that these factors may change the distribution of the random object ξ, but that dependence is suppressed. The functions $f(\cdot, \cdot, \cdot)$ and $g(\cdot, \cdot, \cdot)$ represent the simulation logic used to generate the objective function and left-hand side of any stochastic constraints, respectively, so that $f(\cdot, \cdot, \cdot)$ is real-valued and $g(\cdot, \cdot, \cdot)$ is potentially vector-valued. The potentially vector-valued function

$h(\cdot,\cdot)$ provides the left-hand side of any deterministic constraints. Some constraints could be expressed through both the function h and through the domain $\mathcal{D}(w)$. In such cases, the choice of which to use is a matter of taste and convenience.

This formulation is not completely general. It excludes, for example, objective functions associated with quantiles or nonlinear functions of means. It includes, e.g., unconstrained problems where g and h are vacuous and $\mathcal{D}$ is the domain of $f(\cdot, w)$ with w fixed, problems with box constraints where $\mathcal{D}$ or h restricts the decision variables to a hyper-rectangle aligned with the coordinate axes, and so forth. It includes problems where all decision variables are continuous and also problems where some or all decision variables take integer values. When g is vacuous and h is not, we say the problem has deterministic constraints. If g is non-vacuous then we say the problem has stochastic constraints.

We differentiate between the *model* that appears in an SO problem and the *problem* itself. This allows us to formulate multiple problems associated with a single model. Moreover, models can be studied in isolation; for instance, data farming can be employed to understand how changes to the inputs of a simulation model affect its outputs. Given that the code of a simulation model is usually more complex than that of a problem, this one-to-many relationship helps us rapidly expand the collection of problems in SimOpt.

Example 1 *The SimOpt model* ***`FacilitySize`*** *simulates operations at a set of n facilities with each replication representing a day's worth of operations. Each facility has a capacity κ_i for $i = 1, 2, \ldots, n$ and the demand across all centers is distributed as a truncated Gaussian with mean vector $\mu \in \mathbb{R}^n$ and variance-covariance matrix $\Sigma \in \mathbb{R}^{n\times n}$. If the demand at a facility exceeds its capacity, the facility is said to be stocked out. This model has four factors: n, $\kappa := (\kappa_1, \kappa_2, \ldots, \kappa_n)$, μ, and Σ, which we see can be scalars, vectors, and matrices. A replication of the model returns three responses: S, an indicator of whether any facility stocked out; N, the number of facilities that stocked out; and U, the total amount of unsatisfied demand.*

The ***`FacilitySize`*** *model can be used to formulate several SO problems, e.g.,*

$$\min_{\kappa\in\mathbb{R}^k_+} c^\top \kappa \text{ such that } \Pr(N=0) \geq 1-\epsilon \quad \text{and} \tag{1}$$

$$\max_{\kappa\in\mathbb{R}^k_+} \Pr(N=0) \text{ such that } c^\top\kappa \leq b, \tag{2}$$

where $c_i \in \mathbb{R}$ is the cost of installing a unit of capacity at center i, $\epsilon \in (0,1)$ is an allowable threshold for the probability of stocking out, and $b \in \mathbb{R}$ is a total budget for installation costs. The parameters $c = (c_1, c_2, \ldots, c_n)$, ϵ, and b are regarded as factors of the problems and can be varied. In Problem (1), the objective is to minimize the total installation costs subject to a stochastic constraint that the probability of not stocking out anywhere is sufficiently high. In Problem (2), the objective is to minimize the probability of not stocking out at any facility subject to a deterministic constraint on the total cost of installing capacity at the facilities. Both problems designate the capacities as the decision variables, i.e., $x = \kappa$, while treating the other factors as fixed, i.e., $w = \{n, \mu, \Sigma, c, b, \epsilon\}$.

SimOpt does not include artificial problems that result from adding noise to a deterministic test function such as the Rosenbrock function. Such problems have serious deficiencies that can arise, e.g., when using common random numbers across different solutions. In such cases, the entire deterministic function is shifted vertically by a single random noise realization at all solutions x, as has been noted elsewhere [Eckman et al., 2021b].

In SimOpt, problems are solved by *solvers*, which are implementations of SO algorithms. We classify solvers according to the kinds of problems they can tackle using the same terminology we use to describe problems. Thus, for example, a solver designed for continuous-variable problems cannot be used on a problem with integer variables, and a solver designed for unconstrained problems cannot be used on a problem with stochastic constraints. At present, SimOpt is not designed to directly support ranking-and-selection algorithms that enumerate a finite list of potential solutions, though the models and problems in SimOpt could be adapted to that setting.

Like models and problems, solvers have their own set of factors that can be varied. These can be virtually anything, but representative examples include (1) nothing; (2) coefficients of a step-size sequence; (3) the number of replications to take at each simulated solution; or (4) a categorical variable indicating whether to use a first- or second-order metamodel around the incumbent solution.

At present, few SimOpt test problems return estimates of the gradient of $f(\cdot,\cdot)$, so any gradient-based solvers need to indirectly construct the gradient estimates they require. However, we continue to develop the list of problems in the library and add gradient estimators where we can. An exciting potential research direction is to use automatic differentiation to obtain infinitesimal-perturbation-analysis gradient estimators for *all* problems, though the quality of those estimators would likely be highly variable.

3 Solver Performance

From its inception, SimOpt has sought to answer the question "How do we know if a solver is working well?" SimOpt is designed to help both a researcher who might appreciate testing a solver's ability to rapidly and reliably solve practical problems, and a practitioner who would primarily be interested in solving a particular problem of interest. In this section, we describe how SimOpt runs an SO solver on a problem and reports useful metrics and plots for evaluating and comparing performance.

Unlike with deterministic-optimization solvers, the performance of an SO solver on a given problem varies from run to run due to the random error associated with estimating the objective function and/or stochastic constraints, as well as any intrinsic randomness of the solver, e.g., picking a random search direction. This necessitates performing multiple runs of a solver on a problem, hereafter referred to as *macroreplications*. For a given problem p and solver s, the solver's performance on a particular macroreplication is assessed by fixing a problem-specific simulation budget T—measured in simulation replications—and tracking the solutions recommended over time. In particular, the mth macroreplication generates a stochastic process $\{X_m^{p,s}(t)\colon 0 \leq t \leq 1\}$, where $X_m^{p,s}(t)$ is the solution recommended by Solver s on Problem p after a fraction $t \in [0,1]$ of the budget has been expended. When there is no ambiguity, we suppress p and s from the notation. The recommended solutions are then re-evaluated in a post-processing stage to obtain unbiased objective function estimates and the results are scaled to obtain the solver's relative progress toward optimality. Effectively, SimOpt estimates a solver's progress via two-level simulation, with an outer level consisting of macroreplications and an inner level consisting of *postreplications.* This experimental setup for a given problem-solver pair is outlined below.

