Proceedings of the 2011 Winter Simulation Conference
S. Jain, R. R. Creasey, J. Himmelspach, K. P. White, and M. Fu, eds.

SIMOPT: A LIBRARY OF SIMULATION OPTIMIZATION PROBLEMS

Raghu Pasupathy
Industrial and Systems Engineering
Virginia Tech
Blacksburg, VA 24061, USA

Shane G. Henderson
Operations Research and Information Engineering
Cornell University
Ithaca, NY 14853, USA

ABSTRACT

We present SimOpt — a library of simulation-optimization problems intended to spur development and comparison of simulation-optimization methods and algorithms. The library currently has over 50 problems that are tagged by important problem attributes such as type of decision variables, and nature of constraints. Approximately half of the problems in the library come with a downloadable simulation oracle that follows a standardized calling protocol. We also propose the idea of problem and algorithm wrappers with a view toward facilitating assessment and comparison of simulation optimization algorithms.

1 INTRODUCTION

Simulation optimization (SO) is the practice of finding a minimizer or maximizer of a real-valued function that can only be estimated through stochastic simulation. The appeal of SO is that it allows one to work with essentially arbitrarily complex simulation models, freeing the modeler from the tyranny of restricting model complexity to mathematically tractable forms. The problem of course is that complex models can lead to complex optimization problems. Not only must SO algorithms deal with the estimation error associated with stochastic simulation, but SO also inherits all of the challenges associated with optimizing deterministic functions, including nonlinearity, nondifferentiability, noncontinuity, and so forth.

These challenges have led to the development of a number of SO algorithms that can tackle very general problems, but lack practical guarantees of performance. One often sees results in the SO literature on algorithms that guarantee asymptotic convergence to a local or global optimizer. One also occasionally sees convergence *rate* results that offer hope with regard to practical performance guarantees. Unfortunately, most of these convergence-rate results tend to be built around asymptotic analyses that apply once an algorithm is focusing on a neighbourhood of an optimal solution. While these results are welcome and worthwhile, the asymptotic regime in which they apply is often well beyond the reach of the computational budgets that are relevant in practice.

A library of SO problems could help address this issue. Realizing this, an appeal for an SO problem library was made (Fu et al. 2000, Fu 2002, Glynn 2002), and one was proposed in Pasupathy and Henderson (2006). In this paper we announce the arrival of the library (Pasupathy and Henderson 2011), explain how the library is organized, how problems are implemented in the library and how the library can be used to compare SO algorithms. Our goal is to stimulate *use of* and *contributions to* the library. Indeed, contributions are vital because the library is not a finished product, but rather a living repository that we hope will grow to fulfil its intended purposes, as detailed below.

The primary purpose of the library is to ensure more comprehensive numerical comparisons between SO algorithms. Such comparisons should

- focus attention on the performance of algorithms when applied with computational budgets that are practically relevant;

- stimulate the development of classes of SO problems that, in turn, stimulate algorithm development, just as there are now algorithms for deterministic optimization specialized to linear problems and subclasses thereof (e.g., transportation problems), smooth convex problems, nonsmooth convex problems, multimodular problems on integer domains;
- facilitate comparisons between algorithms to identify their strengths and weaknesses and stimulate their further development; and
- help promote the use of SO.

The library (Pasupathy and Henderson 2011) is organized as a list of problems together with their attributes. Attributes are aspects of the problems that may limit the types of algorithms that can be brought to bear on them, such as the nature of the variables and so forth. Each problem has its own wiki page containing a full description of the problem in pdf format, source code (if any) for the problem, and other comments or information about the problem. More details on library organization, including information on how you can contribute, may be found in Section 2 below.

