RC22664 (W0212-040) December 6, 2002
Mathematics

IBM Research Report

The IBM Stochastic Programming System

Alan J. King[*], Stephen E. Wright[**], Gyana R. Parija[*], Robert Entriken[***]
[*]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

[**]Miami University
Oxford, Ohio

[***]Stanford University
Palo Alto, California

The IBM Stochastic Programming System

Alan J. King[*],
Stephen E. Wright[†],
Gyana R. Parija[‡],
Robert Entriken[§]

June 18, 2002
Revision September 20, 2002

1 Introduction

IBM's stochastic programming product, the Optimization Solutions and Library Stochastic Extensions (OSLSE), was developed at IBM Research's Thomas J. Watson Research Center in Yorktown Heights, New York, in the period 1990-2002. It is a library of subroutines that may be linked with user-written C/C++ programs to model and solve multi-period stochastic linear programs with recourse. Features include: quadratic objectives, integer variables, empirical tree generation, and a flexible nested decomposition solver. A parallel version of the nested decomposition solver is also available.

The currently available version (version 3) has been extensively revised from its initial 1998 release. It now uses the OSL version 3 C/C++ infrastructure for problem data management and solver utilities. OSLSE may be freely downloaded with a 60-day try-and-buy license from the OSL website `http://www-3.ibm.com/software/data/bi/osl/index.html`. Free academic licenses are available for students and academic researchers.

2 A brief history

As with most scientific software projects, the development of the IBM stochastic programming system was conditioned by the needs of its customers. The major influences were a request from the Frank Russell Company to establish the solvability of industrial scale stochastic programs using a simplex-algorithm based

[*]IBM Thomas J. Watson Research Center, Yorktown Heights, New York, `kingaj@us.ibm.com`
[†]Miami University, Oxford, Ohio, `wrightse@muohio.edu`
[‡]IBM Thomas J. Watson Research Center, Yorktown Heights, New York, `parija@us.ibm.com`
[§]Stanford University, Palo Alto, California, `entriken@stanford.edu`

approach, and an extended consultation with the Allstate Insurance Corporation's Research Center in Menlo Park, California, on a strategic asset allocation system.

2.1 The Frank Russell Problem

IBM Research's stochastic programming project originated with a request in late 1989 from Roger Wets and Bill Ziemba, consultants with the Frank Russell Company, to explore the solvability of a multiperiod stochastic program arising from an insurance asset-liability management problem (Cariño et al. 1994). The stochastic program arrived in the form of three large files in a version of the SMPS format (Birge et al. 1987) (see chapter **??** of this volume). It had 10 periods and 2048 scenarios, and was well beyond the capabilities of any stochastic programming software in existence at that time.

Bob Entriken, then a post-doc at IBM Research, wrote an SMPS reader and matrix generator for the Frank Russell problem based on the node-arc metaphors from his thesis (Entriken 1989). He also suggested a simple solution idea: to solve the first 75 scenarios, transfer the first-stage solution to a larger 250 scenario problem, and so bootstrap our way to solving the full 2048 scenario problem. Alan King implemented this idea on an IBM vector mainframe. The total runtime was around 26 hours, which was mostly spent inverting the enormous matrices. This rather crude approach did demonstrate that a simplex method could be adapted to reliably solve stochastic programs of an industrial size, and Frank Russell went on to develop the world's first commercial asset-liability management product.

2.2 Allstate Asset Allocation Project

Allstate Insurance, like many sophisticated firms in the financial industry, analyzes statistics of returns from its investments and losses from its insurance lines of business in order to develop insight into their risk management and investment activities. In the period 1991-3, Allstate and IBM collaborated on the design and implementation of an asset liability management solution for the allocation of investment surplus into asset categories. The members of the Allstate team were Tom Ward and Bill Love, and those of the IBM team were Mike Haydock, Nancy Soderquist, Luke Smith, and Alan King.

The major components of the Allstate asset allocation solution were a simulator for stochastic asset returns and liability flows, a dynamic balance sheet to generate the multiperiod accounting measurements, and a utility function composed of combinations of (up to 26 different) piecewise-linear penalty and reward functions. The simulator could be thought of as arising from a three-factor model, although the linkages between factors and the asset/liability flows were non-linear. The number of time periods was six, with interperiod intervals ranging from one month to more than one year. The number of asset classes was eleven, the majority of which were bonds of various maturities and grades. Trading in any asset class was possible at the beginning of every time period.

The most complex part of the dynamic balance sheet was to account for taxes of various types. Finally, the liability exposure represented Allstate's stochastic property losses from extreme weather in the southern United States. The main use of the Allstate asset allocation system was to understand the tradeoff between three standard accounting quantities: net income, growth of shareholder's equity and risk-based capital — and one statutory quantity: the ratio of premium income to statutory surplus (an important measurement of portfolio quality in the insurance industry). The major insight sought was guidance on optimal investment allocations to the equity and fixed income maturity classes under a variety of economic outlooks.