Step 1. Run $M \geq 1$ independent macroreplications of Solver s on Problem p to generate $\{X_m(t)\colon 0 \leq t \leq 1\}$ for $m = 1, 2, \ldots, M$.

Step 2. Take N independent postreplications of the model at each distinct recommended solution in $\{X_m(t)\colon 0 \leq t \leq 1\}$ for $m = 1, 2, \ldots, M$. The objective function value associated with a recommended solution $X_m(t)$ is estimated by the sample average of the N postreplications, denoted as $f_N(X_m(t))$. (The left-hand sides of any stochastic constraints can be estimated similarly as $g_N(X_m(t))$.)

Step 3. For each solution recommended in macroreplication m, normalize its estimated objective function value using the (estimated) objective function values at an initial solution x_0 and an optimal solution x^* for reference:

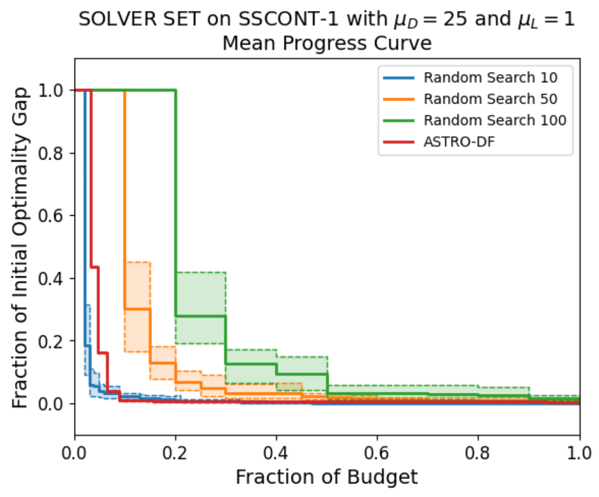
$$\nu_m(t) = \frac{f_N(X_m(t)) - f_L(x_0)}{f_L(x^*) - f_L(x_0)}.$$

The objective function values of x_0 and x^* are estimated based on L postreplications, where typically $L \geq N$. We call the normalized value $\nu_m(t)$ the estimated progress at time t on macroreplication m and we call $\nu_m(\cdot)$ the estimated *progress curve.* A progress curve typically takes values between zero and one, though this is not guaranteed due to sampling variability or other causes. When a problem's true optimal solution is unknown, as is often the case, a known optimal value $f(x^*)$ or a lower (upper) bound on the optimal value for minimization (maximization) objectives may be provided and used in place of $f_L(x^*)$. If these quantities are not provided, SimOpt empirically identifies a proxy optimal solution by using the recommended solution with the best estimated objective function value based on the N postreplications.

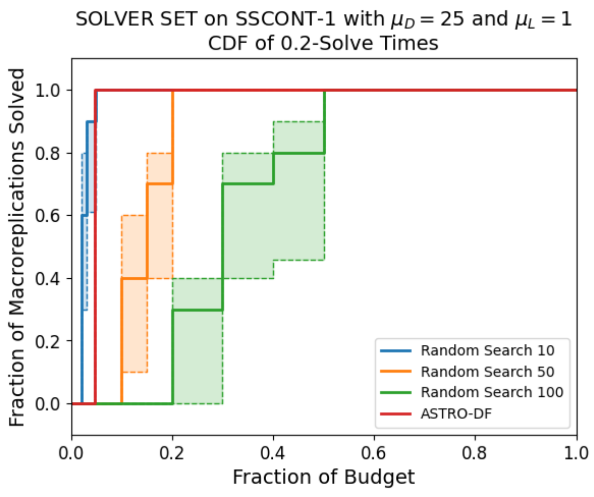
Estimates of many measures of solver performance can be extracted from the estimated progress curves and plotted. SimOpt uses a two-level bootstrapping procedure to obtain error estimates for these metrics, as outlined in Appendix B in Eckman et al. [2021b]. We summarize four types of comparative plots introduced in Eckman et al. [2021b] and show examples in Figure 1; the first two measure solver performance on a single problem while the last two measure solver performance on a set of problems.

- *Aggregated Progress Curves.* The estimated progress curves $\nu_1(\cdot), \nu_2(\cdot), \ldots, \nu_M(\cdot)$ can be aggregated to produce a mean progress curve and a quantile progress curve. These curves depict the solver's average progress over time and how reliable this progress is, respectively. Solvers for which the aggregated progress curves approach 0 more quickly are those that make more rapid progress.

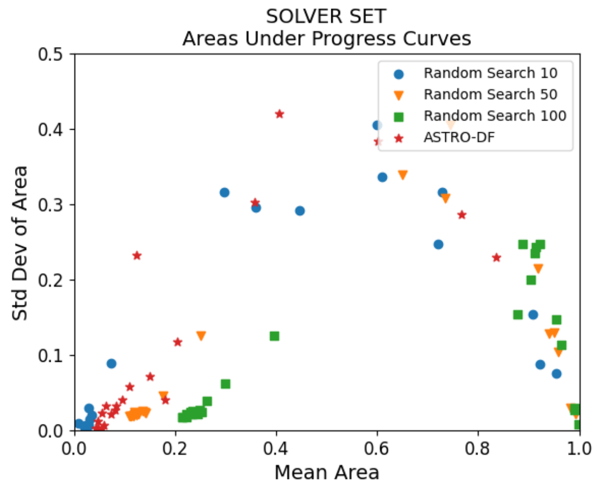
- *Solvability Curves.* One can specify a value $\alpha \in (0, 1)$ that indicates the relative remaining optimality gap required for a problem to be deemed "solved". The corresponding crossing time of each estimated progress curve $\nu_m(\cdot)$ is referred to as the



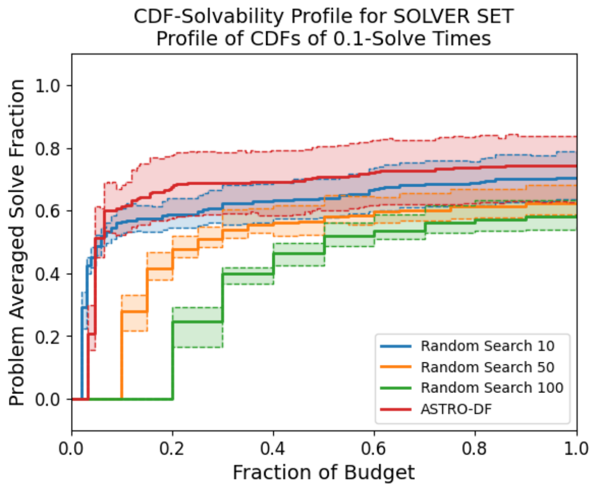
(a) Mean progress curves



(b) Solvability curves



(c) Area-under-progress-curve scatter plots



(d) Solvability profiles

Figure 1: Examples of plots produced by SimOpt for comparing solvers on one or more problems.

