




SimOpt: A Testbed for Simulation-Optimization Experiments

David J. Eckman,^{a,*} Shane G. Henderson,^b Sara Shashaani^c

^aWm Michael Barnes '64 Department of Industrial and Systems Engineering, Texas A&M University, College Station, Texas 77843;

^bSchool of Operations Research and Information Engineering, Cornell University, Ithaca, New York 14853; ^cEdward P. Fitts Department of Industrial and Systems Engineering, North Carolina State University, Raleigh, North Carolina 27695


*Corresponding author

Contact: eckman@tamu.edu,  <https://orcid.org/0000-0002-6473-6434> (DJE); sgh9@cornell.edu,  <https://orcid.org/0000-0003-1004-4034> (SGH); sshasha2@ncsu.edu,  <https://orcid.org/0000-0001-8515-5877> (SS)

Received: January 7, 2022

Revised: August 25, 2022; December 21, 2022

Accepted: January 8, 2023

Published Online in Articles in Advance:  2023

<https://doi.org/10.1287/ijoc.2023.1273>

Copyright: © 2023 INFORMS

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; uses an implementation of the MRG32k3a random number generator that supports streams, substreams, and subsubstreams; supports the automated use of common random numbers for ease and 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 and as a Python package.

History: Accepted by Ted Ralphs, Area Editor for Software Tools.

Funding: This work was supported by the National Science Foundation [Grant CMMI-2035086].

Supplemental Material: The software that supports the findings of this study is available within the paper and its Supplemental Information (<https://pubsonline.informs.org/doi/suppl/10.1287/ijoc.2023.1273>) as well as from the IJOC GitHub software repository (<https://github.com/INFORMSJoC/2022.0011>) at (<http://dx.doi.org/10.5281/zenodo.7468744>).

Keywords: simulation optimization • solvers • experimental design • common random numbers

1. Introduction and Motivation

A simulation-optimization (SO) problem is an optimization problem where the objective function or constraints are evaluated (approximately) through a stochastic simulation. A wide variety of problems can be formulated as SO problems, and although quite a few SO solvers exist, there is tremendous room for more development. SimOpt (Eckman et al. 2021) is a suite of SO problems and solvers that supports the testing and development of SO solvers, with many goals including highlighting the performance of solvers over practically relevant timescales, helping identify challenging problems that defy efficient solution with current solvers, facilitating solver comparisons, and providing a testbed to aid solver development and improvements (see Eckman et al. (2023) for a fuller discussion of these goals). 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).

SimOpt has not yet achieved its primary goal of broad adoption by SO researchers. Certainly, it has been used to a modest extent by researchers seeking test problems, as evidenced by personal communications

received by the authors, but widespread adoption of its solver-comparison capabilities has remained elusive. We believe the reasons are four-fold. First, the earliest instantiation of SimOpt did not have standardized problem interfaces, making it difficult to test a range of problems with a solver. That was resolved fairly recently (Eckman et al. 2019), with Dong et al. (2017) showcasing the potential unlocked by that standardization. Second, the use of the proprietary software MATLAB limited accessibility, which we resolve here by using Python. Third, the class of problems was not rich, which we have partially addressed in this new version through the separation of *models* from *problems*, so that now many problems can be built from the same simulation model. In addition, we can vary *factors* to create families of similar problems from a single base problem. Fourth, the set of people contributing to SimOpt was smaller than it is today; some initial momentum was needed, and we believe we now have that momentum.

This paper introduces a comprehensive reimagining 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 the following:

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 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, for example, 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.

This paper focuses on the *use* of SimOpt, highlighting those aspects of its design that are most important for a range of use cases from a casual SO user to researchers working on designing improved solvers. Eckman et al. (2023) define and explain the rationale behind the diagnostic tools that are implemented in SimOpt. The papers overlap only slightly: Section 3 summarizes the metrics developed in Eckman et al. (2023). Otherwise, the papers are mostly complementary.

