SP/OSL Version 1.0
Stochastic Programming Interface Library
User's Guide

Alan J. King
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

kingaj@watson.ibm.com

October 13, 1994

TABLE OF CONTENTS

1. INTRODUCTION

This User's Guide to SP/OSL (Stochastic Programming Interface to the Optimization Subroutine Library) contains examples and basic information needed for the use of SP/OSL subroutines and data structures. Appendices to the User's Guide contain complete documentation of SP/OSL's interfaces and options.

SP/OSL is a library of portable C-language subroutines that support the modeling, construction and solution of stochastic programs. SP/OSL manages multistage stochastic program data structures, freeing the user to concentrate on modeling issues and/or algorithm development.

Version 1.0 of SP/OSL is an experimental software library developed over the past four years by Alan King and colleagues at IBM's Thomas J. Watson Research Center. Stephen E. Wright (now at Miami University of Ohio) was co-developer of SP/OSL during his postdoc years 1991-3 and a summer visit in 1994. The elegant SP-file object and the flexible and robust implementation of the Benders decomposition were entirely designed and written by him. Robert Entriken (now a consultant with PG&E) wrote the first version of the code, a matrix generator, in early 1990. The important node/arc metaphor basic to SP/OSL data structures is due to him. Hercules Vladimirou (now at University of Cyprus) developed and tested an early parallel implementation of single-cut Benders during his postdoc in 1991-2. Chris Donohue (University of Michigan) developed the prototype bundling software during his stint as a summer student in 1992. Special acknowledgement also is due to Roger Wets, for inspiration, guidance and support during his frequent visits throughout the development period.

SP/OSL is now being distributed under special agreement with principal investigators at selected universities and corporate research centers for field testing. We hope that the feedback from field testing will help SP/OSL to become truly useful for supporting stochastic programming activity in scientific research and industrial applications. We hope especially that SP/OSL will be of service to those in the stochastic programming community who are designing and implementing solution methods, and to those in the model-languages area who are interested in supplying stochastic program modeling facilities to their user communities.

The features of SP/OSL may be summarized as follows.

1. SP/OSL makes no assumptions about the ordering of variables or constraints in a model; it can be used with any modeling language or matrix generator that can define a linear program.
2. SP/OSL places no restrictions on the probability distribution except that it be described by a finite number of sample paths, or *scenarios*, of LP data. (SP/OSL implements the scenarios version of the stochastic programming standard input format [1].)
3. SP/OSL organizes data into a natural tree structure in a specially formatted *SP-file*. SP-files are self-contained and can be spawned from any subtree of the original, facilitating the implementatation of parallel solution methods.
4. SP/OSL data structures are transparent to the user. The user may freely access all data and construct solution methods using natural stochastic programming data structures.
5. SP/OSL is a complete stochastic programming package, from model input to solution output. SP/OSL implements powerful decomposition methods using Optimization Subroutine Library (OSL) [4] facilities, with numerous options to fine-tune for particular problem types.
6. SP/OSL consists entirely of portable C language subroutines callable from C or Fortran programs.

1.1. Getting and Installing.

SP/OSL comes as a tar file `sposl.tar` on diskette or via electronic mail. After transferring the file (as a binary file) to an appropriate directory, type the following commands:

```
% tar -xvf sposl.tar
% sh sposl
```

The shell script `sposl` will unpack the library, header files, examples and documentation.

The subdirectories `./lib` and `./cmn` contain the library and header files. When you wish to compile programs using SP/OSL facilities, you must include these directories in the linker and compiler paths, respectively.

The subdirectory `./examples` contains sample problems and drivers. Follow the directions in the `Readme` file to make the examples. You must have installed the *Optimization Subroutine Library* [4] to use these examples. Source data and descriptions of problem examples may be found in the `./data` subdirectory.

2. STOCHASTIC PROGRAMMING

Stochastic programming is a methodology for solving decision problems under uncertainty. SP/OSL implements "multistage stochastic linear recourse programs with scenario distributions". This section explains what that mouthful means. For more detailed discussions, see [1], [3] and [8]. The "scenario distribution" is a general discrete-time finite-state stochastic process. The basic idea is to model this as a tree; this will be motivated by a consideration of the two-stage case.