α-solve time. The empirical cumulative distribution function of the α-solve times from multiple macroreplications is called a solvability curve and shows how rapidly a solver makes sufficient progress.

- *Area-Under-Progress-Curve Scatter Plots.* For each problem, the sample mean and standard deviation of the area under a solver's estimated progress curves can be plotted in a scatter plot using these summary statistics as (x, y) coordinate pairs. Solvers whose point clouds are concentrated in the lower-left corner of the scatter plot are those that either find better solutions or exhibit faster convergence with greater reliability.

- *Solvability Profiles and Difference Profiles.* The α-solve times of a solver can also be aggregated across problems to yield a *solvability profile*, which has close ties to data profiles [Moré and Wild, 2009]. Solvers with solvability profiles closer to 1 demonstrate better performance at solving a larger fraction of the tested problems. Comparisons between a set of solvers and that of a benchmark solver s_0 can be further highlighted by plotting the difference of solvability profiles, called difference profiles (not shown). The values of a difference profile range between -1 and 1 with positive values indicating a solver outperforming the benchmark.

These metrics and plots have limitations that direct our future endeavors for SimOpt. First, they are not designed for problems with stochastic constraints, or at least do not depict the (in)feasibility of recommended solutions. Second, they are not designed for multi-objective SO problems, which have their own unique aspects when it comes to measuring a solver's performance.

4 Use Cases

The performance metrics and plots discussed above provide a wealth of information on the performance of solvers on problems. Here we discuss how that information can be used in a variety of settings that encompass both practitioners attempting to solve one or more problems and researchers attempting to develop solvers and compare them.

1. How well can Problem p be solved? In practice, we often seek a good solution to a given problem. The experiment consists of the singleton problem set $\mathcal{P} = \{p\}$ together with a collection of candidate solvers $\mathcal{S}$. Progress and solvability curves are both of interest, but a challenge is that the optimal value, or a proxy thereof, is likely unknown. In that event, unnormalized progress curves may be of primary interest.

2. Is Solver s able to solve Problem p? Here a solver developer is interested in whether Solver s can α-solve Problem p for some given α. Since the goal here is solver design rather than problem solution, an optimal value or a proxy may be known. Accordingly, an experiment with a singleton problem set $\mathcal{P} = \{p\}$ and a singleton solver set $\mathcal{S} = \{s\}$ might be run to produce progress and solvability curves.

3. Does Solver s solve Problem p faster or more reliably than Solver s'? A solver developer may want to know how their solver compares with another benchmark solver on a particular problem. We can run an experiment with $\mathcal{P} = \{p\}$ and $\mathcal{S} = \{s, s'\}$. Mean and median progress curves can provide the typical rate of progress and other (than the median) quantile progress curves can provide information about solver reliability. Here s and s' might be the same solver with different factors, thereby facilitating the tuning of solvers, or two entirely different solvers. In this and the next two use cases, the number of solvers being compared can of course be more than two. For instance, if one wishes to explore variants of the solver s—as expressed through choices of solver factors $v_1, v_2, \ldots, v_r$ say—then one would use the solver set $\mathcal{S} = \{s(v_1), s(v_2), \ldots, s(v_r)\}$.

4. Is Solver s more robust than Solver s' in solving problems of the form p? Many applications involve the repeated solution of problems that are very similar. We can explore which solvers are especially adept at solving a class of problems by taking $\mathcal{P} = \{p(w_1), p(w_2), \ldots, p(w_r)\}$, where $w_1, w_2, \ldots, w_r$ are choices of factors. If we take these factor settings to be the initial solution, then we can explore the global convergence properties of solvers. If we take these factor settings to reflect the size, noise-to-signal ratio, or other specifics of Problem p, we can see how the two solvers compare in solving this class of problems. Less consistency in a solver's performance here implies less robustness. Area-under-progress-curve scatter plots, solvability profiles, and difference profiles are all of interest.

5. Is Solver s better than Solver s'? When the user is interested in an overall assessment of one solver versus another, it would be appropriate to perform an experiment with a comprehensive problem set $\mathcal{P} = \{p_1, p_2, \ldots, p_r\}$ that includes many problem instances encompassing a wide range of structural properties and noisy behaviors. Alternatively, the user can choose $\mathcal{P}$ to contain only problems that are hard to solve by current solvers and include an existing high-functioning solver as a benchmark for difference profiles. Or a user might be interested in a comparison of only low (high) dimensional problems. This use case, as well as Use Cases 3 and 4 can help a solver developer test specific internal changes to an existing solver. What it means for a solver to be "better" is subjective and dependent on the needs of the user; speed and reliability are important and can be explored in any of the plots. If the number of problems in $\mathcal{P}$ is reasonably large, solvability and difference profiles will clarify whether one solver is capable of solving a higher percentage of problems than the other. Area-under-progress-curve scatter plots are also beneficial in showing on which problem instances each solver performs poorly.

6. What choices of factors of Solver s work best on a problem or a set of problems? Similar to Use Case 3, a user may want to explore the factors v of Solver s, looking for the (parameterized) solver $s(v)$ with the best performance. For example, data farming [Sanchez, 2020] can be used to design a space-filling set of factor combinations $\mathcal{V} = \{v_1, v_2, \ldots, v_r\}$. The output data from such an experiment can then be used to form response surfaces based on the various performance measures listed in Section 3. A user might seek, for example, the setting of the solver's factors that minimizes some functional of the area under the progress curves. In this way, one might establish some rules of thumb for tuning a solver to a particular problem or class of problems.

7. Can the relationship between the inputs and outputs of a simulation model be understood? Here there is no optimization problem to be solved. Rather, one wishes to explore the effects that a given simulation model's factors have on its responses. The necessary simulations can be run in SimOpt; however, the statistical analysis of the results needed to, for instance, generate a response surface must performed externally.