A single run of the program consisted of: specifying a set of economic scenarios for the simulation run, identifying active components of the utility function, solving several hundred parametrized stochastic linear programs (each with several hundred thousand constraints and variables), and recording features of the optimal solution along the parametric frontier. In the course of their investigation, the Allstate team ran hundreds and hundreds of such runs.

Besides the challenge of building software to efficiently process enormous amounts of stochastic programming data, the Allstate project also posed new functional requirements for IBM's nascent stochastic programming system. These were (1) to process simulations into an empirical scenario tree, and (2) to generate a "frontier" of optimal solutions by parametric variation of the components of the utility function. In addition to those functions needed to support the processing of SMPS data, these became core components of the stochastic programming system under development at IBM Research, described in the next section.

2.3 First generation: SPOSL

Alan King adopted Bob Entriken's SMPS reader/generator, and from 1990 to 1995, with major assistance from Steve Wright during his post-doc appointment 1991-3, developed a general-purpose stochastic programming system, with both SMPS and direct input formats and a nested L-shaped decomposition method. The elegant SPL-file object was entirely designed and written by Steve Wright. He also designed the parallel decomposition code at Miami University during the 1993-1994 academic year and implemented it under a summer contract at IBM in 1994. Hercules Vladimirou assisted with an early implementation of a two-stage L-shaped method in 1992. During a summer student visit in 1993, Chris Donohue contributed a prototype of the code that bundles simulations into scenario trees. The program came to be called SPOSL (King 1994) and was distributed under license to roughly a dozen university students and professors.

2.4 Product release: OSLSE

IBM released the stochastic programming system in 1998 as part of the Optimization Solutions and Library family of products (IBM 1995), calling it the

OSL Stochastic Extensions (OSLSE). Gyana Parija was the lead developer supporting the initial product release. He translated the software to new OSL C/C++ interfaces, and upon joining IBM Research in 2000, undertook the testing and implemention of integer programming techniques for stochastic programs.

3 Stochastic Input Formats

OSLSE supports two types of input formats: *SMPS* and *internal arrays*. The underlying optimization problem can be linear, convex (or concave) quadratic, and/or mixed integer. The distribution types supported are *independent discrete*, *block discrete*, and *scenarios*. In addition, OSLSE supplies a *tree generation* utility that bundles simulations into scenarios.

3.1 SMPS input

The Stochastic MPS format (SMPS) was created chiefly to support the creation of problem test sets. See Chapter **??** of this volume for a definition. It describes a multiperiod stochastic program using specially structured files. The Core file contains an MPS description of a linear program. The Time file indicates how the variables and constraints of the linear program may be divided into time stages. The Stoch file describes the probability distributions of the elements of the Core linear program that are random in the stochastic program. In this section we outline special features or idiosyncracies of the OSLSE implementation of the SMPS standard.

The Core file. Integer variables may be indicated by using delimiters in the Core file in the manner described in (IBM 2001). OSLSE automatically propagates these throughout the stochastic program, so that an integer variable in the second stage of the Core file (say) will result in the generation of an integer variable for each second-stage node in the stochastic program. OSLSE supports free-format input of Core files.

The Time file. OSLSE supports only the `IMPLICIT` keyword in the PERIODS statement. This means that rows and columns in the core file must be sorted in period order. A future release will support the `EXPLICIT` keyword (the direct interface already supports it).

The Stoch File. The Stoch file sections supported by OSLSE are `SCENARIOS`, `INDEP DISCRETE`, and `BLOCK DISCRETE`. There must be only one section type in a given Stoch file. ADD and REPLACE options for all these sections are allowed. The internal data structures in OSLSE allow the Stoch file to contain matrix data elements that are not defined in the Core file.

Infinite core bound warning. An idiosyncracy of OSLSE arises from the design of its internal data structures where stochastic data is stored as a scenario tree whose values *add* to Core values. This design has certain advantages, but one important drawback: if a Stoch file element replaces or adds to an infinite bound in the Core file then a warning is printed and the stochastic element is ignored.

3.2 Input via Internal Arrays

Processing SMPS files can be slow, and writing SMPS files can be exacting. Our experience is that most developers will want to write stochastic programming applications in a programming language such as C/C++ or in a modeling language such as GAMS or AMPL. For this reason OSLSE provides user interfaces for the input of stochastic program data. Calling sequences for these functions are described in the Appendices. A simple driver showing their use is described in Appendix A.

Core and Time Data. The function `ekks_createCore()` passes the linear programming data and time stage information. Time stage information is passed explicitly. All indexing in OSLSE is Fortran-style, so the lowest row, column, or time-stage index is 1 (one). OSLSE supports quadratic and mixed-integer core problems. One may pass non- negative diagonal quadratic matrix entries and set the core problem type to quadratic to invoke a quadratic solver. Integer variables may be declared by a call to `ekks_setIntegersAtCore()`.