Approximately half of the problems in the library have been coded. Thus far, all implementations are in MATLAB$^{\text{TM}}$. We chose this environment because of its wide availability and ease of coding, and also because it facilitates algorithm comparisons. This choice comes at the price of execution speed because MATLAB$^{\text{TM}}$is an interpreted language, but we believe that *accessibility* is more important than *computational efficiency* at this stage. Having said that, it is straightforward to add other versions of problem codes to the library, and so we remain open to other programming languages and environments. We provide more details on the standardized interface we have used for implementing SO problems in Section 3 below. Section 3 also discusses how random number streams are implemented, owing to the importance of this point in employing techniques like common random numbers in SO algorithms.

Given that the primary purpose of the library is to compare SO algorithms, we have developed a set of tools for assisting with such comparisons. Section 4.1 discusses these tools, and Section 4.2 gives an example of their use in comparing two simple algorithms. These two algorithms are really caricatures of algorithms that are included to clarify how comparisons can be undertaken, and are certainly not algorithms that we advocate for SO practice.

Our efforts in developing and promoting SimOpt continue, but the library will not flourish without the participation of the SO research community. Accordingly, we *request your contributions* in using, and contributing to, the library.

- Please upload your own SO test problems, with or without code at http://www.simopt.org/upload.php
- Please upload comments on your experiences with the various problems you tackle on the problem wiki pages.
- Please upload bug fixes, and alternative implementations in other languages.
- Please let us know of suggested improvements in implementations or interfaces.

2 LIBRARY ORGANIZATION

The library (Pasupathy and Henderson 2011) currently consists of a collection of test problems that are listed with their key attributes, described in detail in pdf files, and many are implemented in MATLAB$^{\text{TM}}$codes and discussed further in wiki pages.

2.1 Problem Attributes

The SO library is restricted to finite-dimensional problems. This means, for example, that a stochastic control problem that is parameterized by a dynamic-programming value function defined on a countably infinite, or uncountable state space is explicitly excluded. However, if one were to parameterize, with a finite number of parameters, a class of policies for that same stochastic control problem, then the parameterized version of the problem would not necessarily be excluded.

Within the class of finite-dimensional problems, problems are characterized by the following attributes, which are important in determining what algorithms may be appropriate for a given SO problem.

Variable Class An SO problem may have *continuous*, *integer-ordered*, or *categorical* decision variables. Continuous (decision) variables can take on any real value in an interval, e.g., the x-coordinate of a location in the plane. Integer-ordered variables take on integer values in an interval and the order of the integers has a physical meaning, e.g., the number of agents assigned to a particular shift. Categorical variables capture all other variable types that can take only a finite number of values, and is something of a "catchall," e.g., the sequence of nodes visited in a graph. Some problems contain a mix of these variable classes. In that case, the problem is characterized as categorical if *any* variables are categorical, as integer ordered if *any* variables are integer ordered and there are no categorical variables, and continuous otherwise.

Constraints Problems can be unconstrained, have deterministic constraints only, or have at least one stochastic constraint. Deterministic upper and lower bounds, e.g., $0 \leq x \leq 1$, or $0 \leq x < \infty$ are not considered to be constraints. Deterministic constraints are those that can be computed without the need for stochastic simulation. Stochastic constraints, then, are those that must be evaluated using stochastic simulation. For example, call center problems typically place lower bounds on the fraction of calls that are answered within a certain time threshold of being received. This fraction must be estimated within a simulation.

Code Available? We ultimately hope to make code available for all of the problems in the library, but for now only a subset of the problems have code available. We currently have a preference for MATLAB$^{\text{TM}}$code, but the library will certainly accept other languages and will even accept multiple implementations of problems in multiple languages.

2.2 Wiki Pages

Each SO problem has its own wiki page. The wiki page is broken down into sections:

About the Problem contains at minimum a link to a pdf file that describes the problem. Optional extras include links to source code and/or supplementary files (data files being common). The pdf description should be sufficient to completely specify the problem at the level of specifying any underlying stochastic processes.

Reported Results is the natural place to discuss algorithm performance, best solutions found thus far, and so forth. We want to generate discussion in the research community, so please do "show off" if you are particularly pleased about your algorithm's performance on the problem!