5 Code Design

Previous versions of SimOpt were coded in MATLAB—a choice that was partially an artifact of how the library's first problems and solvers were coded. Our major redesign of SimOpt presented an opportunity to revisit the choice of programming language. Feedback from the simulation community indicated that Python would be an ideal choice for other researchers to use and contribute code. Python is an open-source programming language that has arguably become the de facto choice for scientific computing. Like MATLAB, Python supports object-oriented programming—a central tenet of the redesign. The decision to convert the SimOpt library to Python was also influenced by the existence of another SO library written in Python: PyMOSO [Cooper and Hunter, 2020]. The PyMOSO library provides a Python implementation of the `MRG32k3a` pseudorandom-number generator [L'Ecuyer, 1999, L'Ecuyer et al., 2002], which we further adapted for our purposes; see Section 6.1.

We decided to rebuild SimOpt with an object-oriented design after seeing a similar architecture in the PyMOSO library. In libraries like these, specific problems or solvers are naturally encoded as subclasses of more general `Problem` and `Solver` classes. The overall object-oriented design of SimOpt was developed by considering the kinds of experiments we intended to support and identifying the main entities and how they interact. There is a base design that reflects SimOpt's role as a *library* of problems and solvers with classes including `Model`, `Problem`, `Solver`, and `Solution`. Around this base design is another layer aligned with SimOpt's role as a *testbed* for running experiments; classes at this level include `ProblemSolver` and `ProblemsSolvers`. Woven throughout the design are pseudorandom-number generators that are used for a variety of purposes. We overview the base design and experimental layer in Sections 5.1 and 5.2, respectively, and discuss pseudorandom-number generation in Section 6.

Remark 1 *The object-oriented programming paradigm is well-suited for coding discrete-event simulations, which feature entities interacting stochastically over time. However, our discussion of object-oriented design in this paper pertains to the architecture of SimOpt. Individual models in SimOpt, many of which are discrete-event simulations, may be written as procedural or object-oriented programs.*

5.1 Models, Problems, Solvers, and Solutions

The SimOpt `Model` object represents a simulation model, i.e., a multi-valued function that takes deterministic inputs (factors) and returns one or more stochastic outputs (responses). Stochastic outputs are produced because a `Model` is equipped with one or more mechanisms for generating random primitives. A SimOpt `Problem` object enfolds an underlying `Model` and specifies which inputs of the model are decision variables and which outputs appear in the objective and/or constraints, as described in Section 2. These mappings are central to how we define classes for solutions and solvers.

An instance of the `Solution` class is associated with a vector of decision variables, x. When instantiating a `Solution` object, a `Problem` object is also provided; thus, the mappings of model factors to decision variables and responses to objectives and constraints are impressed on the `Solution` object. A `Solution` object is equipped with a set of random number generators to be used for simulating replications of the model as specified by x; for discussion in greater detail, see Section 6. Each time a solution is simulated, its summary statistics are updated. These include the sample mean and variance of the objective function values and left-hand sides of the stochastic constraints, as posed in Section 2. The individual observations of the objective function (as well those of any stochastic constraints' left-hand sides and any available gradients) are also recorded.

Remark 2 *A `Solution` object has differing concepts of* feasibility *and* simulatability. *Feasibility refers to whether the solution satisfies the constraints of the optimization problem, i.e., whether it lies in the feasible region. Simulatability refers to the ability to run a replication of the model when the problem's decision factors are set as x. For example, if a decision variable reflects the variance of a normal distribution and is negative, then a replication cannot be simulated. For a well-posed problem, all feasible solutions will be simulatable, but the converse need not hold. Both feasibility and simulatability can be checked in the code for exception handling.*

The `Solver` class represents algorithms designed to solve SO problems. Solvers are classified in terms of the number of objectives they can handle (one or multiple), the hardest type of constraints they can handle (unconstrained, box, deterministic, or stochastic, in that order), the types of decision variables they can handle (discrete, continuous, categorical, or mixed), and whether they require gradient estimates. A `Solver` object is also equipped with two sets of pseudorandom-number generators: one for its internal purposes and the other for simulating solutions. The `Solver` class has a method called `solve()` that runs one macroreplication of the solver on a given problem. On a macroreplication, a solver explores solutions, running replications of solutions as it deems appropriate until it has exhausted the problem's specified budget. Part of this process entails creating new instances of the `Solution` class when the solver visits solutions that have yet to be simulated.

To define a particular model, problem, or solver in SimOpt, one creates a subclass of the corresponding parent class (`Model`, `Problem`, or `Solver`), thereby inheriting the common attributes and methods.

5.2 Experiments with Multiple Problems and Solvers

Above the library of models, problems, and solvers, there is a level to the architecture that supports experiments that entail running multiple macroreplications of one or more solvers on one or more problems. A pairing of one solver with one problem is represented by the `ProblemSolver` class. A specified number of macroreplications are run and their results post-processed as described in Section 3. Furthermore, one can post-normalize results from `ProblemSolver` objects corresponding to multiple solvers run on the same problem. The specifics of the post-processing and post-normalization stages, namely the number of postreplications and the use of CRN, are recorded to the `ProblemSolver` object for future reference when bootstrapping. After post-normalization, the results can be plotted.

Multiple `ProblemSolver` objects can be bound together using the `ProblemsSolvers` class. An object of this class is defined by a list of solvers and a list of problems and consists of the `ProblemSolver` objects formed by taking all pairings of the problems and solvers. The `ProblemsSolvers` class facilitates running a large-scale experiment to compare the performances of solvers on a set of problems. The `ProblemSolver` objects that comprise a `ProblemsSolvers` object can be collectively post-replicated and post-normalized and their results plotted.

6 Pseudorandom-Number Design

Pseudorandom numbers pervade the design of SimOpt: simulation models use random primitives within a replication, solvers may be inherently stochastic, and bootstrapping is used to estimate errors for performance metrics. We present a schema that controls how random numbers are used throughout the testbed to run and post-process experiments on multiple problems and solvers. This design enables the user to activate common random numbers (CRN) at various levels.

6.1 Implementation of MRG32k3a

SimOpt uses the MRG32k3a pseudorandom-number generator of L'Ecuyer [1999] and L'Ecuyer et al. [2002]. The MRG32k3a generator has been shown to pass rigorous statistical tests, has a long period of approximately 2^{191}, and facilitates random number streams. In particular, advancing to the start of an arbitrary stream is computationally inexpensive. Our choice of generator is partly influenced by the Python implementation of MRG32k3a found in PyMOSO [Cooper and Hunter, 2020], which allows the user to track streams and substreams. Given the many uses of random numbers in SimOpt, we extend this implementation to permit a third (lower) level of control: subsubstreams. In our implementation, the period is split into approximately 2^{50} streams of length 2^{141}, each containing 2^{47} substreams of length 2^{94}, each containing 2^{47} subsubstreams of length 2^{47}. Where random numbers are needed, SimOpt instantiates an MRG32k3a generator—an object of class `MRG32k3a`—and seeds it at the start of a specified stream-substream-subsubstream triplet that we denote here by (s, ss, sss) for Subsubstream sss of Substream ss of Stream s. (In this paper we index starting from 1, but in Python indexing starts at 0.) By creating multiple `MRG32k3a` objects with different seeds, we control how random numbers are generated. The schema is repeated for each problem-solver pair, i.e., all `ProblemSolver` objects work with the same universe of random number streams, substreams, and subsubstreams, defined with respect to the same reference seed. The rationale for this choice is discussed in Section 6.2.

