Paper SAS256-2014

New Features in SAS/OR® 13.1

Ed Hughes and Rob Pratt, SAS Institute Inc.

ABSTRACT

SAS/OR® software for operations research includes mathematical optimization, discrete-event simulation, and project and resource scheduling capabilities. This paper surveys a number of its new features that better equip you to address decision-making challenges such as planning, resource management, and asset allocation.

Optimization performance improvements help you solve larger, more detailed problems more quickly. Improvements encompass linear, mixed integer linear, and nonlinear optimization, and include multithreading of the mixed integer linear solver and major improvements in the performance and functionality of the decomposition algorithm for linear and mixed integer linear optimization.

The OPTMODEL procedure for optimization modeling adds direct access to the same set of efficient network optimization algorithms available via the OPTNET procedure in SAS/OR, enabling you to embed network optimization as a component of larger solution processes. Other new features enable you to execute multiple optimizations in parallel and use the FCMP procedure to define functions.

The OPTLSO procedure for global and local search optimization adds the ability to work with multiple objective functions and produce a set of Pareto-optimal solutions. This approach enables you to manage the trade-offs that arise between competing objectives and adds to the range of optimization problems that you can solve using PROC OPTLSO. Another new feature is support for the READ_ARRAY function in PROC FCMP, with which you can much more easily input array-structured data to be used in function definitions.

Finally, SAS® Simulation Studio for discrete-event simulation enhances its graphical interface to better support customization and increase ease of use.

INTRODUCTION

This paper surveys the enhancements and new features that are included in SAS/OR 13.1 and SAS Simulation Studio 13.1, a component of SAS/OR 13.1 for Windows operating systems. Recognizing that features are useful only when they are used, the paper emphasizes the practical implications of these additions. Selected examples illustrate several of the new features.

This paper assumes that you have a basic understanding of operations research problems and methods, and so it does not dwell on basic concepts in optimization or discrete-event simulation, except as needed to provide context for the new features in each area. Instead, the paper concentrates on how the changes in SAS/OR 13.1 can make your analytic modeling work easier and more productive.

IMPROVED OPTIMIZATION CAPABILITIES

SAS/OR 13.1 delivers across-the-board improvements in optimization solver performance, with speed increases of 30% in mixed integer linear optimization, 20% in linear optimization, and 6% in nonlinear optimization. This makes it possible for you to solve larger, more complex problems with greater ease and in a shorter amount of time. Moreover, SAS/OR 13.1 broadens the range of problems that you can solve, deepens integration between SAS/OR optimization modeling methods and with SAS® at large, adds more powerful modeling capabilities, and enables you to use your available computational resources more effectively.

THE CLP PROCEDURE

The CLP procedure uses finite domain constraint programming methods to model and solve general and scheduling-oriented constraint satisfaction problems (CSPs). Because a CSP is essentially an optimization problem without an objective, CSPs are useful in finding initial feasible solutions to problems—whether feasibility is the end goal itself or simply a starting point for optimization. In SAS/OR 12.1, PROC CLP added the OBJECTIVE statement, which enables you to specify an objective function and thus use PROC CLP as an alternative optimization solver. In SAS/OR 13.1, this feature moves from experimental to production status, qualifying PROC CLP as a production-quality alternative method of optimization. As always, PROC CLP is especially adept at modeling several classes of constraints that are difficult or impossible to model using conventional optimization, including element constraints (one variable's value determines which of a finite set of values another variable takes), global cardinality constraints (setting upper and lower bounds on the number of variables in a group that can take a given value), and all-different constraints (requiring that each variable in a group take a different value).

THE OPTLSO PROCEDURE

The OPTLSO procedure uses a variety of local and global search methods to perform optimization for general nonlinear functions that are defined using the FCMP procedure in Base SAS® for both continuous and integer variables. PROC OPTLSO makes no assumptions about the nature of these functions; objective functions and constraints can be nonsmooth, discontinuous, and computationally expensive to evaluate.

In SAS/OR 13.1, you can specify more than one objective via the OBJECTIVE= option in the PROC OPTLSO statement, enabling you to work with cases in which multiple criteria are weighed against each other. PROC OPTLSO provides multiobjective optimization output that enables you to explore the trade-offs that can exist between these different, possibly conflicting, objectives. Practical examples of multiobjective optimization are easy to cite. In financial investment, typical goals are to maximize return *and* minimize risk. In transportation system design, you would probably want to minimize cost, minimize delays or waiting times, and maximize use of human and vehicle resources. In industrial design, minimization of weight and maximization of structural strength can both be important to achieve.

In general, PROC OPTLSO requires you only to specify your optimization problem (with functions defined using PROC FCMP) and then applies multiple instances of global and local search algorithms in parallel to solve it. Similarly, for multiobjective optimization, PROC OPTLSO requires only that you specify multiple PROC FCMP–defined functions in the OBJECTIVE= data set, each with a "min" or "max" value of the _SENSE_ variable (to indicate that each function is an objective and not an intermediate function). No special syntax is needed.

In solving a multiobjective optimization problem, PROC OPTLSO returns a set of Pareto-optimal or nondominated solutions. For example, for a problem that minimizes k objectives $f_1,\ldots,f_k$ and two solutions x and y, solution x is dominated by solution y if $f_i(x) \geq f_i(y)$ for all $i \in \{1,\ldots,k\}$ and $f_j(x) > f_j(y)$ for some $j \in \{1,\ldots,k\}$. In general, y dominates x if, for each objective being considered, y performs at least as well as x, and if, for at least one such objective, y performs distinctly better than x. For each solution in a Pareto-optimal set of solutions, there is no *other* solution that dominates it in this manner.