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. Although PyMOSO has the ability to work with the single-objective SO problems that are our focus, it is primarily intended for multiobjective problems. Conversely, SimOpt latently supports multiobjective 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, for example, 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 currently 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. This paper is accompanied by a GitHub repository (Eckman et al. 2022) containing the source code for SimOpt.

2. SO Problems and Solvers

The prototypical SO problem we consider is

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

In this formulation, 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. The vector x of decision variables takes values in a domain $\mathcal{D}(w)$ that could, for example, restrict x to be integer-valued and could depend on w . 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, for example, 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 both the feasible region and the optimal solution of the problem.

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 either through the function h or 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, for example, 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 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 nonvacuous then we say the problem has stochastic constraints.

Relative to the taxonomy of constraints developed in Digabel and Wild (2015), we primarily assume Q**K constraints, in that both $g(\cdot, \cdot)$ and $h(\cdot, \cdot)$ are Quantifiable and Known, but depending on the problem setting one may or may not be able to relax (the first *) these constraints. Within the structure of that taxonomy, the $h(\cdot, \cdot)$ constraints are a priori constraints, whereas the $g(\cdot, \cdot)$ constraints are Simulation constraints (explaining the second *). The domain $\mathcal{D}(w)$ could permit deviations from the Q**K classification, although we prefer quantifiable, known constraints because we believe them to be more tractable.

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 $i = 1, 2, \dots, n$ 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 , $\kappa := (\kappa_1, \kappa_2, \dots, \kappa_n)$, μ , and Σ , demonstrating that factors can be scalars, vectors, or 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, for example,

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

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

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 $c = (c_1, c_2, \dots, c_n)$, 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, that is, $x = \kappa$, while treating the other factors as fixed, that is, $w = \{n, \mu, \Sigma, c, b, \epsilon\}$.

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, for example, 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. 2023).

In SimOpt, problems are solved by *solvers*, which are implementations of SO algorithms. 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.

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. Problems and solvers in the library are categorized using a four-letter-abbreviation coding system detailed in Table 1; for

Table 1. Abbreviations Used to Categorize Problems and Solvers to Recognize Their Compatibility

| Objective | Constraint | Variable | Direct gradient observations |
|--------------|-------------------|----------------|------------------------------|
| Single (S) | Unconstrained (U) | Discrete (D) | Available (G) |
| Multiple (M) | Box (B) | Continuous (C) | Not Available (N) |
| | Deterministic (D) | Mixed (M) | |
| | Stochastic (S) | | |

example, SBCG encodes single-objective, box-constrained, continuous problems with direct gradient estimates. Solver classes represented in the library include random/direct search, model-based search, simplex/pattern-based search, and gradient-search. For a complete list of problems and solvers, see <https://simopt.readthedocs.io/en/latest>.

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 (Ford et al. 2022), although those estimators will exhibit problem-specific degrees of bias (Eckman and Henderson 2020).

SimOpt currently only supports *fixed-budget* solvers, that is, solvers that are constrained to use up to a fixed number of simulation replications over the entire course of the search for optimal solutions. SimOpt is not designed to directly support *fixed-precision* solvers, where the search continues until a stopping condition is met so that the expended budget is random. Perhaps the most prominent class of these yet unsupported solvers are ranking-and-selection algorithms that enumerate a finite list of potential solutions and provide a statistical guarantee on the selected system. Some solvers, that we call *budget-specific* solvers, explicitly use knowledge of the overall budget of simulation replications in setting key parameters (Nemirovski et al. 2009), whereas *budget-agnostic* solvers do not. Most SimOpt metrics are intended for budget-agnostic solvers, although SimOpt provides additional *terminal* plots to enable the comparison of budget-specific solvers and budget-agnostic solvers (see Section 3).

Some solvers require nontrivial computing overhead beyond that needed to generate simulation replications. Such overhead does not appear in the metrics that SimOpt produces, although the overall computing time for each macroreplication is logged and can be accessed (see Section 3).