6.2 Schema for Running Experiments

We dedicate $M+1$ streams to run every experiment of a given solver on a given problem. One stream is reserved for overhead (signified by "O"), namely, the solver's internal randomness; future extensibility will allow for random initial solutions, random restart solutions, and random problem instances. Apart from the overhead stream, different streams are used for each of the M macroreplications. Within each of these streams, different substreams are used for the model's sources of randomness, and different subsubstreams are used for model replications. Thus the random-number schema for running multiple macroreplications is

Table 1: Summary of CRN management and user control.

Stage	Form of CRN	Default	Controllable
Running (optimization)	Across Solutions	✓	✓
	Across Problem-Solver Pairs	✓	
Post-Processing/Bootstrapping (evaluation)	Across Solutions	✓	✓
	Across Macroreplications		✓
	Between x_0 and x^*	✓	✓
	Across Problem-Solver Pairs	✓	

$(s, ss, sss) = (m, i, r)$, where r is the replication number of the solution being visited by the solver (during the optimization) and $i = 1, 2, \ldots, I$ is the index of the source of randomness in the model. The term "source of randomness" refers to distinct needs for uniform random numbers in a model. For example, a simple single-server queueing model might designate two sources of randomness: one that generates interarrival times and another that generates service times. (For more discussion on implementing sources of randomness, see Kelton, 2006.) In this queueing example, $I = 2$ and $(1, 1, 10)$ and $(1, 2, 10)$ denote the sequences of random numbers used to generate the arrival times and service times, respectively, for the tenth replication of a given solution visited on the first macroreplication.

SimOpt's design allows the user to flexibly control how random numbers are used according to their preferences. Specifically, the user can switch CRN on or off at various levels. We proceed to discuss these levels, working our way up from the lowest level of synchronization to the highest. Table 1 summarizes the different levels at which CRN are or can be activated, along with the default settings.

Remark 3 *At what is perhaps the lowest level, replications of a given simulation model return independent and identically distributed outputs. Specifically, when simulating a replication, all `MRG32k3a` objects used by the simulation model are advanced to the start of the next subsubstream. There is currently no support for variance-reduction techniques that induce dependent outputs across replications, e.g., antithetic variates and stratified sampling.*

CRN across solutions. The most prevalent use of CRN is synchronizing the random primitives used by a simulation model when run at different solutions. SimOpt supports this variance-reduction technique to a high degree. Each model specifies the number of sources of randomness needed to run a single replication. Random inputs for a given replication index are then synchronized across solutions using copies of the same `MRG32k3a` object, primed to start at the beginning of the subsubstream with the corresponding index. This form of CRN can help a solver determine the correct ordering of performances of the solutions it simulates on a given macroreplication and thus better identify an optimal solution; if disabled, the solver obtains independent outputs across solutions.

CRN across solvers on one problem. Consider running two solvers on the same problem. The effect of CRN across the two solvers is most pronounced when the solvers also use CRN across solutions. In this case, if the solvers ever simulate the same solution, they observe the same sequence of outputs. Thus, solvers employing a sample-average approximation effectively optimize the same sample-average functions on any given macroreplication. This form of CRN also synchronizes the random numbers used by the solver for its internal purposes, such as picking random directions and breaking ties. This synchronization has limited upside for solvers that behave very differently. Still, for different versions of the same solver, it could lead to a variance reduction in the difference between their performances. This form of CRN also influences how the performances of solvers are compared in difference profiles and other metrics.

CRN across problem-solver pairs. We can take a broader view of the previous form of CRN by allowing the problem to vary as well. As previously mentioned, all problem-solver pairs work from the same universe of random numbers and the same reference seed. In other words, the same (s, ss, sss) schema is implemented when running experiments for *any* problem-solver pair. For pairings that feature different problems *and* different solvers, this form of CRN should neither harm nor benefit a comparative analysis. We implement this form of CRN for convenience since the experimental results are easily reproducible by using Stream 1 for Macroreplication 1, Stream 2 for Macroreplication 2, etc., for all problem-solver pairs. Were distinct streams used for different problem-solver pairs, the order in which we experiment on the pairs would influence the results.

6.3 Schema for Post-Processing Experiments

A dedicated stream for post-processing, signified by "P", is used for generating random numbers within the model when simulating postreplications—the solver is not involved. Post-processing entails re-evaluating the collection of recommended solutions returned by each macroreplication of each problem-solver pair. As a consequence of working with a single stream, the substream level must now accommodate indexing over both macroreplications and sources of randomness, e.g., post-processing 50 macroreplications and 5 sources of randomness requires 250 substreams. Subsubstreams are still used for distinct replications, in this case, postreplications. Hence, the random-number schema used in the post-processing stage is $(s, ss, sss) = (\mathrm{P}, I \times (m-1) + i, n)$ where n is the postreplication number. For example, in the simple queueing model, $(\mathrm{P}, 3, 1)$ and $(\mathrm{P}, 4, 1)$ represent the sequence of random numbers used for the arrivals and service times respectively in the post-processing of the solutions on the second macroreplication.

CRN across recommended solutions on a given macroreplication. The same random numbers are used to take postreplications at each solution recommended by a solver on a given macroreplication, which helps in ranking performance of the recommended solutions.

CRN across macroreplications. Different substreams are used for the set of postreplications at solutions recommended on different macroreplications. We do not advise using CRN across macroreplications here because the results from different macroreplications are combined in some of the summary measures that SimOpt computes. Dependence across macroreplications would inflate the variance of estimators of many of those performance measures [Eckman et al., 2021b].

CRN between postreplications at x_0 and x^*. A post-normalization step involves taking a fixed number of postreplications at the initial solution x_0 and a proxy optimal solution x^*. CRN is used to take postreplications at these solutions; this helps to correctly order their performances.

CRN across problem-solver pairs. As in the schema for running experiments, all problem-solver pairs are post-processed using the same set of random numbers. This again is for convenience, not variance reduction, since the post-processing results are easily reproducible under this setup.