Figure 1 (SAS Institute Inc. 2013) illustrates the concepts of Pareto optimality and domination. Two functions, $f_1(x)$ and $f_2(x)$, are being minimized and 10 points, each corresponding to a solution, are plotted. Points $a, b,$ and d are Pareto-optimal, and the entire Pareto-optimal set, also known as the efficient frontier, is depicted by the curve. Point a dominates points $e, f,$ and k; point b dominates points $e, f, g,$ and k; and point d dominates points i and k. Point c dominates points $g, h,$ and k but is not in the Pareto-optimal set, because there are points near it that have smaller values of both $f_1(x)$ and $f_2(x)$.

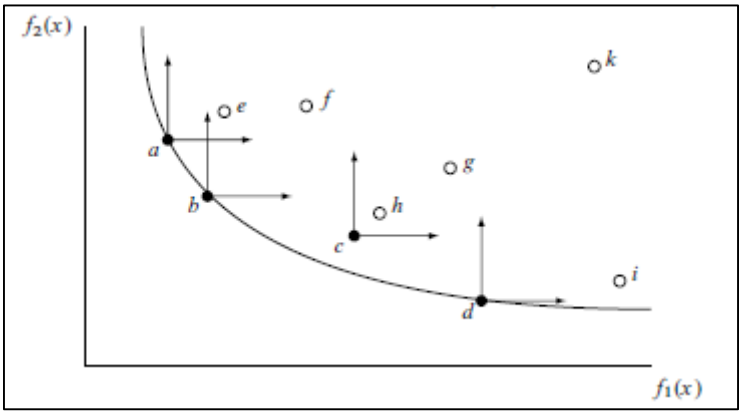


Figure 1. Pareto-Optimal Set and Dominated Solutions

Note that you can limit the size of the Pareto-optimal set of solutions by specifying a value for the PARETOMAX= option in the PROC OPTLSO statement.

Using genetic algorithms, PROC OPTLSO can identify a Pareto-optimal set of solutions to a multiobjective optimization problem in a single solution process, rather than needing to solve several problems. Local searches are conducted around these solutions to improve objective function values and to reduce crowding among the solutions; this method ensures that the solutions that PROC OPTLSO returns correspond to a real range of choices and do not simply represent minor variations on or perturbations of the same solution.

As the genetic algorithms proceed through successive generations, the population of Pareto-optimal solutions evolves. With the derivative-free approach that PROC OPTLSO takes, it's difficult to verify directly that solutions are Pareto-optimal, and so PROC OPTLSO instead uses a heuristic measure to gauge how the population of Pareto-optimal solutions changes from generation to generation. This is an indirect measure of the convergence of the

population of solutions to a Pareto-optimal set of solutions. This metric sums, for all solutions that transition from nondominated to dominated status in the current generation, the distance between the solution and the Pareto-optimal set, accounting for both the objective functions and any constraints. Stabilization in the value of this metric across several generations indicates that the Pareto-optimal set is becoming stable and thus suggests convergence.

Here's a simple example of multiobjective optimization that uses PROC OPTLSO to minimize two functions, f_1 and f_2, where

$$f_1(x) = (x_1 - 1)^2 + (x_1 - x_2)^2 \quad \text{and} \quad f_2(x) = (x_1 - x_2)^2 + (x_2 - 3)^2$$

subject to variable bounds $0 \leq x_1, x_2 \leq 5$. The data set VarData describes the variables and their bounds:

```
data vardata;
   input _id_ $ _lb_ _ub_;
   datalines;
x1 0 5
x2 0 5
;
```

PROC FCMP defines the two functions that are used as objectives:

```
proc fcmp outlib=sasuser.myfuncs.mypkg;
   function fdef1(x1, x2);
      return ((x1-1)**2 + (x1-x2)**2);
   endsub;

   function fdef2(x1, x2);
      return ((x1-x2)**2 + (x2-3)**2);
   endsub;
run;
```

The data set ObjData declares the two PROC FCMP functions as objective functions and indicates that each objective function is to be minimized:

```
data objdata;
   input _id_ $ _function_ $ _sense_ $;
   datalines;
f1 fdef1 min
f2 fdef2 min
;
```

Finally, PROC OPTLSO receives this information via its VARIABLE= and OBJECTIVE= input data set options and solves the problem. The set of Pareto-optimal solutions is stored in the Solution data set, which is specified using the PRIMALOUT= option.

```
options cmplib = sasuser.myfuncs;
proc optlso
   primalout = solution
   variables = vardata
   objective = objdata
   logfreq = 50
;
run;
```

The Solution data set contains information about 4,425 Pareto-optimal solutions that PROC OPTLSO finds, and stores information about each solution in multiple observations. The TRANSPOSE procedure produces a data set that contains one observation per solution, and the GPLOT procedure displays a plot of f_1 versus f_2 for every solution in the set.

```
proc transpose data=solution out=pareto label=_sol_ name=_sol_;
   by _sol_;
   var _value_;
   id _id_;
run;

proc gplot data=pareto;
```

```
   plot f2*f1;
run;
```

Figure 2 shows a plot of these solutions, with f_1 plotted along the horizontal axis and f_2 plotted along the vertical axis. Your choice of an individual solution from this Pareto-optimal set would be guided by your interpretation of the relative importance of minimizing each of the two objectives. Placing more weight on minimizing f_1 would mean choosing a solution on the left side of the graph, whereas an emphasis on minimizing f_2 would result in choosing a solution on the right side of the graph. In practice, a graphical display like this could provide general guidance, but your final choice of a solution would come from direct exploration of the underlying data set.