Parallelization is not yet supported but is planned through the parallel execution of macroreplications. Solvers can exploit parallel computing, but the metrics that SimOpt produces are most easily interpreted in the context of a serial model of computation.

The metrics that SimOpt produces are not customized in any way for SO problems with stochastic constraints, although this is planned. Multiobjective SO problems are not supported, but some preparations have been made for future extensibility.

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 or stochastic constraints, as well as any intrinsic randomness of the solver, for example, 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 m th macroreplication generates a stochastic process $\{X_m^{p,s}(t) : 0 \leq t \leq 1\}$, 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 postprocessing stage to obtain unbiased objective function estimates and the results can be 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 here.

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

Step 2. Take N independent postreplications of the model at each distinct recommended solution in $\{X_m(t) : 0 \leq t \leq 1\}$ for $m = 1, 2, \dots, M$. 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. Optionally, 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:

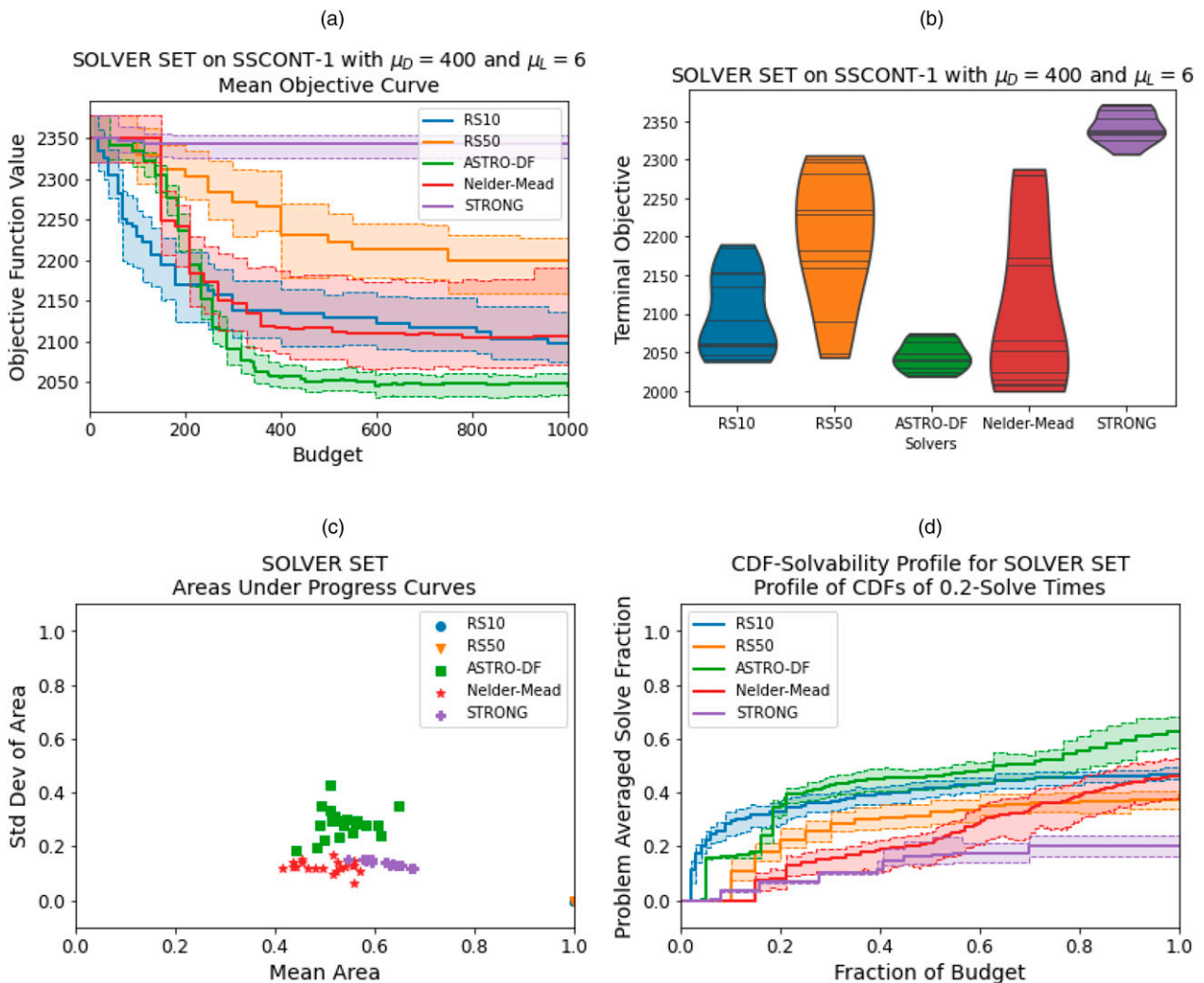
$$v_m(t) = \frac{f_N(X_m(t)) - f_L(x_0)}{f_L(x^*) - f_L(x_0)}.$$