When producing plots, a bootstrapping procedure is optionally run to estimate the error associated with the progress curves and other metrics. For a given problem-solver pair and a user-specified number of bootstraps, the bootstrapping procedure entails resampling with replacement from the outputs of the postreplications from different solutions recommended on different macroreplications. These resampled outputs are then used to construct bootstrapped progress curves. When resampling, CRN is used exactly as implemented in the post-processing stage, but the random numbers come from yet another dedicated stream, signified by "B." The default random-number schema used in bootstrapping is $(s, ss, sss) = (\mathrm{B}, b, j)$, where b is the index of the bootstrap instance, and j denotes the distinct subsubstreams used to resample macroreplication and postreplication indexes. Further details are provided in the library's documentation.

Remark 4 *By meticulously accounting for which streams, substreams, and subsubstreams are used for different purposes, it is possible to instantiate multiple `MRG32k3a` objects—initialized at the designated seeds—and then dispatch them to a set of processors running macroreplications (or problem-solver pairs) in parallel. This degree of parallelization is not yet implemented.*

7 Deployment

This section discusses how users can access and interact with SimOpt.

7.1 Access

SimOpt is hosted in a GitHub repository [Eckman et al., 2021a]. The `master` branch contains the Python version discussed in this paper and a now deprecated MATLAB version. Although SimOpt's transition to GitHub was part of a previous redesign described in Eckman et al. [2019], it is worth reiterating the advantages this offers. Automated version control allows users to access previous versions of the code and to indicate which version they used by referencing the repository's commit hex code, e.g., `commit 79bc9e414b1ac4033858d2066ed-848cb22b030d5`. Research experiments carried out in SimOpt are thus easily reproduced.

GitHub also provides a more streamlined workflow for developing the library and troubleshooting issues with external contributions through the pull-request feature.

SimOpt also uses automatic documentation to provide up-to-date reference materials for the Python source code. This documentation is hosted at `https://simopt.readthedocs.io/en/latest/index.html` and updates with each pushed commit to the `master` branch of the library. Read the Docs (`https://readthedocs.org`) generates restructured text (.rst) files by reading the docstrings in the commented code. Models and solvers also have dedicated .rst files that provide detailed descriptions and links to external references.

7.2 Usage

The most straightforward way for users to interact with SimOpt is to *fork* the GitHub repository. Forking creates a copy of the repository on the user's personal GitHub account that they can then use for running experiments. Any branching or commits on the forked repository will not directly affect the main repository. If the user wishes for their changes to be incorporated into the main repository, as might arise if they were to fix a bug or to contribute a new model, problem, or solver, they can initiate a *pull request.* The pull request notifies the development team of the requested changes, which are then reviewed before being merged into the main repository.

Remark 5 *This setup requires users to have a GitHub account, which can be obtained free of charge. Many researchers already use GitHub repositories to maintain source code for experiments featured in their published work. In the future, we hope to package SimOpt so that users can `pip install` it through the Python Package Index (PyPi) (`https://pypi.org/`).*

After forking the repository, users should *clone* it to their personal computer and open the `simopt/simopt` folder within their preferred integrated development environment. Users can then run experiments by calling functions from the command line or using a graphical user interface (GUI).

7.3 Command-Line/Scripting Interfacing

The file `wrapper_base.py` contains a listing of high-level functions for running and post-processing SO experiments and plotting the results. Likewise, the file `data_farming_base.py` defines functions and classes for data-farming experiments that involve varying factors of the models. While users can read the documentation for these functions and directly call them from the command line, the folder `demo` contains a handful of Python scripts that interact with the source code at different levels. By modifying a few lines of code in these files, as directed in the comments, users can specify the model, problem, and/or solver they wish to study and override the default values of any factors. These scripts provide a mechanism for testing without invoking higher-level wrappers.

- `demo_model.py`: Run multiple replications of a simulation model and report its responses.

- `demo_problem.py`: Run multiple replications of a given solution for an SO problem and report its objective function values and left-hand sides of stochastic constraints.
- `demo_solver_problem.py`: Run multiple macroreplications of a solver on a problem, save the outputs to a .pickle file in the `experiments/outputs` folder, and save plots of the results to .png files in the `experiments/plots` folder.
- `demo_data_farming_model.py`: Create a design over model factors, run multiple replications at each design point, and save the results to a comma separated value (.csv) file in the `data_farming_experiments` folder.

Another script called `demo_sscont_experiment.py` demonstrates how experiments with multiple solvers and multiple problems were run to produce the plots in Figure 1.

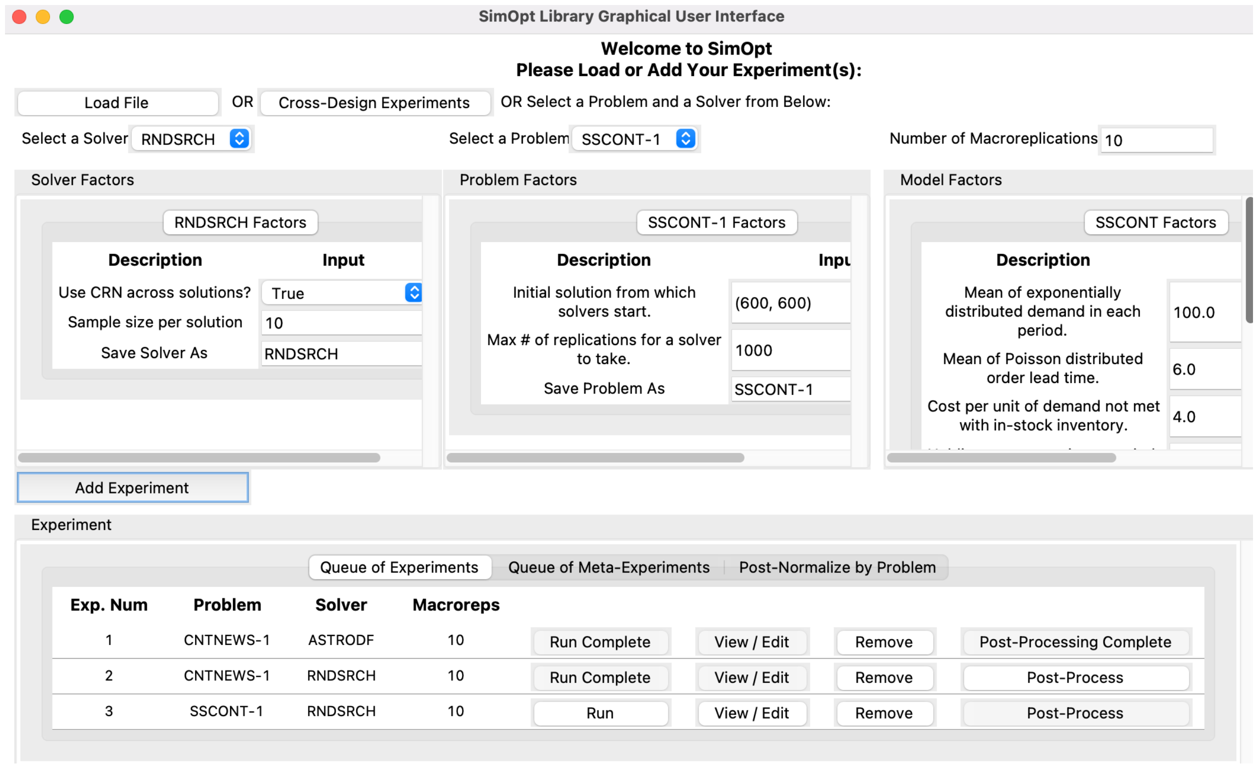
The functions that run data-farming experiments call Ruby functions to produce a design over the factors, e.g., a nearly orthogonal Latin hypercube (NOLH) design. For this code to run properly, the user must first install a version of Ruby on their computer that can be executed from the command line and additionally install the `datafarming` gem using a command like `gem install datafarming -v 1.3.0`. A data-farming experiment produces a table showing the factors describing each design point and the associated observed responses from each replication. These outputs are saved in a .csv file, which can be imported into the user's preferred statistical software package for further analysis.