Stoch Data. The function `ekks_addScenario()` passes stochastic data in the *scenarios* data format. All indexing in OSLSE is Fortran-style, so the lowest scenario index is 1 (one). A repeat call to `ekks_addScenario()` with the same scenario index merely adds to the probability weight for that scenario. The number of rows and columns in the scenario data must be identical to their counterparts in the core data. The value of the variable `replace` indicates whether the stochastic data adds to or replaces the core data. Future implementations will support functions `ekks_addIndep()` and `ekks_addBlock()` to allow one to pass independent discrete distributions directly to OSLSE.

3.3 Generating Scenario Tree by Bundling Simulations

OSLSE includes an independent set of routines for the purpose of building a scenario tree out of a sequence of simulations. The basic idea is to implicitly generate a subfiltration from a finite partition of the support of the random variables and use simulation to assign empirical weights to the nodes of the subfiltration.

For a concrete example, suppose each member of a sequence of random variables $\{X_t\}_{t=1}^{T}$ has a sample space that is a subset of $\mathbf{R}^d$. Partitioning each coordinate into (say) 5 intervals generates a finite subfiltration with 5^{dT} possible paths. Simulations can be used to build an empirical scenario tree as follows.

Each simulation $\{\xi_t\}_{t=1}^T$ is mapped to a sequence of *events*, $\{e_t\}_{t=1}^T$, which are integers from $1,\ldots,5^d$ recording the label of the d-rectangle that ξ_t lies in. When a new simulation is added (by calling `ekks_addEventToTree()`) its initial event sequence is compared against all event sequences so far received and the longest match found is recorded as the parent of the incoming event sequence. Weights of the unmatched nodes in the incoming sequence are initialized at one and the weights of the matched nodes of the parent sequence are incremented by one.

The result is a scenario tree that describes the empirical distribution of the sequence of simulations relative to the event subfiltration. The event data, together with scenario numbers, branching node, and probability weights can be retrieved by `ekks_getScenarioFromTree()`. To pass the scenario to the stochastic program, one maps the event index to scenario data values (say, to the centroid of the d-rectangle or the expected value conditioned on the d-rectangle) and calls `ekks_addScenario()`.

Simulation relies on the laws of large numbers to reduce the effort needed to approximate aspects of the solution. Unfortunately, the dimension of a stochastic programming solution is so large that one cannot expect laws of large numbers effects in any stage but the first. In the Allstate case, experiments were performed to determine the effectiveness of this kind of sampling of scenario trees. It was found that the error distributions of the first stage solution vector (considered one dimension at a time) did appear to show central limit properties: errors were distributed normally about a central value with a variance that was decreasing linearly in the number of simulations.

4 The Decomposition Solver

OSLSE implements a flexible nested decomposition solver to take advantage of the structural properties of stochastic programs. It is difficult to discuss the solver without reference to a presentation of the method, so for completeness we give a brief overview here.

4.1 The L-shaped Method

Consider an L-shaped linear program

$$\begin{array}{lll} \min_{(x_1,x_2)} & c_1x_1 + c_2x_2 & \\ \text{s.t.} & A_{11}x_1 & \in [y_1^-, y_1^+] \\ & A_{21}x_1 + A_{22}x_2 & \in [y_2^-, y_2^+] \\ & x_1 \in [x_1^-, x_1^+] & \\ & x_2 \in [x_2^-, x_2^+] & \end{array} \tag{1}$$

Form the *parent subproblem* by minimizing out the second set of variables

$$\begin{array}{lll} \min_{(x_1,\theta_1)} & c_1x_1 + \theta_1 & \\ & A_{11}x_1 & \in [y_1^-, y_1^+] \\ \text{s.t.} & x_1 & \in [x_1^-, x_1^+] \\ & \theta_1 & \geq f_2(x_1) \end{array} \tag{2}$$

The impact of the second set of variables is represented abstractly in the parent by the constraint involving the value of the *child subproblem*

$$f_2(\bar{x}_1) = \begin{cases} \min_{x_1,x_2} & c_2 x_2 \\ & x_1 = \bar{x}_1 \\ \text{s.t.} & A_{21}x_1 + A_{22}x_2 \in [y_2^-, y_2^+] \\ & x_2 \in [x_2^-, x_2^+] \end{cases} \tag{3}$$

The L-shaped method algorithm iteratively approximates the abstract constraint by computing *optimality cuts* and *feasibility cuts* from the optimality conditions of the child subproblem for successive parent proposals. (In passing the child problem 3 to OSL, it turns out to be convenient to make the constraints $x_1 = \bar{x}_1$ explicit. The cuts then turn out to be the reduced costs associated with this set of constraints. OSL will in any case substitute the fixed values $\bar{x}_1$ where required.)

The parent subproblem after the k-th step of the method has the form

$$\begin{array}{lll} \min_{(x_1,\theta_1)} & c_1x_1 + \theta_1 & \\ & A_{11}x_1 & \in [y_1^-, y_1^+] \\ \text{s.t.} & x_1 & \in [x_1^-, x_1^+] \\ & \theta_1 e + F^k x_1 & \geq f^k \\ & G^k x_1 & \geq g^k \end{array} \tag{4}$$

where e is the vector of ones with dimension equal to the number of rows in the matrix F^k. The rows of the matrices F^k and G^k and the right-hand side vectors f^k and g^k are formed by the collection of optimality and infeasibility cuts, respectively. Van Slyke and Wets (1969) showed that the following algorithm terminates in a finite number of steps.