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 $v_m(t)$ the estimated

progress at time t on macroreplication m and we call $v_m(\cdot)$ the estimated *progress curve*. A progress curve typically takes values between zero and one, although 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. We summarize five types of comparative plots introduced in Eckman et al. (2023) here and show four examples in Figure 1. The first two plots in Figure 1 measure the performance of multiple solvers on a single

Figure 1. (Color online) Plots Produced by SimOpt for Comparing Solvers on One or More Problems



Notes. (a) Mean progress curves. (b) Terminal progress violin plots. (c) Area-under-progress-curve scatter plots. (d) Solvability profiles.

problem and the final two plots depict the performance of multiple solvers on multiple problems. SimOpt uses a two-level bootstrapping procedure to obtain error estimates for these metrics, as outlined in appendix B of Eckman et al. (2023). Pointwise 95% bootstrap confidence intervals are indicated in the plots with shading.

- *Aggregated Progress Curves.* The estimated progress curves $v_1(\cdot), v_2(\cdot), \dots, v_M(\cdot)$ 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 zero 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 $v_m(\cdot)$ is referred to as the 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.

- *Terminal Progress Comparative Violin Plots and Scatter Plots.* The progress of solvers once the budget T is exhausted is of particular interest, partly to enable the direct comparison of budget-specific and budget-agnostic solvers. Violin plots, one per solver, depict the distribution of the terminal progress $v(1)$, whereas the mean and standard deviation of $v(1)$ for each problem-solver pair can be plotted in a scatter plot.

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 multiobjective SO problems, which have their own unique aspects when it comes to measuring a solver's performance. Third, they do not show the computational effort a solver requires beyond that needed to run simulation replications, which can be considerable for some solvers. Presently, we log the computation times for each macroreplication of each problem-solver pair, which can be analyzed manually; we do not provide diagnostic plots for these computation times partly because they depend so heavily on the computing platform that is used. Last, the computational effort required to construct estimated progress curves, and associated metrics, grows with the number of solutions recommended by a solver. In these cases, estimating the progress on a regular grid of times can alleviate this burden while still conveying the general performance of a solver; this alternative construction is not yet implemented.

4. Use Cases

The performance metrics and plots discussed previously 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 and these are easily produced in SimOpt by simply setting an option in the call to generate the progress curves.

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 . Because 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, s , compares with another benchmark solver, s' , 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 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, \dots, v_r , say, then one would use the solver set $\mathcal{S} = \{s(v_1), s(v_2), \dots, s(v_r)\}$.