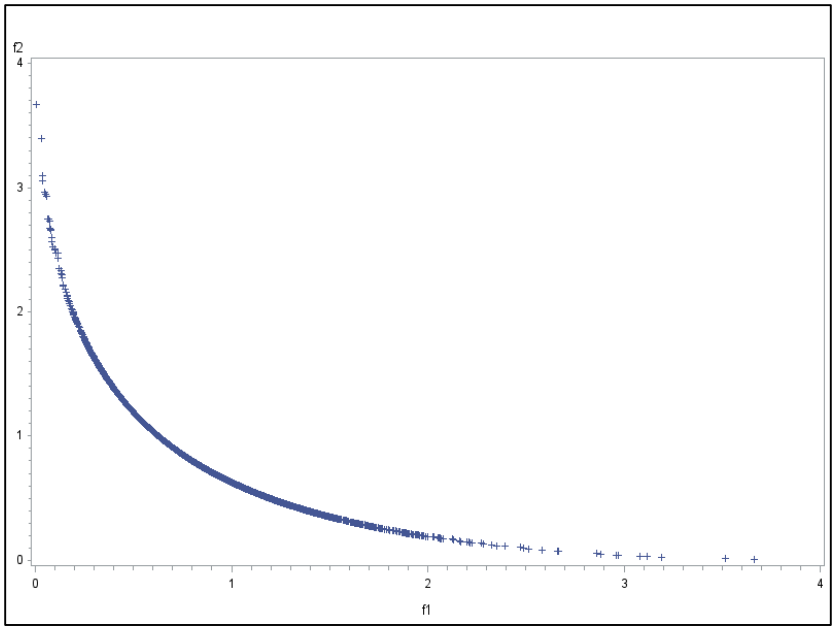


Figure 2. Plot of Pareto-Optimal Set for Multiobjective Optimization Problem

In SAS/OR 13.1, PROC OPTLSO also adds the READARRAY statement, which in turn enables you to use the READ_ARRAY function in PROC FCMP as you define functions to be used in your optimization problem. PROC FCMP uses the READ_ARRAY function to read a data set into an array variable that is used to define a function. In the READARRAY statement, you list each data set that PROC FCMP reads in this way. The purpose of the READARRAY statement is to ensure that the data sets used in defining functions are available. Because reading array-structured data is a natural way to specify indexed elements of a function, the addition of the READARRAY statement is a very significant improvement in ease of use for PROC OPTLSO.

A short example shows how the READARRAY statement helps you use a data set to define a function in a problem to be solved using PROC OPTLSO. This example shows how to minimize the Bard function (Moré, Garbow, and Hillstrom 1981), a function of three variables that has three classes of parameters and 15 subfunctions and is defined as

$$f(x) = \frac{1}{2}\sum_{k=1}^{15} f_k^2(x), \qquad x = (x_1, x_2, x_3)$$

where

$$f_k(x) = y_k - \left(x_1 + \frac{k}{v_k x_2 + w_k x_3}\right)$$

where $v_k = 16 - k,\ w_k = \min(u_k, v_k)$ and

$$y = (0.14, 0.18, 0.22, 0.25, 0.29, 0.32, 0.35, 0.39, 0.37, 0.58, 0.73, 0.96, 1.34, 2.10, 4.39)$$

The variable bounds are $-1000 \le x_i \le 1000$ for $i = 1, 2, 3$. Variable information is stored in the data set VarData.

```
data vardata;
   input _id_ $ _lb_ _ub_ ;
   datalines;
```

```
x1  -1000 1000
x2  -1000 1000
x3  -1000 1000
;
```

The key issue is which method you use to specify the y vector of parameters. Before the availability of the READARRAY statement, you were required to include these parameters explicitly in the PROC FCMP syntax that defines the Bard function, as in the following:

```
proc fcmp outlib=sasuser.myfuncs.mypkg;
   function bard(x1, x2, x3);
      array y[15] /nosym (0.14 0.18 0.22 0.25 0.29
                          0.32 0.35 0.39 0.37 0.58
                          0.73 0.96 1.34 2.10 4.39);
      fx = 0;
      do k=1 to 15;
         vk = 16 - k;
         wk = min(k,vk);
         fxk = y[k] - (x1 + k/(vk*x2 + wk*x3));
         fx = fx + fxk**2;
      end;
      return (0.5*fx);
   endsub;
run;

data objdata;
   input _id_ $ _function_ $ _sense_ $;
   datalines;
f   bard     min
;

options cmplib = sasuser.myfuncs;
proc optlso
   primalout = solution
   variables = vardata
   objective = objdata;
run;
```

The READARRAY statement enables you to store the contents of the y vector in a data set and then read that data set in PROC FCMP by using its READ_ARRAY function. The PROC OPTLSO syntax adds the READARRAY statement to indicate the use of the BardData input data set by PROC FCMP.

```
data barddata;
   input y @@;
   datalines;
0.14 0.18 0.22 0.25 0.29
0.32 0.35 0.39 0.37 0.58
0.73 0.96 1.34 2.10 4.39
;

proc fcmp outlib=sasuser.myfuncs.mypkg;
   function bard(x1, x2, x3);
      array y[15] /nosym;
      rc = read_array('barddata', y);
      fx = 0;
      do k=1 to 15;
         dk  = (16-k)*x2 + min(k,16-k)*x3;
         fxk = y[k] - (x1 + k/dk);
         fx = fx + fxk**2;
      end;
      return (0.5*fx);
   endsub;
```

```
run;

data objdata;
   input _id_ $ _function_ $ _sense_ $;
   datalines;
f   bard     min
;

options cmplib = sasuser.myfuncs;
proc optlso
   primalout = solution
   variables = vardata
   objective = objdata;
   readarray barddata;
run;
```

This is preferable to the former method because it uses the source data directly rather than requiring you to reproduce the data in the ARRAY statement in PROC FCMP. This eliminates the possibility that data errors will occur during this transition. The new approach is even more preferable as the size of the data set grows, because the potential for errors grows accordingly and PROC FCMP code that explicitly describes the parameter data becomes unwieldy.