Decomposition Algorithm.

0. Solve (4).

1. If infeasible, original problem (1) is infeasible. STOP.
2. If unbounded, continue with ray $\bar{x}_1^k + t\bar{u}_1^k, t \geq 0$.
3. If optimal, continue with solution $(\bar{x}_1^k, \bar{\theta}_1^k)$.

1. Solve the child subproblem (3) with parent proposal $\bar{x}_1^k$.

1. If unbounded, original problem is unbounded. STOP.
2. If infeasible, generate infeasibility cut from the sum of infeasibilities $f_2^I(\bar{x}_1^k)$ and reduced costs $\bar{\delta}_1^k$

   $$-\bar{\delta}_1^k x_1 \geq f_2^I(\bar{x}_1^k) - \bar{\delta}_1^k \bar{x}_1^k \tag{5}$$

   Add coefficients as a new row of matrix G^k and element of right-hand side g^k. Return to step 0 with $k \mapsto k+1$.

3. If optimal with optimal value $f_2(\bar{x}_1^k)$ and parent was bounded go to the Optimality Test. Otherwise go to the Unboundedness Test.

Optimality Test. Test the optimality gap $f_2(\bar{x}_1^k) - \bar{\theta}_1^k$.

1. If it is within tolerance, optimal solution is found. STOP.
2. Otherwise, generate optimality cut from the reduced costs $\bar{\delta}_1^k$

$$\theta_1 - \bar{\delta}_1^k x_1 \geq f_2(\bar{x}_1^k) - \bar{\delta}_1^k \bar{x}_1^k \tag{6}$$

   Add the coefficients as a new row of matrix F^k and element of right-hand side f^k. Go to step 0 with $k \mapsto k+1$.

Unboundedness Test. Form the child recession subproblem from child subproblem (3). (Set all finite bounds to zero, all positive column infinite bounds to 1, and all negative infinite column bounds to -1.) Solve with the proposal $\bar{u}_1^k$.

1. If infeasible, generate infeasibility cut and add coefficients to matrix G^k and right-hand side g^k. Go to step 0 with $k \mapsto k+1$.
2. If optimal with solution $\bar{u}_2^k$ and $c_1\bar{u}_1^k + c_2\bar{u}_2^k \geq 0$, generate optimality cut and add coefficients to matrix F^k and right-hand side f^k. Go to step 0 with $k \mapsto k+1$.
3. Otherwise problem is unbounded. STOP.

4.2 The Decomposition Solver in OSLSE

The solver `ekkse_nestedLSolve()` in OSLSE implements a flexible, nested L-shaped method (King and Wright 2002). The implementation is *flexible* in the sense that subproblems may contain any number of nodes from the scenario tree. The implementation is *nested* in the sense that the L-shaped method may be applied to child subproblems.

Subproblem Generation. Subproblems are automatically generated in one of three ways. One, specify that the scenario tree be cut at an arbitrary time stage (`CUTATSTAGE`). Two, specify that no subproblem contain more than a given number of rows (`CUTBYROWSIZE`). Three, specify that there be no more than a fixed number of subproblems (`CUTBYMAXNODES`). Each way of generating subproblems has advantages. Decomposition methods tend to be slower than direct solvers for small problems, so it may make sense to generate subproblems of a size that is efficient for the underlying solver. Specifying the number of rows for each subproblem (`CUTBYROWSIZE`) controls the size of the subproblems directly. Specifying the number of subproblems (`CUTBYMAXNODES`) may affect the speed of convergence of the decomposition algorithm by reducing the communication overhead. But however desirable such properties may seem, the advantages could be undercut by having subproblems that are unbounded. OSLSE applies the "augmented root" construction when cutting at a timestage (`CUTATSTAGE`), in which the root subproblem is combined with the first leaf subproblem. This may bound the root subproblem and so speed up the convergence.

Solver Directives. The decomposition solver uses the OSL simplex solvers. The user may specify directives to the LP solvers, such as scaling or the use of a crash or presolve. If the problem type is *quadratic*, then OSL's quadratic solver will be invoked. A future implementation of OSLSE will allow advanced users to provide their own solver through the OSI solver interface (COIN-OR 2001).

Termination. The decomposition solver terminates if unboundedness or infeasibility has been detected, when the optimality test succeeds, when the maximum number of root iterations has been exceeded, or when no new cuts were generated for the incumbent proposal from the root subproblem. In the latter case, the progress of the algorithm has stalled with an optimality gap that is larger than desired by the user. In such a case the algorithm indicates that an optimal solution has been found, but reports the actual optimality gap attained. The user can specify that the decomposition solver be followed by a simplex solver. This problem will be hot-started from the solution found by the decomposition solver.

Access to Solutions. The user can retrieve or print primal and dual solutions by node and scenario, and also print the distribution of objective function values. The solution vectors are in the order determined by the core data.