Related Problems is a section that describes other problems (not listed in the library) that are "similar" to the problem in question. This is intended to possibly give the user additional insight on problem structure and the nature of the solution.

Comments is a placeholder for general comments, including necessary bug fixes, minor issues with the problem statement and data, and any problem quirks.

References is a section that contains any external references from which the problem originates.

Contributing Content contains a link giving directions to the user on contributing content to the wiki. As noted, for each problem's wiki, volunteers will have the ability to contribute to the text. Contributions will be accepted after appropriate editing.

3 INTERFACES AND IMPLEMENTATION

In this section we explain the standardized MATLAB interface for each problem, along with a discussion about random number streams, owing to the importance of these in enabling common random numbers, independent replications, etc. As a running example, we use the problem "Ambulances in a square."

3.1 Problem Interface

The MATLAB implementation of every SO problem in the library requires two separate MATLAB functions. The first function specifies the problem structure, while the second implements the simulation model itself. These functions should be named UniqueProblemNameStructure and UniqueProblemName respectively. The standardized names are important, because they allow us to easily perform algorithm comparisons, as discussed in Section 4.1 below.

Before describing the two functions in more detail, we comment on problem constraints. Each constraint in a problem is assumed to be of the form $g(x) \geq 0$, where g is some function value that is either computed "exactly" (to numerical precision) or estimated in the simulation code. Problems with equality constraints can be exceedingly hard to handle numerically because the solution space of such problems often has an empty interior, so where possible we encourage the avoidance of equality constraints. For example, a common form of constraint is $x_1 + x_2 + \cdots + x_d = b$, where b is some budget parameter. In this case one can simply drop the dimension by one and exclude the constraint, since the value of x_d is implied by the other variables. If, in addition, x_d was required to be nonnegative, then after dropping the dimension by one we must also specify the constraint $b - x_1 - x_2 - \cdots - x_{d-1} \geq 0$.

The interface for the function specifying the problem structure is

```
function [minmax d m VarNature VarBds FnGradAvail NumConstraintGradAvail StartingSol budget ObjBd OptimalSol] =
  AmbulanceStructure(NumStartingSol, seed)
```

This "structure" function also generates any starting solution(s) required. Indeed, some algorithms, such as stochastic approximation, need a single starting solution, while others, such as genetic algorithms, require a family of solutions. The generation of such starting solutions is part of the problem description, and performed in the "structure" function. This explains the integer input parameter NumStartingSol (which can be zero). If the problem description requires random starting points, then the positive integer input parameter seed will be required to set the seed of the random number generator, as discussed in Section 3.2 below.

The structure function returns the following quantities.

- minmax gives $+1$ for maximization problems, and -1 for minimization problems.
- d gives the dimension (number of decision variables)
- m gives the number of constraints (0 for unconstrained problems)
- VarNature is a d dimensional column vector where the jth component gives the nature of the jth decision variable: continuous (0), integer-ordered (1) or categorical (2).
- VarBds is a d $\times 2$ matrix, the jth row of which gives lower and upper bounds on the jth variable. These bounds can be −inf, +inf or any real number for continuous or integer-ordered variables. Categorical variables are assumed to take integer values, including the lower and upper bound endpoints (assumed to be integer). Thus, a categorical variable taking 3 values could have bounds $1,3$ or $2,4$, etc.
- FnGradAvail equals 1 if function gradient estimates are available, and 0 otherwise.
- NumConstraintGradAvail gives the number of constraints for which gradient estimates of the left-hand side of the constraints are available. Any such differentiable constraints are assumed to come first in the collection of m constraints in the simulation implementation function below.
- StartingSol is a matrix with NumStartingSol rows and d columns, where each row contains a starting solution. If NumStartingSol is 0 then this equals **NaN** (not a number).
- budget is a column vector containing a collection of suggested budgets. (Recall that a budget is the amount of computer time, measured as specified in the problem description, allowed for the optimization problem to complete its search.
- ObjBd is a bound (upper bound for maximization problems, lower bound for minimization problems) on the optimal solution value, or **NaN** if no such bound is known.