In both cases, PROC OPTLSO identifies the correct optimal solution, as shown in Figure 3.

Optimal Solution

Obs	_sol_	_id_	_value_
1	0	_obj_	0.00411
2	0	_inf_	0.00000
3	0	x1	0.08240
4	0	x2	1.13259
5	0	x3	2.34411
6	0	f	0.00411

Figure 3. Optimal Solution Minimizing the Bard Function

THE OPTMODEL PROCEDURE: NETWORK SOLVER

PROC OPTMODEL provides a powerful algebraic optimization modeling language as part of an interactive environment in which you can build and solve a wide range of optimization models. In SAS/OR 13.1, PROC OPTMODEL adds the network solver (experimental), providing direct access to the set of 11 network analysis and optimization algorithms that are also accessible through PROC OPTNET in SAS/OR.

These algorithms work with networks (also called graphs), in which nodes (representing locations, time periods, individuals, and so on) are connected by arcs (representing communication, travel, flow of material, and so on). Arcs between nodes can be directed, indicating a direction of flow from one node to another, or undirected, simply indicating connection between two nodes. There are myriad applications of network optimization and analysis.

The SOLVE WITH NETWORK statement invokes the network solver. This statement is especially useful when you are solving a network-oriented problem or when a network analysis is used as a component of a larger solution or analytic process. Unlike other optimization solvers that PROC OPTMODEL invokes, the network solver operates directly on arrays and sets that you specify. You don’t need to explicitly define decision variables, constraints, and objectives. These model elements are handled implicitly when you specify the arrays and sets that define your network.

Options in the SOLVE WITH NETWORK statement fall into three categories. *General options* include basic options for any network and controls on time spent and log output that are applicable to any chosen algorithm. *Input and output options* specify the names of input sets and arrays that describe the network (LINKS=, NODES=, and others) and output sets and arrays produced by the individual algorithms (OUT= and many other sets and arrays that contain output produced by a specific algorithm). Finally, *algorithm options and suboptions* specify which of the 11 major algorithm classes to use and also detail options regarding convergence criteria, limits on the amount of output data produced, and other detailed settings. Table 1 lists the network algorithm classes and their corresponding options in the SOLVE WITH NETWORK statement.

Algorithm Class	SOLVE WITH NETWORK Option
Biconnected components	BICONCOMP
Maximal cliques	CLIQUE=
Connected components	CONCOMP
Cycle detection	CYCLE=
Linear assignment (matching)	LINEAR_ASSIGNMENT
Minimum-cost network flow	MINCOSTFLOW
Minimum cut	MINCUT=
Minimum spanning tree	MINSPANTREE
Shortest-path	SHORTPATH=
Transitive closure	TRANSITIVE_CLOSURE
Traveling salesman	TSP=

Table 1. Algorithm Classes in the Network Solver

A short example illustrates the use of the network solver in PROC OPTMODEL. A SAS employee wants to identify the shortest path (measured in travel time) from her home in Raleigh (614 Capital Boulevard) to SAS headquarters in Cary (SAS Campus Drive). The travel distances and speed limits between these two points and several intermediate locations (road intersection points) during the morning rush hour are described in the following data set:

```
/* Road Network Shortest Path                                    */

data LinkSetInRoadNC10am;
   input start_inter $1-20 end_inter $20-40 miles miles_per_hour;
   datalines;
614CapitalBlvd      Capital/WadeAve      0.6  25
614CapitalBlvd      Capital/US70W        0.6  25
614CapitalBlvd      Capital/US440W       3.0  45
Capital/WadeAve     WadeAve/RaleighExpy  3.0  40
Capital/US70W       US70W/US440W         3.2  60
US70W/US440W        US440W/RaleighExpy   2.7  60
Capital/US440W      US440W/RaleighExpy   6.7  60
US440W/RaleighExpy  RaleighExpy/US40W    3.0  60
WadeAve/RaleighExpy RaleighExpy/US40W    3.0  60
RaleighExpy/US40W   US40W/HarrisonAve    1.3  55
US40W/HarrisonAve   SASCampusDrive       0.5  25
;
```

PROC OPTMODEL transforms speed limit and distance data into travel time for each link and then invokes the network solver, specifying the shortest-path algorithm. This formulation generalizes so that you can compute shortest paths between many pairs of locations; for this example, the origin and destination for the desired shortest path are simply the employee's home and SAS headquarters, respectively.

```
proc optmodel;
   set<str,str> LINKS;
   num miles{LINKS};
   num miles_per_hour{LINKS};
   num time_to_travel{<i,j> in LINKS} = miles[i,j]/ miles_per_hour[i,j] * 60;
   read data LinkSetInRoadNC10am into
      LINKS=[start_inter end_inter]
      miles miles_per_hour
   ;
   /* You can compute paths between many pairs of source and destination,
```

```
      so these parameters are declared as sets */
   set HOME = /"614CapitalBlvd"/;
   set WORK = /"SASCampusDrive"/;

   /* The path is stored as a set of: Start, End, Sequence, Tail, Head */
   set<str,str,num,str,str> PATH;

   solve with network /
      links     = ( weight = time_to_travel )
      shortpath = ( source = HOME
                    sink   = WORK )
      out       = ( sppaths = PATH )
   ;
   create data ShortPath from [s t order start_inter end_inter]=PATH
      time_to_travel[start_inter,end_inter];
quit;
```

The shortest path in order from home to SAS is shown in Output 1. The estimated travel time is 11.5582 minutes.