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 $\mathcal{P} = \{p(w_1), p(w_2), \dots, p(w_r)\}$, where w_1, w_2, \dots, 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 vs. another, it would be appropriate to perform an experiment with a comprehensive problem set $\mathcal{P} = \{p_1, p_2, \dots, p_r\}$ 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 the plots and log files. 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 $\mathcal{V} = \{v_1, v_2, \dots, v_r\}$. 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 be 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, that is, a multivalued 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 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 further 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 any stochastic constraints, as posed in Section 2. The individual observations of the objective function (as well as 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, that is, 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, 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 postprocessed as described in Section 3. Furthermore, one can postnormalize results from `ProblemSolver` objects corresponding to multiple solvers run on the same problem. The specifics of the postprocessing and postnormalization stages, namely the number of postreplications and the use of CRN, are recorded to the `ProblemSolver` object for future reference when bootstrapping. After postnormalization, 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 problems and a list of solvers 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 postreplicated and postnormalized 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 postprocess experiments on multiple problems and solvers. This design enables the user to activate common random numbers (CRN) at various levels. For background on CRN, see chapter 11 of Law (2015), and for insight into the value of streams and substreams, see Kelton (2006).

Table 2. Summary of CRN Management and User Control

| Stage | Form of CRN | Default | Controllable |
|---|-----------------------------|---------|--------------|
| Running (optimization) | Across solutions | ✓ | ✓ |
| | Across problem-solver pairs | ✓ | |
| Postprocessing/bootstrapping (evaluation) | Across solutions | ✓ | ✓ |
| | Across macroreplications | | ✓ |
| | Between x_0 and x^* | ✓ | ✓ |
| | Across problem-solver pairs | ✓ | |

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} . Our implementation is separately packaged for use outside of SimOpt and can be downloaded at <https://pypi.org/project/mrg32k3a> or installed from the terminal using the command `pip install mrg32k3a`.

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 one, but in Python, indexing starts at zero.) By creating multiple `MRG32k3a` objects with different seeds, we control how random numbers are generated. The schema is repeated for each problem-solver pair, that is, 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 $(s, ss, sss) = (m, i, r)$, where r is the replication number of the solution being visited by the solver (during the optimization) and $i = 1, 2, \dots, I$ 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 10th 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. In particular, 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 2 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, after 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, for example, antithetic variates and stratified sampling.

6.2.1. 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.

6.2.2. 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 using 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.

6.2.3. 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, and so on, 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 Postprocessing Experiments

Postprocessing entails re-evaluating the collection of recommended solutions returned by each macroreplication of each problem-solver pair. A dedicated stream for postprocessing, signified by “P,” is used for generating random numbers within the model when simulating postreplications—the solver is not involved. As a consequence of working with a single stream, the substream level must now accommodate indexing over both macroreplications and sources of randomness, for example, postprocessing 50 macroreplications and five 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 postprocessing stage is $(s, ss, sss) = (P, I \times (m - 1) + i, n)$, where n is the postreplication number. For example, in

the simple queueing model, $(P, 3, 1)$ and $(P, 4, 1)$ represent the sequence of random numbers used for the arrivals and service times, respectively, in the postprocessing of the solutions on the second macroreplication.

6.3.1. 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 the performance of recommended solutions.

6.3.2. 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.

6.3.3. CRN Between Postreplications at x_0 and x^* . A postnormalization 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.

6.3.4. CRN Across Problem-Solver Pairs. As in the schema for running experiments, all problem-solver pairs are postprocessed using the same set of random numbers. This again is for convenience, not variance reduction, because the postprocessing 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 postprocessing stage, but the random numbers come from yet another dedicated stream, signified by “B.” The default random-number schema used in bootstrapping is $(s, ss, sss) = (B, b, j)$, 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. 2021). The `master` branch contains the Python version discussed in this paper; a separate branch contains the deprecated MATLAB code. 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, for example, `commit 86cd5fdbf610f6cd9b20564d974a734a29e7bfa9`. 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.

The preferred 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. After forking the repository, users should *clone* it to their personal computer and open the root directory within their preferred integrated development environment.

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.

SimOpt is also available as a package named `simoptlib` at <https://pypi.org/project/simoptlib> and can be installed from the terminal using the command `pip install simoptlib`. (The `mrq32k3a` package will be automatically installed when installing `simoptlib`.) Users who take this approach can then directly import models, problems, and solvers from the library within the Python environment, for example, `from simopt.models.cntnv import CntNV` imports the class for the continuous newsvendor model.