- OptimalSol is a d dimensional column vector giving an optimal solution if known, and it equals **NaN** if no optimal solution is known.

Let us turn our attention now to the function that performs the simulation itself.

function [fn, FnVar, FnGrad, FnGradCov, constraint, ConstraintCov, ConstraintGrad, ConstraintGradCov] = Ambulance(x, runlength, seed, other);

The meaning of the inputs to this function are as follows.

- x is a d-dimensional column vector giving the solution at which to simulate.
- runlength is a nonnegative real number specifying the runlength, measured in time units as specified in the problem description.
- seed is a strictly positive integer giving the seed for random variate generation. In fact, it is not actually a seed, but rather the index of the substream of random numbers to be used. See Section 3.2 below for full details.
- other is a catchall for, e.g., problem parameters like arrival rates in queues if the problem is so parameterized. For most problems it is ignored.

The meaning of the outputs from this function are as follows.

- fn is the real-valued estimate of the function value at the point x. The value **NaN** is returned if the point x is "hard" infeasible in the sense that the x values cannot be used in the simulation, as opposed to "soft" infeasible when the problem constraints are not satisfied.
- FnVar is a real-valued estimate of the variance of the estimator fn. In the common case where n i.i.d. replications are performed, this quantity equals s^2/n, where s^2 is the sample variance of the n observations. If a variance estimate is not computed, this is returned as **NaN**.
- FnGrad is a d-dimensional column vector giving an estimate of the gradient of the function at x, returned as a d-vector or **NaN** if gradients are unavailable or inappropriate.
- FnGradCov is a d $\times$ d estimated covariance matrix of the function gradient estimate, including any scaling to account for the number of replications (see FnVar above). The constant (not a matrix) **NaN** is returned if the gradient covariance is not available or not appropriate.
- constraint is an m-dimensional column vector giving the estimated left-hand side (LHS) of inequality constraints that are assumed to be of the form LHS ≥ 0. If $m = 0$ so there are no constraints, then the constant (not a matrix) **NaN** is returned.
- ConstraintCov is the estimated covariance matrix of the constraint LHS estimates. This could have zero rows/columns corresponding to deterministic constraints. It is returned as the constant (not a matrix) **NaN** if there are no constraints, or if constraint covariance estimates are not computed.
- ConstraintGrad is a d $\times$ k matrix giving the estimated gradients of the $k =$NumConstraintGradAvail constraint LHS values for which gradients are available, one in each column. If there are no constraints then the constant (not a matrix) **NaN** is returned.
- ConstraintGradCov is a d$k\times$ dk matrix giving the estimated covariance matrix of the Constraint LHS gradient estimates, where $k =$ NumConstraintGradAvail. Entries are ordered by constraint and then by gradient component, i.e., the ith row (or column) corresponds to Component $(i \mod \mathrm{d})$ of the gradient of Constraint $(\lfloor (i-1)/\mathrm{d} \rfloor + 1)$. The constant (not a matrix) **NaN** is returned if constraint gradients are not available.

In this setup we have chosen to separately identify the function, function gradient, constraint and constraint gradient estimates, rather than returning some kind of matrix of outputs with all of these quantities contained therein. We have done so because this approach seems clearer. This clarity comes at a small (from our perspective) cost. For example, there could be dependence between the function estimates and the function gradient estimates, but such dependence is not reported in the above interface.

```
% Create a separate stream for each source of primitive inputs needed, e.g., for M/M/c+M queue
[ArrivalStream, ServiceStream, PatienceStream] = RandStream.create('mrg32k3a', 'NumStreams', 3);

% Set the substream for each source of randomness to the "seed" provided.
ArrivalStream.Substream = seed;
ServiceStream.Substream = seed;
PatienceStream.Substream = seed;

% Pregenerate the random variables for the simulation.
OldStream = RandStream.setDefaultStream(ArrivalStream); % Store old stream and set stream for generating arrivals
InterarrivalTimes = exprnd(InterArrivalMean, runlength, 1);
ArrivalTimes = cumsum(InterarrivalTimes);

RandStream.setDefaultStream(ServiceStream);
ServiceTimes = exprnd(ServiceMean, runlength, 1);

RandStream.setDefaultStream(PatienceStream);
PatienceTimes = exprnd(PatienceMean, runlength, 1);

RandStream.setDefaultStream(OldStream);
```