4.3 The Parallel Decomposition Solver

OSLSE supports the running of the decomposition solver in parallel environments. This feature is automatically invoked by setting the parallel switch on `ekkse_setParallelOn()` before calling the decomposition solver. The parallel solver detects how many machines are in the computing environment and invokes a solver process called `OSLSE_Parallel` on each one. The subproblems are then divided among the solver processes and initialized. Each solver process loops over the subproblems assigned to it and performs whatever actions are required by the decomposition algorithm. Cuts, proposals, and subproblem status are passed using a message-passing infrastructure.

Currently the PVM (ORNL 1993) and MPI (ANL 1995) message-passing environments are supported. IBM Research is also considering implementing OSLSE's parallel solver in a grid computing environment (Globus 2001).

5 Stochastic Mixed Integer Problems

OSLSE supports a branch-and-bound algorithm for the solution of stochastic mixed-integer programs (Parija, Ahmed, and King 2002). Integer variables are declared either through the Core file or through the function `ekks_setIntegersAtCore()`. The type of the integer may be specified as binary, general integer, or SOS2.

Once the integers are specified, the function `ekks_markIntegers()` propagates the integer variables through the stochastic program following the convention that the Core problem specifies a template for the stochastic program. This will mark as integer every stochastic variable corresponding to a core variable marked as integer. (The user may override the markings by accessing the stochastic program as an EKKModel object and using OSL methods to change the integer variable settings.) The function `ekks_markIntegersWithStagePriorities()` influences the selection of branching variables. Finally the function `ekks_branchAndBound()` applies OSL's branch and bound mixed integer programming algorithm to the resulting stochastic program.

6 Miscellaneous Special Features of OSLSE

OSLSE is a library designed much like its parent OSL. All data is stored privately in a "`Stoch`" object. Public modules set/get parameters and data and perform various processes on the `Stoch` object. The basic uses of OSLSE are described in the public documentation (IBM 1998). In this section we choose to highlight a few special features of OSLSE.

SPL File. OSLSE stores the `Stoch` object in a binary formatted SPL file, which can be used to separate data input from repeated solution trials. At the conclusion of data input one can save the stochastic program to a permanent SPL file by calling the `ekks_writeNativeData()` module. The solver part may then begin by calling `ekks_readNativeData()`. This is an extremely efficient way to regenerate the `Stoch` object because data blocks in the SPL file can be copied directly to solver memory regions. The SPL file is a compact scenario description of the stochastic program, and so will be much smaller in size than the SMPS description using scenario format.

SMPS and MPS File Writers. It is sometimes useful to be able to debug stochastic programming data input logic by examining the SMPS files or even the MPS files for the deterministic equivalent LP. OSLSE includes functions `ekks_outMatrixSMPS` to write out the data in SMPS scenarios format and `ekks_outMatrixMPS` to write out the MPS file.

References

ANL (1995). MPI: the Message Passing Interface standard. Technical report, Argonne National Laboratory, `http://www-unix.mcs.anl.gov/mpi/index.html`.

Birge, J., M. Dempster, H. Gassmann, E. Gunn, A. King, and S. Wallace (1987). A standard input format for multiperiod stochastic programs. *Mathematical Programming Society Committee on Algorithms Newsletter 17*, 1–20.

Cariño, D., T. Kent, D. Meyers, C. Stacy, M. Sylvanus, A. Turner, K. Watanabe, and W. T. Ziemba (1994). The Russell-Yasuda Kasai model: an asset/liability model for a Japanese insurance company using multistage stochastic programming. *Interfaces 24*, 29–49.

COIN-OR (2001). Open solver interface. Technical report, Common Optimization Interface for Operations Research, `http://www-124.ibm.com/developerworks/opensource/coin/index.html`.

Condevaux-Lanloy, C., E. Fragniere, and A. King (2002). SISP, Simplified Interface for Stochastic Programming: Establishing a hard link between mathematical programming modeling languages and SMPS codes. *Optimization Methods and Software 17*(3), 423 – 443. Special Issue on Stochastic Programming. Guest Editors: A. Prekopa and A. Ruszczynski.

Entriken, R. (1989). The parallel decomposition of linear programs. Ph.D. Dissertation, Department of Operations Research, Stanford University, Stanford, California.

Globus (2001). The globus project. Technical report, The Globus Project, `http://www.globus.org/`.

IBM (1995). IBM Optimization Solutions and Library. Technical report, International Business Machines Corporation, `http://www-3.ibm.com/software/data/bi/osl/index.html`.

IBM (1998). IBM Optimization Solutions and Library: Stochastic Extensions. Technical report, International Business Machines Corporation, `http://www-3.ibm.com/software/data/bi/osl/features/stex.html`.

IBM (2001). Optimization Solutions and Library: Guide and Reference. World Wide Web, `http://www6.software.ibm.com/sos/features/libuser.htm`.

Kall, P. and S. W. Wallace (1994). *Stochastic Programming*. Chichester etc.: Wiley.