This option may be more appealing for educational purposes, where users wish to experiment with the library, but not contribute.

Users who have either forked the repository or installed the `simoptlib` package can then conduct experiments by running scripts from the command line or using the graphical user interface (GUI).

7.2. Scripts

The module `experiment_base.py` contains high-level functions for running and postprocessing SO experiments and plotting the results. Likewise, the module `data_farming_base.py` defines functions and classes for data-farming experiments that involve varying factors of the models. Although users can read the documentation for these functions and directly call them from within the Python environment, the `demo` folder in the GitHub repository 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 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. (Users who have installed the `simoptlib` package may download the scripts from the repository and modify them as needed.)

- `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, gradients (if available), and left-hand sides of stochastic constraints.
- `demo_problem_solver.py`: Run multiple macroreplications of a solver on a problem, save the outputs to a `.pickle` file in the `experiments/outputs` folder, save a log `.txt` file in the `experiments/logs` folder, and save plots of the results to `.png` files in the `experiments/plots` folder.
- `demo_problems_solvers.py`: Run multiple macroreplications of multiple solvers on multiple problems and save the results.
- `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_san-sscont-ironore-cont_experiment.py` demonstrates how experiments with multiple solvers and multiple problems were run to produce the plots in Figure 1. These experiments took about two hours to run on a standard laptop.

The functions that run data-farming experiments call Ruby functions to produce a design over the factors, for example, 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`. A data-farming experiment produces a table showing the model 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.

7.3. GUI

SimOpt has a GUI that aids the user in running SO experiments—a GUI for data farming is under development. The GUI is opened by executing the command `python3 -m simopt.GUI` from the terminal, where the syntax `python3` may vary from system to system. If cloning the repository, the previous command should be run after navigating to the root directory. A README file on the GitHub repository provides a step-by-step user guide for all GUI activities including the following:

1. Adding/loading problem-solver pairs or groups,
2. Running/postprocessing problem-solver pairs or groups,
3. Postnormalizing problem-solver pairs that share the same problem, and
4. Producing plots of problem-solver pairs and groups with customizable settings.

Figure 2 shows the main form in which all activities are initiated. New problem-solver pairs can be added

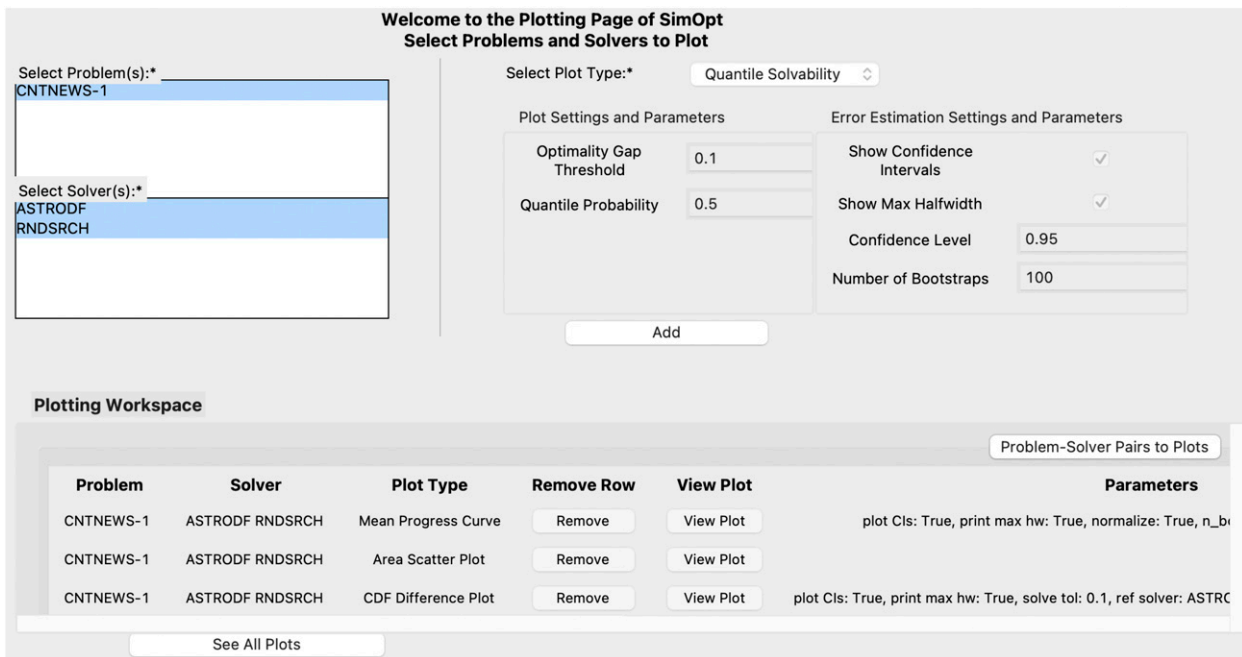
by choosing the problem and solver, with the option of modifying any of their factors, instance names or number of macroreplications. The “Add Problem-Solver Pair” button adds the pair to a list under “Queue of Problem-Solver Pairs.” One can also load a problem-solver pair previously saved in a `.pickle` file. Alternatively, several problem-solver pairs can be grouped together either using the “Create a Problem-Solver Group” button and selecting compatible problems and solvers (with their default factors) or by first creating and selecting several problem-solver pairs (with their customized factors) before pushing “Create a Problem-Solver Group from Selected.” This approach completes the cross design of all problems and solvers in the selected pairs.