Shortest Morning Commute (Home to SAS)

order	start_inter	end_inter	time_to_travel
1	614CapitalBlvd	Capital/WadeAve	1.4400
2	Capital/WadeAve	WadeAve/RaleighExpy	4.5000
3	WadeAve/RaleighExpy	RaleighExpy/US40W	3.0000
4	RaleighExpy/US40W	US40W/HarrisonAve	1.4182
5	US40W/HarrisonAve	SASCampusDrive	1.2000
			11.5582

Output 1. Shortest Path for Road Network for Morning Commute

THE OPTMODEL PROCEDURE: CONCURRENT FOR LOOP

PROC OPTMODEL has always included a FOR loop, in which a specified set of statements is executed iteratively for each member of an index set. In SAS/OR 13.1, PROC OPTMODEL also provides a concurrent FOR loop, specified using the new COFOR statement, in which SOLVE statements are executed concurrently with other statements in the loop. Other iterations of the COFOR loop can be processed while an earlier iteration waits for a SOLVE statement to finish. When the SOLVE statement completes execution, any subsequent statements in the same iteration are processed.

In most cases, you can substitute a COFOR loop for a FOR loop simply by altering the FOR statement, with minimal or no other changes. A COFOR loop can contain other PROC OPTMODEL control and looping statements, and COFOR loops can be nested within each other. Each solver invocation in a COFOR loop occupies a single computational thread, and the maximum number of threads that any COFOR loop can use is controlled by the specifications of the PERFORMANCE statement and other SAS options.

It's worth noting that in a COFOR loop, the order of the output from different iterations can vary between runs, because the SOLVE statements in the iterations can finish in a different order. If you require deterministic output—or, more importantly, if results from a SOLVE statement in one iteration must be used in a subsequent iteration—it's best to use a FOR loop instead. If instead you execute several SOLVE statements in a single COFOR loop, then it's permissible for a solver invocation to depend on the results from an earlier solver invocation, but only within the same COFOR loop iteration.

A portfolio optimization example illustrates the use of the COFOR statement. The task is to select assets for investment. The restrictions are that the entire pool of money must be invested and at least 10% must be invested in any selected asset. Risk, which is measured by the weighted sum of the pairwise covariance among asset returns, must be held at or below a maximum acceptable level. Return should be maximized. The presence of the 10% minimum investment constraint usually requires you to formulate this problem as a mixed integer nonlinear

optimization problem, because before choosing investment levels for all assets, you must decide which assets to invest in (there are more than 10 assets, so simply requiring a 10% investment in every asset is infeasible). Representing inclusion or exclusion of assets requires you to use binary (0 or 1) variables, and these variables transform a nonlinear optimization problem into a mixed integer nonlinear optimization problem.

The approach that is taken here avoids including variables that represent asset inclusion or exclusion by randomly selecting a set of assets to include for each of a series of independent trials. In each trial, you solve a nonlinear optimization problem:

$$
\begin{aligned}
&maximize \quad r^T x \\
&subject\ to \sum_{a \in A_i} x_a = 1 \\
&\qquad x^T W x \leq R_{max} \\
&\qquad 0.1 \leq x \leq 1
\end{aligned}
$$

where A_i is the set of assets selected for trial i, x is the vector of decision variables that represent the proportion invested in each asset, r is the vector of assets' mean annual returns, W is the covariance matrix of asset returns, and R_{max} is the maximum acceptable level of risk. For each trial, you record the return that the optimal portfolio produces and compare it to the best return achieved for any trial, updating the best return if the portfolio from the current trial outperforms all prior portfolios.

You can use PROC OPTMODEL to implement this solution method. Ten trials are specified, and the nonlinear optimization problem to be solved in each trial is declared.

```
proc optmodel;
   set ASSETS;
   num return {ASSETS};
   num cov {ASSETS, ASSETS} init 0;
   read data lib.means into ASSETS=[_n_] return;
   read data lib.covdata into [asset1 asset2] cov cov[asset2,asset1]=cov;
   num riskLimit init 0.00025;
   num minThreshold init 0.1;
   num numTrials = 10;

   /* declare NLP problem for fixed set of assets */
   set ASSETS_THIS;
   var AssetPropVar {ASSETS_THIS} >= minThreshold <= 1;
   max ExpectedReturn =
      sum {i in ASSETS_THIS} return[i] * AssetPropVar[i];
   con RiskBound:
      sum {i in ASSETS_THIS, j in ASSETS_THIS} cov[i,j] * AssetPropVar[i] *
      AssetPropVar[j] <= riskLimit;
   con TotalPortfolio:
      sum {asset in ASSETS_THIS} AssetPropVar[asset] = 1;

   num infinity = constant('BIG');
   num best_objective init -infinity;
   set INCUMBENT;
```

Random selection of assets (using the RAND function), nonlinear optimization, and updates to the best return take place in a COFOR loop.

```
call streaminit(1);
cofor {trial in TRIALS} do;
      put;
      put trial=;
      ASSETS_THIS = {i in ASSETS: rand('UNIFORM') < 0.5};
      put ASSETS_THIS=;
      solve with NLP / logfreq=0;
      put _solution_status_=;
      if _solution_status_ ne 'INFEASIBLE' then do;
         if best_objective < ExpectedReturn then do;
            best_objective = ExpectedReturn;
```