Figure 1: Random number streams for random variate generation in an $M/M/c+M$ queue.

3.2 Random Number Streams

Some optimization algorithms rely on the use of common random numbers (CRN), while others rely on independence. To accommodate these preferences, the optimization code should call the simulation with a specified "seed," which is a positive (≥ 1) integer. This integer is not really a seed, but instead specifies the substream used in generating primitive inputs within the simulation model. Indeed, MATLAB$^{\text{TM}}$contains a limited implementation of streams and substreams, as described in L'Ecuyer et al. (2002). The simulation code creates its own random number generators, so that the optimization code is free to use a randomized algorithm that will be independent of the estimates produced by the simulation code.

Hence, to implement CRN, the optimization code should simply use the same "seed" with every call to the simulation code. To implement independence for, e.g., stochastic approximation, the optimization code should use a new seed at every step, which could simply be a counter that reflects the number of times the simulation code has been called (starting at 1).

As an example of how this is implemented in MATLAB$^{\text{TM}}$, Figure 1 depicts code for generating the inputs for an $M/M/c+M$ (multiserver queue with abandonments) queue, for the first runlength customers, each of which needs an interarrival time, a service time and an abandonment time.

Statistics Toolbox generators, e.g., exprnd, betarnd, random use **rand** or **randn**, which in turn use the default random number generator. (See the MATLAB$^{\text{TM}}$documentation on Random Number Generators under "Statistics Toolbox: Distribution Functions.") Therefore, we set the default stream before generating each set of inputs, and finish by restoring the stream that was used before the simulation was called.

We have elected to pre-generate and store all random variables needed for the simulation, rather than to generate them as needed. There are two reasons for this. First, MATLAB$^{\text{TM}}$operations are more efficiently completed when vectorized, and so there is a speed advantage this way, at the expense of some storage. Second, this may also help with synchronization issues if the number of random variables needed in a replication depends on x.

4 COMPARING ALGORITHMS

In this section, we present the idea of *wrappers* — function protocols designed to facilitate the comparison of competing SO algorithms through rigorous and automatic finite-time performance assessment. Wrappers are a realization of assessment ideas discussed in Pasupathy and Henderson (2006). In what follows, we present the structure of wrappers and provide a simple example for illustration.

4.1 Wrappers

A wrapper is essentially a protocol designed to seamlessly execute a chosen algorithm on a chosen problem. The idea of wrappers is meant to take us one step further toward our medium-term objective of developing and maintaining a sister library of contributed algorithms for SO problems. Like SimOpt, the sister library will be a "living" archive of algorithms organized by attributes specific to the class of problems they are designed to solve.

The inputs to the proposed wrapper are "AlgorithmName" corresponding to the algorithm chosen for execution, "aparam" giving a vector of algorithm parameters corresponding to "AlgorithmName,", "ProblemName" corresponding to the problem on which the chosen algorithm is to be executed, "budget" giving suggested budgets for the optimization algorithm, and "quantiles" specifying which quantiles of performance to estimate. The primary performance measure of interest is the (true) objective function value $g(X(t))$ evaluated at the solutions $X(t)$ returned by the algorithm at each of the budgets $t = t_1, t_2, \dots, t_n$ specified through the problem description. Accordingly, towards estimating summary measures of the random variable $g(X(t))$, the wrapper performs independent and identically distributed runs of the algorithm to obtain realizations $X_1(t), X_2(t), \dots,$ of the solutions at each of the budgets $t = t_1, t_2, \dots, t_n$. The objective function value $g(X(t))$ is then estimated (to negligible error) by calling the simulation at each of the realized solutions.