King, A. (1994). SP/OSL version 1.0 stochastic programming interface library user's guide. Technical report, Research Report RC19757, IBM Thomas J. Watson Research Center, Yorktown Heights, New York.

King, A. J. and S. E. Wright (2002). A flexible-partition, nested L-shaped method for linear programming. Submitted to *Mathematical Programming*.

ORNL (1993). PVM: Parallel Virtual Machine. Technical report, Oakridge National Laboratory, `http://www.epm.ornl.gov/pvm/`.

Parija, G., S. Ahmed, and A. King (2002). On bridging the gap between stochastic integer programming and mip solver technologies. Submitted to *INFORMS Journal of Computing*.

Van Slyke, R. and R. Wets (1969). L-shaped linear programs with applications to control and stochastic programming. *SIAM Journal on Applied Mathematics 17*, 638–663.

A Sample Input with Internal Arrays

This section illustrates the use of internal arrays to pass problem data on the `KandW` example, which is a modified version of a production planning problem from the text (Kall and Wallace 1994). A refinery can blend 2 raw materials into 2 different products. They must decide how much raw material to purchase and stock so that they can be blended to satisfy the demand for the products in two future time periods. The demand has to be completely satisfied, and in case of raw material shortage the products can be outsourced at a higher cost. There is an inventory constraint on how much raw material can be stocked in total. The algebraic statement of the problem is

$$\begin{array}{lll} \min_{x,y} & \sum_{t=1}^{3}\sum_{i=1}^{2} c_{it}x_{it} + E\sum_{t=2}^{3}\sum_{j=1}^{2} f_{jt}y_{jt} & \\ \text{s.t.} & \sum_{t=1}^{3}\sum_{i=1}^{2} x_{it} \quad \le b & \\ & \sum_{i=1}^{2} a_{ij}x_{it} + y_{jt} \quad \ge \tilde{d}_{jt} & t = 2,3 \quad j = 1,2 \end{array}$$

The sample problem driver for the data as given in the Samples section of the OSLSE guide and reference (IBM 1998), is as follows.

```
/* Core Model */
int ncol=8, nrow=5, nels=16;
/* Sparse matrix data */
int    mcol[]={1, 2, 3, 4, 1, 2, 3, 4, 1, 2,   3, 4,   5,6,7,8};
int    mrow[]={1, 1, 1, 1, 2, 2, 4, 4, 3, 3,   5, 5,   2,3,4,5};
double dels[]={1, 1, 1, 1, 2, 6, 2, 6, 3, 3.4, 3, 3.4, 1,1,1,1};
/* Objective */
double dobj[]={ 2.0, 3.0, 2.0, 3.0, 7.0, 12.0, 10.0, 15.0 };
/* Column bounds */
double dclo[]={ 0.0, 0.0, 0.0, 0.0, 0.0,  0.0,  0.0,  0.0 };
double dcup[]={ INF, INF, INF, INF, INF,  INF,  INF,  INF };
/* Row bounds */
double drlo[]={ -INF, 0.0, 0.0, 0.0, 0.0 };
double drup[]={ 50.0, INF, INF, INF, INF };
/* Stages */
int nstg=3;
int rstg[]={ 1,2,2,3,3 };
int cstg[]={ 1,1,1,1,2,2,3,3 };
/* Scenarios */
int nscen=9;
double dp[]={ .06, .15, .09, .12, .16, .12, .12, .12, .06 };
/* local variables */
int rc,is,ia,ib,ii;
EKKContext *env = ekks_initializeContext();
EKKStoch *stoch=ekks_newStoch(env,"KandWs",nscen);
/* Create Core */
rc=ekks_createCore(stoch,nstg,rstg,cstg,nrow,ncol,nels,
        dobj,drlo,drup,dclo,dcup,mrow,mcol,dels,0 );
```

```
/* Generate Scenarios using REPLACE_CORE_VALUES */
drlo[1]=200; drlo[2]=180; drlo[3]=200; drlo[4]=180;
for(is=0; is<nscen; is++){
    ia=0; ib=2;
    rc=ekks_addScenario(stoch,is+1,ia,ib,dp[is],1,
            nrow,ncol,nels,dobj,drlo,drup,dclo,dcup,
            mrow,mcol,dels,REPLACE_CORE_VALUES );
    ia=is+1; ib=3;
    for( ii=is+1; ii<is+3; ii++){
        drlo[3] -= 20; drlo[4] -= 20;
        rc=ekks_addScenario(stoch,ii+1,ia,ib,dp[ii],1,
                nrow,ncol,nels,dobj,drlo,drup,dclo,dcup,
                mrow,mcol,dels,REPLACE_CORE_VALUES );
    }
    drlo[3]=200; drlo[4]=180; drlo[1] -= 20; drlo[2] -= 20; is+=2;
}
/* Solve and test optimal value */
rc=ekks_describeFullModel(stoch,1);
rc=ekkse_nestedLSolve(stoch,CUTATSTAGE,0);
assert(fabs(ekk_getRobjvalue(ekkse_getCurrentModel(stoch))-2613.000)<0.001);
ekks_deleteStoch(stoch);
ekks_endContext(env);
```