Once a problem-solver group is created, it appears in the “Queue of Problem-Solver Groups” tab. Pairs and groups can be viewed and edited before pressing the “Run” or “Post-Process” buttons. Running a problem-solver pair produces macroreplications and thus sequences of recommended solutions. Postprocessing the solutions yields the estimated objective function values at each solution using the default or user-modified CRN settings. All the postprocessed problem-solver pairs appear under the tab “Post-Normalize by Problem.” One can select multiple problem-solver pairs, as long as they share the same problem, and press the “Post-Normalize Selected” button. One can optionally modify the reference solution and the number of postreplications for these solutions. For problem-solver

Figure 2. Main Form in the SimOpt GUI



Figure 3. (Color online) Plotting Form in the SimOpt GUI



groups, the “Post-Process and Post-Normalize” button performs both procedures for each pair in the group with one click, using the user-specified number of macroreplications for all pairs. After completing postnormalization for either problem-solver pairs or problem-solver groups, the user can generate plots (Figure 3). One can select post-normalized problems and solvers and the plots of interest, with the option of changing plot parameters and error-estimation parameters. Pressing the “Add” button generates and saves the new plot and lists it under “Problem-Solver Pairs to Plots.” One can view each plot individually or all in one page through the GUI.

7.4. 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 7.2 can also be used to help develop and debug contributed code.

For additional support, SimOpt 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> 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.

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 we aspire to provide. 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 be 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. Last, 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 (Ford et al. 2022).

Acknowledgments

The authors thank the associate editor and reviewers for feedback that helped improve the paper and software; Kyle Beck, Nolan Berry, Noah Bigler, Nicole Colberg, Rina Davila, Lilibeth Escamilla, Matthew Ford, Yunsoo Ha, Zack Horton, Pranav Jain, Natalia Londono, Suraj Ponnaganti, Patrick Rangel, Anita Shi, Joe Ye, Eva Zhang, Mark Zhang, and Jody Zhu for help with coding; Susan Sanchez for a suggestion to incorporate data farming into the SimOpt redesign; and Pierre L’Ecuyer for advice on random number management.