Output from the wrapper includes a graph containing estimated quantiles of $g(X(t))$, plotted as a function of the desired vector of budgets. (See Pasupathy and Henderson (2006) for more details on finite-time assessment of SO algorithms.) The output from the wrapper is unique to the algorithm-problem combination. Users wanting to compare multiple algorithms on a set of problems, or assess a single algorithm on multiple problems, can easily write driver programs that repeatedly call the wrapper.

We now provide a little more detail on the inputs and outputs of the proposed wrapper.

```
function [OptVal, OptGap, ErrOptSol, handleOptVal, handleOptGap, handleOptSol]
            = SOwrapper(AlgorithmName, aparams, ProblemName, budget, quantiles);
```

The meaning of the various quantities in this function are as follows.

- AlgorithmName is the unique name given to the algorithm being assessed.
- aparams is a column vector of algorithm parameters (of type double) required to execute the chosen algorithm. For instance, the stochastic approximation algorithm may need a constant K_1 used to construct the gain sequence, and a constant K_2 used to construct the step-sizes needed for estimating derivatives. The two constants K_1, and K_2 are then specified through aparams(1), and aparams(2). The size of the vector aparams might be different for different algorithms.
- ProblemName as mentioned in Section 3.1 is the unique name of the problem on which the chosen algorithm is being assessed.
- budget is a column vector of budgets for the algorithm to use in solving the problem. If this quantity equals **NaN** then the default budgets specified as part of the problem description are used, but otherwise it overrides the problem-specified budgets.
- quantiles is a column vector of values in the interval $(0,1)$ specifying the quantiles of $g(X(t))$ that are to be estimated for each budget t. If this quantity equals **NaN** then the default quantiles $0.1, 0.5$ and 0.9 are used.

- OptVal is an output matrix of real numbers corresponding to the estimated quantiles of the objective function values measured at the solutions returned by the algorithm. More formally, OptVal is the matrix of estimated pre-specified quantiles of the random variables $g(X(t)), t = t_1, t_2, \ldots, t_n$ where $t_1, t_2, \ldots, t_n$ are the budgets requested through the problem description. Accordingly, OptVal has size $l \times q$, where l is the size of the vector budget and q is the number of reported quantiles. For example, if budget is [100; 500; 1000], and the 0.25, 0.50, 0.75 and 0.90 quantiles are to be reported, then $l = 3$ and $q = 4$.
- OptGap is an output matrix similar to OptVal, but reports estimated quantiles of an upper bound on the optimality gaps measured at the solutions returned by the algorithm. So, if $X(t)$ is the random variable described earlier and v^* is some known lower bound on the optimal value of a minimization problem, then OptGap returns various estimated quantiles of $|g(X(t)) - v^*|$. Like OptVal, the matrix OptGap has size $l \times q$, where l is the size of the vector budget and q is the number of reported quantiles. The value v^* is returned as the output ObjBd from the problem-structure function, and if v^* is unknown then OptGap takes the value **NaN**.
- ErrOptSol is an output matrix similar to OptGap, but it reports estimated quantiles of the deviations (in L_2 norm) of the solutions $X(t)$ returned by the algorithm from a known optimal solution x^*. The solution x^* is returned as the output OptimalSol from the problem-structure function. Like OptVal and OptGap, the matrix ErrOptSol has size $l \times q$, where l is the size of the vector budget and q is the number of reported quantiles. The value **NaN** is returned if x^* is unknown.
- handleOptVal, handleOptGap, and handleErrOptSol are handles to graphs with abscissae as elements of budget, and each having q curves corresponding to the estimated quantiles reported through the matrices OptVal, OptGap, and ErrOptSol respectively. If v^* or x^* are not available, the corresponding handle is returned as **NaN**.