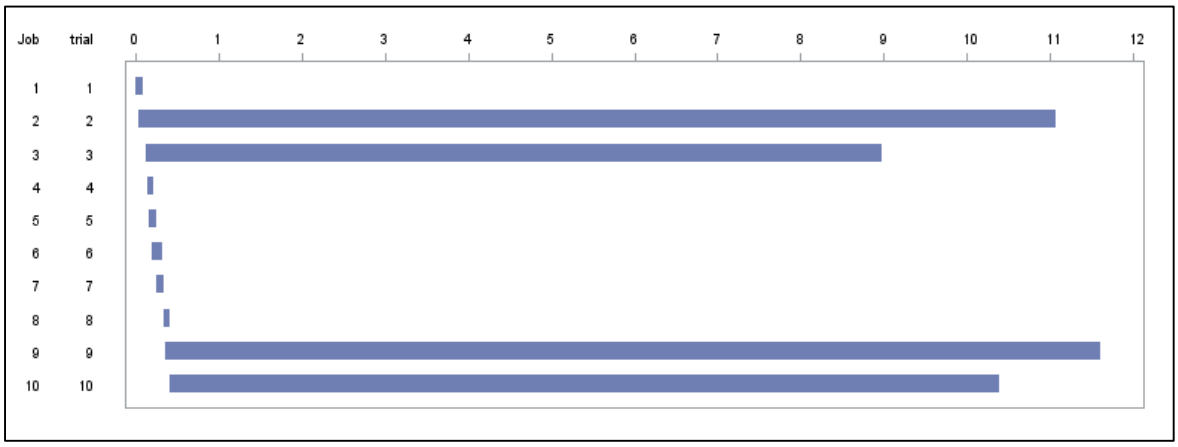
```
            INCUMBENT = ASSETS_THIS;
            put best_objective= INCUMBENT=;
         end;
      end;
   end;
   put best_objective= INCUMBENT=;
```

Use of the COFOR loop enables the nonlinear optimizations to take place concurrently, on multiple computational threads, as needed. In fact, this parallel execution produces significant time saving, as indicated in the SAS log excerpt shown in Output 2.

```
NOTE: PROCEDURE OPTMODEL used (Total process time):
      real time           11.91 seconds
      cpu time            39.31 seconds
```

Output 2. Real Time and CPU Time Used by PROC OPTMODEL

The CPU time that is used is more than triple the real time that is used, indicating that substantial concurrent optimization occurred. You can confirm this by adding some minor time tracking to the program and using the GANTT procedure in SAS/OR to display a Gantt chart of the execution of the 10 iterations, as shown in Output 3.



Output 3. Gantt Chart of Overlapping COFOR Loop Iterations

THE OPTMODEL PROCEDURE: PROC FCMP–DEFINED FUNCTIONS AND SUBROUTINES

PROC OPTMODEL in SAS/OR 13.1 can call functions and subroutines that are defined and compiled using PROC FCMP in Base SAS. In PROC OPTMODEL, you can use a PROC FCMP function anywhere you can use any standard SAS function, and you can use the CALL statement to call PROC FCMP subroutines. This capability enables you to reuse previously defined functions and subroutines, significantly reducing the work that you must do to create sophisticated optimization models and more deeply integrating your use of PROC OPTMODEL with your use of other SAS procedures. In addition, PROC FCMP can call a broader variety of functions than PROC OPTMODEL and can create some functional forms that are cumbersome or impossible to create directly by using PROC OPTMODEL.

Along with parameters and variables, you can pass PROC OPTMODEL arrays to PROC FCMP functions and subroutines that are designed to accept matrix input. PROC FCMP functions return values that can be used in PROC OPTMODEL, and PROC FCMP subroutines can return data by updating PROC OPTMODEL parameters that are passed as arguments in the corresponding CALL statement and declared using the OUTARGS statement in the subroutine definition.

A short example illustrates the value of this capability. Hadamard's maximal determinant problem (Hadamard 1893; Sloane and Orrick 2011) asks you to find the largest possible determinant of a matrix of a specified order in which each element has values from a specified set, usually $\{-1, 1\}$ or $\{0, 1\}$. In this example, the requirements of the classical Hadamard problem are relaxed, so each element of the matrix has a value between −1 and 1. PROC OPTMODEL cannot access the special PROC FCMP determinant function directly since it is not a generally available SAS function. Performing the calculation of the determinant of an array variable using PROC OPTMODEL syntax would be cumbersome at best. Using PROC FCMP, however, you can call the determinant function, and because

PROC OPTMODEL can call functions that are defined using PROC FCMP, you can access the determinant function from PROC OPTMODEL.

This PROC FCMP call creates the MYDET function:

```
proc fcmp outlib=work.myfuncs.test;
     function mydet(x[*,*]);
        call det(x, result);
        return (result);
     endsub;
quit;
```

PROC OPTMODEL solves a modified Hadamard problem for a 5 × 5 matrix. It declares the elements of the matrix as bounded decision variables, uses the MYDET function as the objective function to be minimized, and calls the nonlinear optimization solver in multistart mode.

```
options cmplib = work.myfuncs;
proc optmodel;
     num n = 5;
     var x {1..n, 1..n} >= -1 <= 1;
     max z = mydet(x);
     solve with NLP / ms;
     print x;
quit;
```

Even though there is no integer restriction on the values of the decision variables, in the optimal solution that is reported each variable has an integer value −1 or 1, as shown in Output 4. The determinant of this matrix is 48.

	x				
	1	2	3	4	5
1	-1	-1	1	-1	-1
2	-1	1	1	1	-1
3	-1	-1	-1	1	1
4	1	-1	-1	1	-1
5	1	-1	1	1	1

Output 4. Optimal Solution to Relaxed Hadamard Problem

THE OPTMODEL AND OPTLP PROCEDURES: CONCURRENT LINEAR AND NONLINEAR OPTIMIZATION

The concurrent solve capabilities for linear and nonlinear optimization attain full production status in SAS/OR 13.1. Specified using the ALGORITHM=CONCURRENT option in the PROC OPTLP statement or the SOLVE statement in PROC OPTMODEL, concurrent solve executes all applicable solution algorithms in parallel (single-machine mode), as permitted by the number of computational cores available. The first algorithm to terminate returns its optimal solution. This feature can provide valuable information to guide you in selecting an optimization solver and is especially helpful if you need to solve similarly structured problems repeatedly. This need often arises if optimization is used as a subcomponent of a larger repetitive process. In such cases, it makes sense to use concurrent solve on one or more sample problems for guidance on solver performance. Then you can select the best algorithm to use in a full-scale implementation. Concurrent solve is available for linear optimization in PROC OPTMODEL and PROC OPTLP and for nonlinear optimization in PROC OPTMODEL.