B OSLSE Interfaces for Passing Problem Data

B.1 Core and Time Data

```
/* Pass Core model using in-core-memory data */
int ekks_createCore(EKKStoch *stoch,
        int nstg, int *rstag, int *cstag, /* stage info               */
        int irow, int icol, int iels,     /* # rows, cols, elements   */
        double *dobj,                     /* objective                */
        double *drlo, double *drup,       /* row bounds               */
        double *dclo, double *dcup,       /* col bounds               */
        int *mr, int *mc, double *dels,   /* matrix triples           */
        double *dqdg);                    /* diag quadratic terms     */

/* Set Core problem  type to LP or QP */
typedef enum {linear=1, quadratic=2} CoreType;
int ekks_setCoreProblemType(EKKStoch *stoch, CoreType type);
```

B.2 Stochastic Data

```
/* Define Scenario using in-core-memory data */
int ekks_addScenario(EKKStoch *stoch,
        int nscn,                           /* scenario number         */
        int ianc,                           /* # of ancestor scenario  */
        int istg,                           /* branching stage         */
        double dprob,                       /* probability             */
        int itype,                          /* matrix type (not used)  */
        int nrow, int ncol, int nels,
        double *dobj,
        double *drlo, double *drup,
        double *dclo, double *dcup,
        int *mrow, int  *mcol, double *dels,
        int replace );                      /* replace = ADD_TO_CORE_VALUES
                                              or        REPLACE_CORE_VALUES */
/* Reset probability weights */
int ekks_changeProbability(EKKStoch *stoch, double *dp);

/* Replace tree with samples from original scenario tree             */
/* samplemode =0 - uniform random sampling (the only supported mode)  */
int ekks_generateSamples(EKKStoch *stoch, int numsamples, int samplemode);
```

B.3 Generating Scenario Tree from Simulations

```
/* Initialize EKKTree structure for a maximum of mxsmp scenarios       */
/* and create first event from the array of data values                */
/* arr[] with narr entries.  The length of the arr[] array, narr,      */
/* is the maximum number of branch nodes between root and leaf.        */
EKKTree * ekks_createScenarioTree(int *arr, int narr, int mxsmp, double dp);

/* Add event to EKKTree object. Finds the event anc_arr[] with longest */
/* sequence max{k: arr[0] = anc_arr[0], ... , arr[k] = anc_arr[k]},    */
/* and adds the branch (arr[k+1] ... arr[narr-1]).                     */
int  ekks_addEventToTree(EKKTree *tree,int *arr, int narr, double dp);

/* Get the scenario branching information for event number nsmp.       */
/* The meaning of the returned data is the same as in ekks_addScenario */
int ekks_getScenarioFromTree(EKKTree *tree, int nsmp,
        int *ianc, int *istg, double *dprob, int *nscn,
        int **arr);                           /* points to array values */

/* Single function versions of ekks_getScenarioFromTree() */
int ekks_getScenarioAncestorFromTree(EKKTree *tree,int nsmp);
int ekks_getScenarioBranchStageFromTree(EKKTree *tree,int nsmp);
double ekks_getScenarioWeightFromTree(EKKTree *tree,int nsmp);
int ekks_getScenarioNumberFromTree(EKKTree *tree,int nsmp);
int ekks_getDataLengthFromTree(EKKTree *tree);
int* ekks_getScenarioDataFromTree(EKKTree  *tree,int nsmp);
```

C Reading and Writing SPL, SMPS and MPS files

```
/* Read or write using OSLSE Native Data model (very fast) */
int ekks_readNativeData(EKKStoch *stoch, char *splfilename);
int ekks_writeNativeData(EKKStoch *stoch, char *splfilename);

/* read SMPS data and return type of distribution in stoch file */
int ekks_readSMPSData(EKKStoch *stoch,
         const char *corefile, const char *timefile, const char *stochfile);
/* Generate SMPS files --- useful for debugging! */
int ekks_outMatrixSMPS(EKKStoch *stoch,
        int smpstype, /* SCENARIOS, BLOCK_DISCRETE, INDEP_DISCRETE */
        int  replace,  /* ADD_TO_CORE_VALUES, REPLACE_CORE_VALUES */
        const char *corefile, const char *timefile, const char *stochfile);
/* Generate MPS file */
int ekks_outMatrixMPS(EKKStoch *stoch, const char *mpsfile);
```