A few other comments relating to wrappers are warranted.

(i) Problem names and algorithm names are pre-specified through the SimOpt library and the proposed algorithm library, respectively.

(ii) Recall that the vector input aparams allows using some of the algorithm parameters as user inputs. This is only to promote flexibility — algorithm submitters can choose to hardcode some or all of these parameter choices inside their code, in which case the hardcoded parameters will not be included in the input vector aparams. For instance, if an implementation of stochastic approximation includes the option of using common random numbers, then a switch to use (or not) common random numbers will appear as an element in aparams. Likewise, implementations that do not provide this flexibility will not include this switch as part of aparams.

(iii) The wrapper assumes a standardized calling function for the algorithm. Specifically, it assumes that the algorithm can be called by

```
function [x, other] = AlgorithmName(aparams, ProblemName, budget, seed);
```

where aparams, ProblemName and budget are as described earlier, and seed is a positive integer (an initial seed) that tags the specific independent run of the algorithm. Independent and identically distributed runs of the algorithm can thus be obtained by changing the value of seed. The output x is a matrix whose jth column $(j \geq 2)$ corresponds to the solution returned by the algorithm after expending budget(j−1) computing effort, where budget is the column vector of budgets suggested either by the chosen problem (see Section 3.1) or by the call to the wrapper (see above). The first column of x returns the starting solution. The vector other is a "catch-all" for all other outputs returned by the algorithm.

(iv) As mentioned earlier, the proposed wrapper will facilitate assessing and comparing algorithms for sets of problems. We envision users writing driver programs that repeatedly call the wrapper to generate quantile curves depicting the performance of algorithms of interest. Due to obvious difficulties, and as discussed in Pasupathy and Henderson (2006), we have deliberately refrained from constructing single measures of finite-time performance for SO algorithms.

(v) Depending on the nature of the problem, the matrices OptGap and ErrOptSol may make little sense to the user. For example, in settings where a local extremum is sought, v^* and x^* are not of much relevance because the number of local extrema may be numerous.

(vi) Due to the potential need to return large matrices, and since the quality of a solution is usually measured in the function space rather than in the design space, we have chosen not to return any summary statistics on the solutions attained by the algorithm.

4.2 An Example

As an example of using the proposed wrapper, we discuss the calling commands for assessing a basic version of the "RandomLocalSearch" algorithm on the ambulance problem, and demonstrate the output generated by the wrapper.

Assuming that a user wishes to assess RandomLocalSearch on Ambulance through the proposed wrapper, the relevant MATLAB syntax is

```
function [OptVal, OptGap, ErrOptSol, handleOptVal, handleOptGap, handleOptSol]
          = SOwrapper('RandomLocalSearch', [], 'Ambulance', NaN, [0.15; 0.5; 0.75; 0.9])
```

As explained in the previous section, the first and third inputs are the name of the algorithm and the name of the problem being assessed. The second input is an empty vector because RandomLocalSearch has no algorithm parameters that are user inputs. The budget is specified as NaN since we will use the default budget specified in the Ambulance problem description, which is $[10000; 25000; 50000; 75000; 100000]$, and the vector of quantiles overrides the defaults in the wrapper.

The wrapper returns **NaN** for OptGap, ErrOptSol, handleOptGap and handleOptSol because a lower bound on the optimality gap v^* and the optimal solution x^* were not provided as part of Ambulance. OptVal is a 4×5 matrix of estimated $0.15, 0.5, 0.75, 0.90$ quantiles of the performance measure $g(X(t))$ at each of the budget points $t = 10000, 25000, 50000, 75000, 100000$. A handle to a corresponding plot is provided through the output handleOptVal, and is depicted in Figure 2.

In Figure 2, all quantiles are equal at $t = 0$ since this corresponds to the fixed starting solution provided through the problem. Also, all randomness in the random variable $g(X(t))$ is assumed to come from the random variable $X(t)$. In other words, when estimating $g(X(t))$ at a specified $X(t)$, enough replications are performed by the wrapper so that the sampling error due of the objective function (for a given solution $X(t)$) is negligible.