7.4 GUI

SimOpt also has a graphical user interface (GUI) that aids the user in running SO experiments (a GUI for data farming is under development). The GUI source code depends on the following libraries: `Python 3`, `numpy`, `scipy`, `matplotlib`, `Pillow`, and `tkinter`. The GUI is opened by navigating to the `simopt/simopt` directory and running the `GUI.py` script from the terminal; i.e., executing the command `python3 GUI.py` where the syntax `python3` may vary from system to system. A README file on the GitHub repository provides a step-by-step user guide for all GUI activities including:

1. adding/loading experiments,
2. running/post-processing experiments,
3. post-normalizing experiments that share the same problem, and
4. producing plots.

Figure 2 shows the main form in which all activities are initiated. The user can add a new experiment by choosing the problem and solver with the option of modifying any of their factors, changing their instance names, and modifying the default number of macroreplications. Pressing the "Add Experiment" button will add the experiment to a list under "Queue of Experiments." Alternatively, the user can load an experiment previously saved in a .pickle file. Or several experiments can be created at once with the "Cross-Design Experiment" button and selecting compatible problems and solvers; a new grouping of experiments will


SimOpt Library Graphical User Interface

Welcome to SimOpt
Please Load or Add Your Experiment(s):

Load File OR Cross-Design Experiments OR Select a Problem and a Solver from Below:

Select a Solver: RNDSRCH — Select a Problem: SSCONT-1 — Number of Macroreplications: 10

Solver Factors — RNDSRCH Factors

Description	Input
Use CRN across solutions?	True
Sample size per solution	10
Save Solver As	RNDSRCH

Problem Factors — SSCONT-1 Factors

Description	Inpu
Initial solution from which solvers start.	(600, 600)
Max # of replications for a solver to take.	1000
Save Problem As	SSCONT-1

Model Factors — SSCONT Factors

Description	
Mean of exponentially distributed demand in each period.	100.0
Mean of Poisson distributed order lead time.	6.0
Cost per unit of demand not met with in-stock inventory.	4.0

Add Experiment

Experiment — Queue of Experiments | Queue of Meta-Experiments | Post-Normalize by Problem

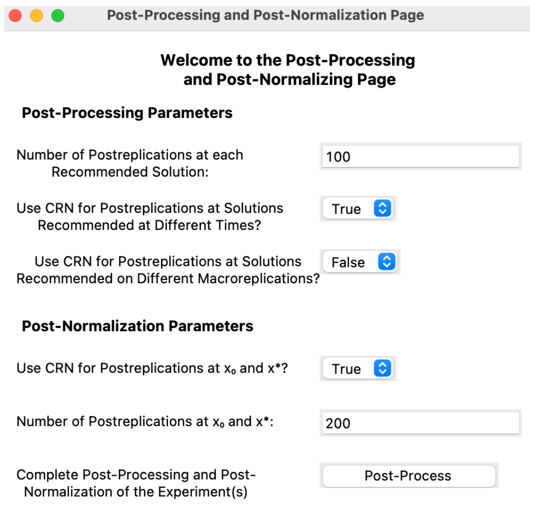
Exp. Num	Problem	Solver	Macroreps				
1	CNTNEWS-1	ASTRODF	10	Run Complete	View / Edit	Remove	Post-Processing Complete
2	CNTNEWS-1	RNDSRCH	10	Run Complete	View / Edit	Remove	Post-Process
3	SSCONT-1	RNDSRCH	10	Run	View / Edit	Remove	Post-Process

Figure 2: The main form in the SimOpt GUI.

appear under "Queue of Meta-Experiments." The user can view and edit any experiment before pressing the "Run" or "Post-Process" buttons. Running the experiments produces sequences of recommended solutions, and post-processing them yields the estimated objective function values at each solution using the default or user-modified CRN settings and number of postreplications (see Fig. 3a). All the post-processed experiments will appear under the tab "Post-Normalize by Problem." Here the user can select multiple experiments, as long as they share the same problem, and press the "Post-Normalize Selected" button. A pop-up form will appear that allows the user to modify the reference solutions and the number of postreplications for these solutions (see Fig. 3a). After post-normalization, the user is directed to the plotting pop-up form (see Fig. 3b). In this form, the user can select the post-normalized problems and solvers and the plot type of interest with the option of changing the plot's parameters or settings. Pressing the "Add" button will generate and save the new plot and list it under "Experiments to Plots." The user can view each plot individually or all in one page through the GUI.

7.5 Contributing Code

Users can contribute models, problems, and solvers to the library. To help ensure that contributed code properly interfaces with the current architecture, we recommend that users copy and modify code from similar models, problems, and solvers already present in the library. The demo scripts mentioned in Section 7.3 can also be used to help develop and debug contributed code.