D OSLSE Interfaces for the Decomposition Solver

```
/* The Nested L-shaped Method solver decomposes using the cutstrategy:  */
#define CUTATSTAGE      1    /* equivalent to ekks_bendersLSolve   */
#define CUTBYROWSIZE    2    /* sets max # rows in subproblems     */
#define CUTBYMAXNODES   3    /* sets max # of subproblems          */

int ekkse_nestedLSolve(EKKStoch *stoch, int cutstrategy, int startmode);

/* Get/set cut period for cut-at-stage decomposition */
int   ekks_getCutPeriod(EKKStoch *stoch);
void   ekks_setCutPeriod(EKKStoch *stoch, int period);

/* Get/set number of rows for cut-by-row-size decomposition */
int  ekks_getMinNumRows(EKKStoch *stoch);
void  ekks_setMinNumRows(EKKStoch *stoch, int numrows);

/* Get/set number of models for cut-by-max-models decomposition */
int  ekks_getMaxsubmodels(EKKStoch *stoch);
void  ekks_setMaxsubmodels(EKKStoch *stoch, int maxmodels);
```

D.1 Solver Directives

```
/* set direction of optimization */
void  ekks_setMinimize(EKKStoch *stoch);
void  ekks_setMaximize(EKKStoch *stoch);
```

```
/* Get/set maximum number of major iterations in decomposition */
int ekks_getMaxiter(EKKStoch *stoch);
void ekks_setMaxiter(EKKStoch *stoch, int maxiter);

/* Get/set gap used to assess convergence of decomposition. */
double OSLLINKAGE ekks_getRelOptimalityGap(EKKStoch *stoch);
void ekks_setRelOptimalityGap(EKKStoch *stoch, double tol);

/* Set scaling on/off for subproblems in decomposition. */
/* It is usually a good idea to set it on. */
void ekks_setScaleOn(EKKStoch *stoch);
void ekks_setScaleOff(EKKStoch *stoch);
/* Set OSL presolve type on/off for subproblems. */
void ekks_setPresolve(EKKStoch *stoch,int presolvetype);
void ekks_setPresolveOff(EKKStoch *stoch);
/* Set OSL crash type on/off for subproblems */
void ekks_setCrash(EKKStoch *stoch, int crashmode);
void ekks_setCrashOff(EKKStoch *stoch);

/* Set OSL Simplex solver type for subproblems. */
void  ekks_setSimplexAlg(EKKStoch *stoch, int alg);

/* After decomposition solve/don't solve with simplex solver */
void ekks_setFinalSimplexSolverOn(EKKStoch *stoch);
void ekks_setFinalSimplexSolverOff(EKKStoch *stoch);

/* Set large bounds on/off (may affect speed of decomposition). */
void ekks_setLargeBoundsOn(EKKStoch *stoch);
void ekks_setLargeBoundsOff(EKKStoch *stoch);
```

D.2 Access to Solutions

```
/* Solution Status
   (0 - optimal, 1 - infeasible, 2 - unbounded, >=3 - incomplete) */
int ekks_getStatus(EKKStoch *stoch);
/* Objective Value */
double ekks_getRobjvalue(EKKStoch *stoch);

/* Getting and Printing Solutions */
/*   mode = 0 get column solution */
/*   mode = 1 get row solution    */

/* OSLSE sorts the matrix.  index[] shows the internal index number. */
int ekks_getNodeSolution(EKKStoch *stoch, int scenario, int stage,
        int mode, double *solution, int *index);
int ekks_getScenarioSolution(EKKStoch *stoch, int scenario,
        int mode, double *solution, int *index);
int ekks_getNodeDualSolution(EKKStoch *stoch, int scen, int stg,
        int mode, double *solution, int *index);
int ekks_getScenarioDualSolution(EKKStoch *stoch, int scen,
        int mode, double *solution, int *index);

/* Printing */
int ekks_printObjectiveDistribution(EKKStoch *stoch);
int ekks_printNodeSolution(EKKStoch *stoch, int scen, int stg,
         int mode);
int ekks_printNodeDualSolution(EKKStoch *stoch, int scen, int stg,
         int mode);

/* Stochastic LP dimensions */
int ekks_estimateLPSize(EKKStoch *stoch);
int ekks_getNumScenarios(EKKStoch *stoch);
int ekks_getNumStages(EKKStoch *stoch);
int ekks_getNumNodes(EKKStoch *stoch);
int ekks_getCoreNumcols(EKKStoch *stoch);
int ekks_getCoreNumrows(EKKStoch *stoch);
int ekks_getNumcolsAtStage(EKKStoch *stoch,int stg);
int ekks_getNumrowsAtStage(EKKStoch *stoch,int stg);
```

E OSLSE Interfaces for Stochastic Mixed Integer Problems

```
/* ****
ekks_setIntegersAtCore marks a set of columns in the core model as integers.
The integer array intType[numints] specifies the types of each integer
(1 - binary, 2 - general integer, 3 - SOS2, 0 - continuous).
**** */
int ekks_setIntegersAtCore(EKKStoch *stoch,
        int numints, int *intnums, int *intType);

/* propagate integer variables through EKKStoch and return total */
int ekks_markIntegers(EKKStoch *stoch);
/* propagate with priorities and pseudocosts */
int ekks_markIntegersWithStagePriorities(EKKStoch *stoch,
        int *stagePriority, double *upperPseudoCost, double *lowerPseudoCost);

/* solve with branch and bound */
int ekks_branchAndBound(EKKStoch *stoch,
        const char *matrixFile, const char *basisFile);
```