5 DISCUSSION

We end with a renewed call for contributions to increase the number and variety of problems available on SimOpt. While we currently have a sizable library, there is a noticeable skew in the nature of available problems. For example, integer-ordered and continuous-variable problems constitute the bulk of what is currently available, with a severe dearth of SO problems having categorical variables. Similarly, amongst the available continuous-variable problems, most are either unconstrained or have deterministic constraints. Towards remedying this skew, we particularly welcome SO problems having categorical variables of all constraint types, and SO problems having continuous variables with stochastic constraints.

We also encourage volunteers to test their SO algorithms on problems that are currently available in the library. The wrapper interface described in Section 4 is specifically meant to facilitate such testing. Standardized output from the wrapper interface will ease distribution and comparison. The proposed sister library of algorithms, in addition to hosting particular algorithm implementations, will contain summarized output generated from executing the wrapper interface on some or all of the available algorithms, and on chosen sets of problems.

Figure 2: Quantile curves generated by the proposed wrapper when RandomLocalSearch is executed on Ambulance.

ACKNOWLEDGMENTS

We thank the participants, and particularly the organizers Michael C. Fu and Barry L. Nelson, of a workshop and a follow-up meeting on simulation optimization held on May 24–25, 2010, for advice and suggestions. We had additional valuable conversations with Bruce Schmeiser, particularly about wrappers. We also thank the many contributors to the library, including a number of students at both Virginia Tech and Cornell, who have written up and coded problems. This work was supported by the National Science Foundation under Grant Numbers CMMI-0800608 and CMMI-0800688.

REFERENCES

Fu, M. C. 2002. “Optimization for simulation: Theory vs. practice”. *INFORMS Journal on Computing* 14:192–215.

Fu, M. C., S. Andradóttir, J. S. Carson, F. Glover, C. R. Harrell, Y. C. Ho, J. P. Kelly, and S. M. Robinson. 2000, December. “Integrating optimization and simulation: research and practice”. In *Proceedings of the 2000 Winter Simulation Conference*, edited by J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, 610–616. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Glynn, P. W. 2002. “Additional perspectives on simulation for optimization”. *INFORMS Journal on Computing* 14:220–222.

L’Ecuyer, P., R. Simard, E. J. Chen, and W. D. Kelton. 2002. “An Object-Oriented Random-Number Package with Many Long Streams and Substreams”. *Operations Research* 50 (6): 1073–1075.

Pasupathy, R., and S. Henderson. 2006, December. “A Testbed of Simulation-Optimization Problems”. In *Proceedings of the 2006 Winter Simulation Conference*, edited by L. F. Perrone, F. P. Wieland,

J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, 255–263. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

R. Pasupathy and S. G. Henderson 2011. “SimOpt”. Accessed Oct. 23, 2011. http://www.simopt.org.

AUTHOR BIOGRAPHIES

RAGHU PASUPATHY is an associate professor in the Industrial and Systems Engineering Department at Virginia Tech. His research interests lie broadly in Monte Carlo methods with a specific focus on simulation optimization and stochastic root finding. He is a member of INFORMS, IIE, and ASA, and serves as an Associate Editor for ACM TOMACS and the INFORMS Journal on Computing. His e-mail address is pasupath@vt.edu and his web page is https://filebox.vt.edu/users/pasupath/pasupath.htm.

SHANE G. HENDERSON is a professor in the School of Operations Research and Information Engineering at Cornell University. His research interests include discrete-event simulation and simulation optimization, and he has worked for some time with emergency services. He is the simulation area editor at Operations Research, an associate editor for *Management Science* and for *Stochastic Systems*, and currently serves as the chair of the INFORMS Applied Probability Society. He co-edited the Proceedings of the 2007 Winter Simulation Conference. His web page is http://people.orie.cornell.edu/∼shane.