THE OPTMODEL AND OPTMILP PROCEDURES: PARALLEL MIXED INTEGER LINEAR OPTIMIZATION

An experimental feature in SAS/OR 13.1 enables the branch-and-cut mixed integer linear optimization solver to execute in parallel on multiple computational cores (single-machine mode). To enable parallel execution, you specify

the PARALLEL=1 option in the PROC OPTMILP statement or the SOLVE WITH MILP statement in PROC OPTMODEL. This new capability can significantly reduce the time needed to complete optimization and in general means that you can solve larger, more complex problems more quickly.

Figure 4 shows a performance profile that compares solution times for the parallel implementation of the mixed integer linear optimization solver that uses four computational threads (labeled SAS/OR 13.1 4t) and the serial implementation (labeled SAS/OR 13.1). Each implementation was run on a suite of 500 test problems that have great diversity in problem size, problem complexity, and difficulty of the solution process.

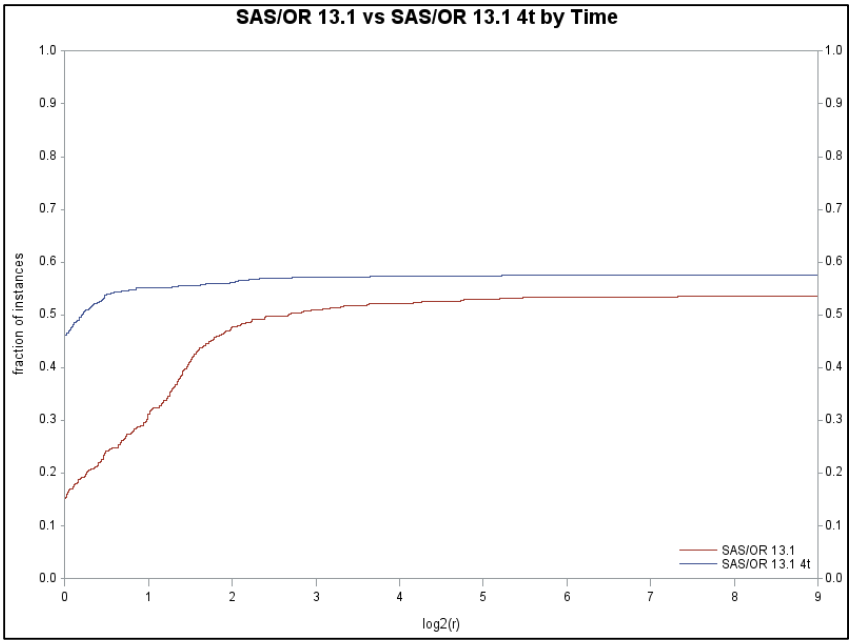


Figure 4. Performance Profile Comparing SAS/OR 13.1 Parallel (Four Threads) and Serial Mixed Integer Linear Optimization Solvers

In a performance profile, for each optimization solver (here, either parallel or serial) the plotted line indicates the proportion of the overall problem set that can be solved as a function of the time allotted to solve each problem. The time scale is logarithmic and is relative to the shortest solution time for each problem. Thus, the plotted position for each solver that corresponds to zero on the time axis indicates the proportion of the problems that it can complete as quickly as the fastest solver tested, the point for 1 on the time axis indicates the proportion solved in up to twice as much time as the fastest solver requires, and so on. The plotted position at the right limit of the time axis approximates the proportion of the problems that the solver can complete within the one-hour time limit per problem.

In this performance profile, the plot for the parallel solver is consistently above the plot for the serial solver, indicating that it solves almost all the problems more quickly and solves more of the problem set overall than the serial solver. Only the smallest of the problems are solved too quickly to benefit from parallel processing. Overall, the parallel solver completes 288 of the 500 problems, whereas the serial solver completes 268.

Note that in PROC OPTMODEL, parallel execution of the MILP solver is not possible within a COFOR loop, because by rule each SOLVE statement that is executed in a COFOR loop is allocated a single computational thread. Parallel execution is usually more efficient at higher levels than at lower levels.

DISCRETE-EVENT SIMULATION UPDATES

SAS Simulation Studio 13.1, a component of SAS/OR 13.1 for Windows environments that performs discrete-event simulation, includes several enhancements to its graphical modeling and analysis interface. These changes make SAS Simulation Studio easier to use and enhance your ability to customize your models.

When you are running a model in SAS Simulation Studio, the analog clock portion of the Simulation Clock icon animates, displaying moving analog clock hands, to indicate that model execution is occurring. This is especially helpful, for example, when data are being read in or written and when a SAS program or JMP® script is being processed using the SAS Program block, because at these times the simulation clock does not advance. In addition, if you choose to pause a model's execution, this is confirmed in the Simulation Clock icon by a small blue "Pause" icon being overlaid on the upper left region of the clock icon. When the model completes execution, a small red

“Stop” icon is overlaid instead. If you stop execution, the digital portion of the simulation clock display is reset to zero and the entire display (including the replication count) is inactive.

SAS Simulation Studio 13.1 enables you to define a custom image for an instance of a block, and it provides a new tab in each block’s Block Properties dialog box to facilitate this. The **Visual** tab, shown in Figure 5, contains the new Block Image controls in addition to the Block Label controls, which have been reproduced for this tab.