References

- Cooper K, Hunter SR (2020) PyMOSO: Software for multiobjective simulation optimization with R-PERLE and R-MinRLE. *INFORMS J. Comput.* 32(4):1101–1108.
- Cooper K, Hunter SR (2021) PyMOSO. Accessed March 3, 2021, <https://github.com/pymoso/PyMOSO>.
- Digabel SL, Wild SM (2015) A taxonomy of constraints in simulation-based optimization. Preprint, submitted May 28, <https://arxiv.org/abs/1505.07881>.
- Dong N, Eckman DJ, Zhao X, Poloczek M, Henderson SG (2017) Empirically comparing the finite-time performance of simulation-optimization algorithms. Chan WKV, D’Ambrogio A, Zacharewicz G, Mustafee N, Wainer G, Page E, eds. *Proc. Winter Simulation Conf.* (IEEE, Piscataway, NJ), 2206–2217.
- Eckman DJ, Henderson SG (2020) Biased gradient estimators in simulation optimization. Bae KH, Feng B, Kim S, Lazarova-Molnar S, Zheng Z, Roeder T, Thiesing R, eds. *Proc. Winter Simulation Conf.* (IEEE, Piscataway NJ), 2935–2946.
- Eckman DJ, Henderson SG, Pasupathy R (2019) Redesigning a testbed of simulation-optimization problems and solvers for experimental comparisons. Mustafee N, Bae KHG, Lazarova-Molnar S, Rabe M, Szabo C, Haas P, Son YJ, eds. *Proc. Winter Simulation Conf.* (IEEE, Piscataway, NJ), 3457–3467.
- Eckman DJ, Henderson SG, Shashaani S (2022) SimOpt: A testbed for simulation-optimization experiments v2022.0011. Accessed December 21, 2022, <https://github.com/INFORMSJoC/2022.0011>.
- Eckman DJ, Henderson SG, Shashaani S (2023) Diagnostic tools for evaluating and comparing simulation-optimization algorithms. *Articles in Advance at INFORMS. J. Comput.*, ePub ahead of print January 5, <https://pubsonline.informs.org/doi/10.1287/ijoc.2022.1261>.
- Eckman DJ, Henderson SG, Shashaani S, Pasupathy R (2021) SimOpt. Accessed March 3, 2021, <https://github.com/simopt-admin/simopt>.
- Ford MT, Eckman DJ, Henderson SG (2022) Automatic differentiation for gradient estimators in simulation. Feng B, Pedrielli G, Peng Y, Shashaani S, Song E, Corlu CG, Lee LH, Chew EP, Roeder T, Lendermann P, eds. *Proc. Winter Simulation Conference* (IEEE, Piscataway NJ), 3134–3145.
- Glasserman P (1991) *Gradient Estimation Via Perturbation Analysis* (Kluwer).
- Kelton WD (2006) Implementing representations of uncertainty. Henderson SG, Nelson BL, eds. *Simulation*, vol. 13 of *Handbooks in Operations Research and Management Science* (Elsevier, Amsterdam), 181–191.
- Law AM (2015) *Simulation Modeling and Analysis*, 5th ed. (McGraw-Hill, New York).
- L’Ecuyer P (1999) Good parameters and implementations for combined multiple recursive random number generators. *Oper. Res.* 47(1):159–164.
- L’Ecuyer P, Simard R, Chen EJ, Kelton WD (2002) An object-oriented random number package with many long streams and substreams. *Oper. Res.* 50(6):1073–1075.
- Moré JJ, Wild SM (2009) Benchmarking derivative-free optimization algorithms. *SIAM J. Optim.* 20(1):172–191.
- Nemirovski A, Juditsky A, Lan G, Shapiro A (2009) Robust stochastic approximation approach to stochastic programming. *SIAM J. Optim.* 19(4):1574–1609.
- Pasupathy R, Henderson SG (2006) A testbed of simulation-optimization problems. Perrone LF, Wieland FP, Liu J, Lawson BG, Nicol DM, Fujimoto RM, eds. *Proc. Winter Simulation Conf.* (IEEE, Piscataway, NJ), 255–263.
- Sanchez SM (2020) Data farming: Methods for the present, opportunities for the future. *ACM Trans. Modeling Comput. Simulation* 22:1–30.