Post-Processing and Post-Normalization Page

Welcome to the Post-Processing and Post-Normalizing Page

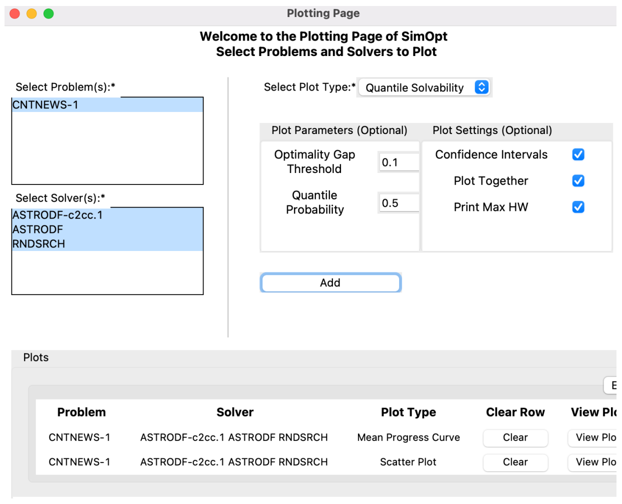
Post-Processing Parameters

Number of Postreplications at each Recommended Solution:	100
Use CRN for Postreplications at Solutions Recommended at Different Times?	True
Use CRN for Postreplications at Solutions Recommended on Different Macroreplications?	False

Post-Normalization Parameters

Use CRN for Postreplications at x_0 and x*?	True
Number of Postreplications at x_0 and x*:	200
Complete Post-Processing and Post-Normalization of the Experiment(s)	Post-Process

(a) Post-Processing Page



Plotting Page

Welcome to the Plotting Page of SimOpt
Select Problems and Solvers to Plot

Select Problem(s):* CNTNEWS-1

Select Solver(s):* ASTRODF-c2cc.1, ASTRODF, RNDSRCH

Select Plot Type:* Quantile Solvability

Plot Parameters (Optional)

Optimality Gap Threshold	0.1
Quantile Probability	0.5

Plot Settings (Optional)

Confidence Intervals	✓
Plot Together	✓
Print Max HW	✓

Add

Plots

Problem	Solver	Plot Type	Clear Row	View Pl
CNTNEWS-1	ASTRODF-c2cc.1 ASTRODF RNDSRCH	Mean Progress Curve	Clear	View Plo
CNTNEWS-1	ASTRODF-c2cc.1 ASTRODF RNDSRCH	Scatter Plot	Clear	View Plo

(b) Plotting Page

Figure 3: Two of the pop-up forms in the SimOpt GUI.

8 Conclusion

We present the latest version of the SimOpt testbed for SO and data-farming experiments. The transition to Python and top-to-bottom redesign are big steps toward making SimOpt the valuable resource for researchers and educators that we aspire for it to be. As with any active open-source project, SimOpt will continue to evolve as new experimental capabilities are added and community members contribute. This paper lays out formative principles of SimOpt's design that we expect will persist for years to come. Specifically, the versatility achieved through ascribing factors of models, problems, and solvers and the careful control of pseudorandom numbers sets SimOpt apart from conventional code implementations of SO solvers and problems and past versions of SimOpt.

Our near-term objective is to quickly populate the library with many problems and solvers that reflect the diversity of the SO field. We greatly welcome contributions; these can submitted through pull requests to the GitHub repository or correspondence with the development team. Python implementations of models and solvers are more easily integrated with the existing architecture, but we will explore opportunities to "wrap" models and solvers written in other languages. After the library has reached a critical mass of problems, one could imagine holding a competition to determine which solvers have best-in-class finite-time performance.

The next phase of SimOpt's development will aim to further enhance its capabilities. Under the current design, problem-solver pairings (and macroreplications thereof) are readily parallelized, but we have not yet enabled experiments to be run in such a fashion. We plan to develop the infrastructure for generating random problem instances by randomly generating model and problem factors from specified distributions. We are also working to facilitate parameter tuning and sensitivity analysis by allowing for more elaborate data-farming designs formed over model, problem, and solver factors. Lastly, we intend to support the computation of stochastic gradients of performance measures, when available, either via analytical derivation (e.g., infinitesimal perturbation analysis (IPA) Glasserman, 1991) or automatic differentiation software.

Acknowledgments

This work was partially supported by National Science Foundation grant CMMI-2035086. We thank Nicole Colberg, Matthew Ford, Yunsoo Ha, Zack Horton, Pranav Jain, Suraj Ponnaganti, Anita Shi, Joe Ye, Eva Zhang, and Jody Zhu for help with coding. We also thank Susan Sanchez for her suggestion to incorporate data farming into the SimOpt redesign.

References

K. Cooper and S. R. Hunter. PyMOSO. `https://github.com/pymoso/PyMOSO`, 2021.

Kyle Cooper and Susan R. Hunter. PyMOSO: Software for multiobjective simulation optimization with R-PERLE and R-MinRLE. *INFORMS Journal on Computing*, 32(4): 1101–1108, 2020.

D. J. Eckman, S. G. Henderson, S. Shashaani, and R. Pasupathy. SimOpt. `https://github.com/simopt-admin/simopt`, 2021a.

David J. Eckman, Shane G. Henderson, and Raghu Pasupathy. Redesigning a testbed of simulation-optimization problems and solvers for experimental comparisons. In N. Mustafee, K.-H. G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, editors, *Proceedings of the 2019 Winter Simulation Conference*, pages 3457–3467, Piscataway, New Jersey, 2019. Institute of Electrical and Electronics Engineers, Inc.

David J. Eckman, Shane G. Henderson, and Sara Shashaani. Evaluating and comparing simulation-optimization algorithms. Under review, 2021b.

P. Glasserman. *Gradient Estimation Via Perturbation Analysis.* Kluwer, The Netherlands, 1991.

W. D. Kelton. Implementing representations of uncertainty. In S. G. Henderson and B. L. Nelson, editors, *Simulation*, volume 13 of *Handbooks in Operations Research and Management Science*, chapter 20, pages 617–631. Elsevier, North Holland, 2006.

Pierre L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999.

Pierre L'Ecuyer, Richard Simard, E Jack Chen, and W David Kelton. An object-oriented random number package with many long streams and substreams. *Operations Research*, 50(6):1073–1075, 2002.

Jorge J Moré and Stefan M Wild. Benchmarking derivative-free optimization algorithms. *SIAM Journal on Optimization*, 20(1):172–191, 2009.

Raghu Pasupathy and Shane G. Henderson. A testbed of simulation-optimization problems. In L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, editors, *Proceedings of the 2006 Winter Simulation Conference*, pages 255–263, Piscataway, New Jersey, 2006. Institute of Electrical and Electronics Engineers, Inc.

Susan M. Sanchez. Data farming: Methods for the present, opportunities for the future. *ACM Transactions on Modeling and Computer Simulation*, 30(4):Article 22. 1–30, 2020.