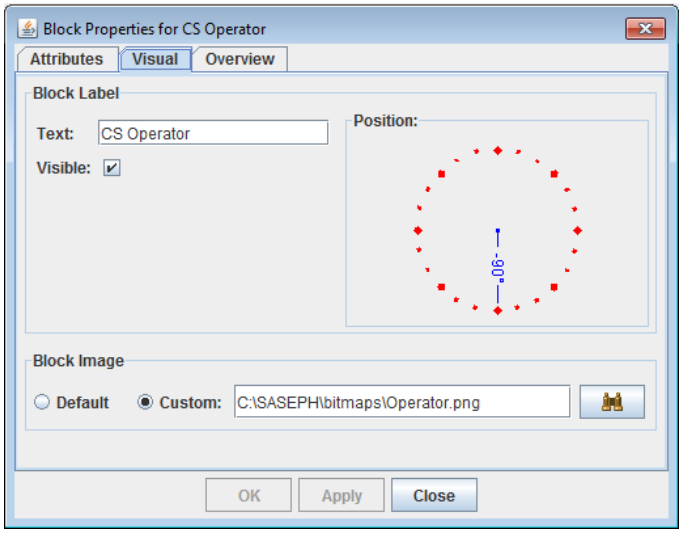


Figure 5. Visual Tab of a Block Properties Dialog Box

For submodel blocks, changes in SAS Simulation Studio 13.1 make it easier for you to add blocks in the Definition view. A small template icon appears for the Definition view of a submodel next to the Instance/Definition selector box; hovering the mouse pointer over this icon causes a new instance of the Block Template Display Area to appear in a pop-up window. From this window, you can select a block from any block template and drag it to your submodel; you no longer need to drag blocks from the templates in the main interface of SAS Simulation Studio (which might be concealed by the submodel window). Figure 6 shows a submodel window in which this additional Block Template Display Area is activated.

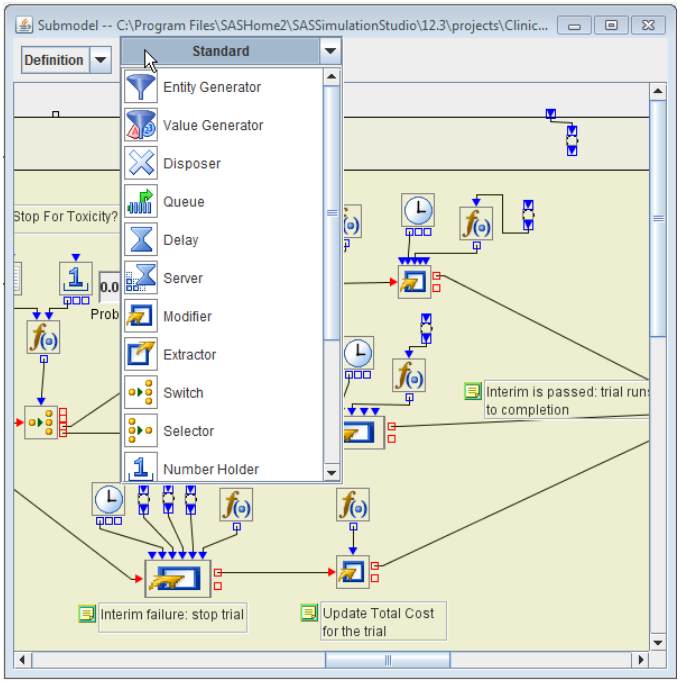


Figure 6. Submodel Window with Additional Block Template Display Area Activated

SAS Simulation Studio 13.1 also includes improved and expanded statistics calculations in the Resource Stats Collector block. Finally, SAS Simulation Studio 13.1 streamlines the method that you use to specify an InStreamPolicy port value or a DataStreamDescription factor for a Numeric Source block to define the probability distribution that is sampled. You no longer need to specify a pathname as a prefix to each distribution type specification. This means, for example, that the means by which you specify a normal distribution with mean 1 and standard deviation 2 changes from

```
class==com.sas.analytics.simulation.datastream.distribution.Normal;Mean==1;Std Dev==2
```

for SAS Simulation Studio 12.3 and earlier to

```
class==Normal;Mean==1;Std Dev==2
```

for SAS Simulation Studio 13.1.

CONCLUSION

SAS/OR 13.1 includes significant new and enhanced capabilities for several types of optimization and for discrete-event simulation. These changes enable you to build models and solve problems more easily, address a wider range of problems more efficiently, interact more extensively with other SAS software and data, and make more productive use of your computational resources.

REFERENCES

Hadamard, J. (1893). "Résolution d'une question relative aux déterminants." *Bulletin des Sciences Mathématiques, 17,* 240–246.

Moré, J. J., Garbow, B. S., and Hillstrom, K. E. (1981). "Testing Unconstrained Optimization Software." *ACM Transactions on Mathematical Software, 7,* 17–41.

SAS Institute Inc. (2013). "*SAS/OR 13.1 User's Guide: Local Search Optimization.*" Cary, NC: SAS Institute Inc.

Sloane, N. J. A. and Orrick, W. P. (2011). "A003433: Hadamard Maximal Determinant Problem: Largest Determinant of (+1,–1)-matric of Order n." *The On-Line Encyclopedia of Integer Sequences*. http://oeis.org/A003433.

ACKNOWLEDGMENTS

The authors thank Manoj Chari, Menal Guzelsoy, Emily Lada, Yu-Min Lin, Yan Xu, and Lois Zhu for their contributions of information and examples. The authors also express their gratitude to Ed Huddleston for his invaluable contributions as editor.

RECOMMENDED READING

- *SAS/OR 13.1 User's Guide: Mathematical Programming*
- *SAS/OR 13.1 User's Guide: Mathematical Programming Examples*
- *SAS/OR 13.1 User's Guide: Network Optimization Algorithms*
- *SAS/OR 13.1 User's Guide: Local Search Optimization*
- *SAS/OR 13.1 User's Guide: Constraint Programming*
- *SAS Simulation Studio 13.1: User's Guide*

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors:

Ed Hughes
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Ed.Hughes@sas.com

Rob Pratt
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Rob.Pratt@sas.com