2.1 Two-Stage Stochastic Programming

A two-stage stochastic linear recourse program may be formulated as:

$$\begin{aligned} \text{minimize}_{x_1} \quad & c_1 x_1 + \sum_{s \in S} p_s Q_1(x_1; s) \\ \text{subject to} \quad & A_{11} x_1 = b_1 \\ & x_1 \geq 0 \end{aligned} \tag{2.1}$$

where for each scenario the *recourse function* $Q_1(\cdot;\cdot)$ is computed as the solution of the recourse problem:

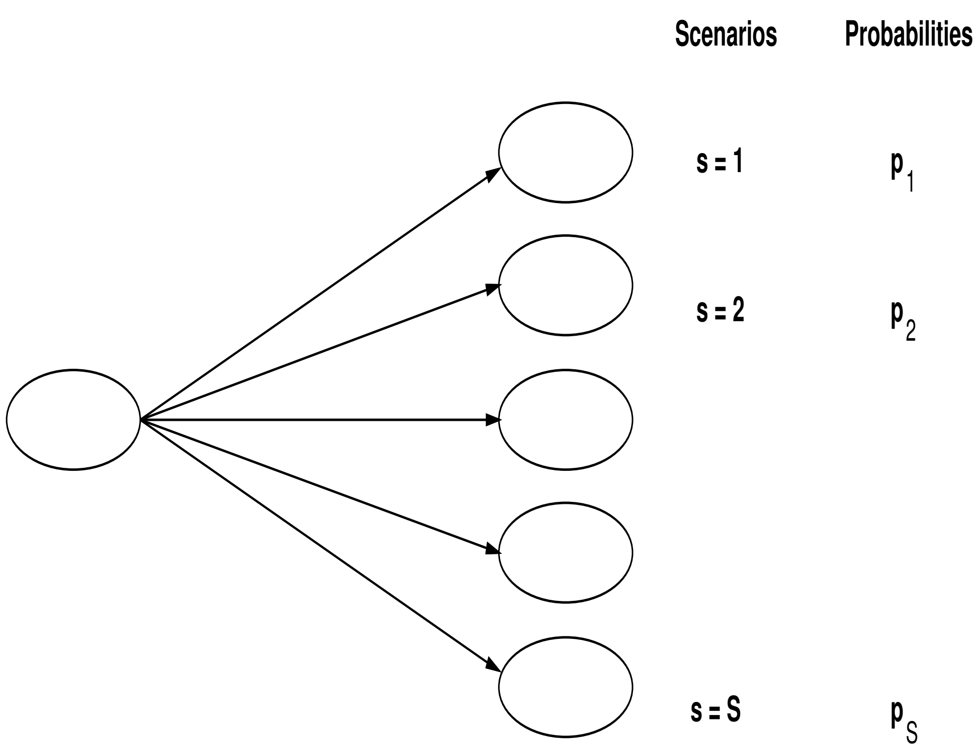
$$\begin{aligned} Q_1(x_1; s) = \text{minimize}_{x_2} \quad & c_2(s) x_2 \\ \text{subject to} \quad & A_{22}(s) x_2 = b_2(s) - A_{21}(s) x_1 \\ & x_2 \geq 0 \end{aligned}$$

The first-stage decision, x_1, incurs costs $c_1 x_1$ and satisfies constraints $A_{11} x_1 = b_1$, and $x_1 \geq 0$. At a later stage, an event s occurs and the values of certain parameters in the second-stage objective c_2, right-hand side b_2, and matrices A_{21}, A_{22} are revealed. A subsequent "recourse" decision x_2 is made in order to satisfy the second stage constraints $A_{22} x_2 = b_2 - A_{21} x_1$ and incurs costs $c_2 x_2$. The unknown parameters of the second stage are discretely distributed, taking values $[c_2(s), b_2(s), A_{21}(s), A_{22}(s)]$ with probability p_s, $s = 1, \ldots, S$.

The key point is that the first-stage decision x_1 must be made *before* information about the scenario s is revealed, in such a way as to hedge against the different scenarios that may arise. The expected value of the recourse function provides the incentive to hedge; it evaluates the quality of the first-stage decision by computing the expected cost of responding to the scenarios $s = 1, \ldots, S$.

The recourse problem can have many interpretations beyond that of "expected recourse cost". A common formulation is that of *simple recourse*, where $c_2 = [e, e]$ and $A_{22} = [I, -I]$; e and I are respectively the unit vector and identity matrix of dimension equal to that of b_2. Simple recourse "tracks" uncertain targets $b_2(s)$ with uncertain technology $A_{21}(s)x_1$; the tracking risk is expressed as the expectation of the sum of the absolute deviations. Setting $c_2 = [0, e]$ gives a "one-sided" tracking risk that penalizes the expected value of the sum of the below-target deviations. (Such models can be interpreted as approximate utility maximization; see the reference [6] which discusses tracking models in a financial setting.)

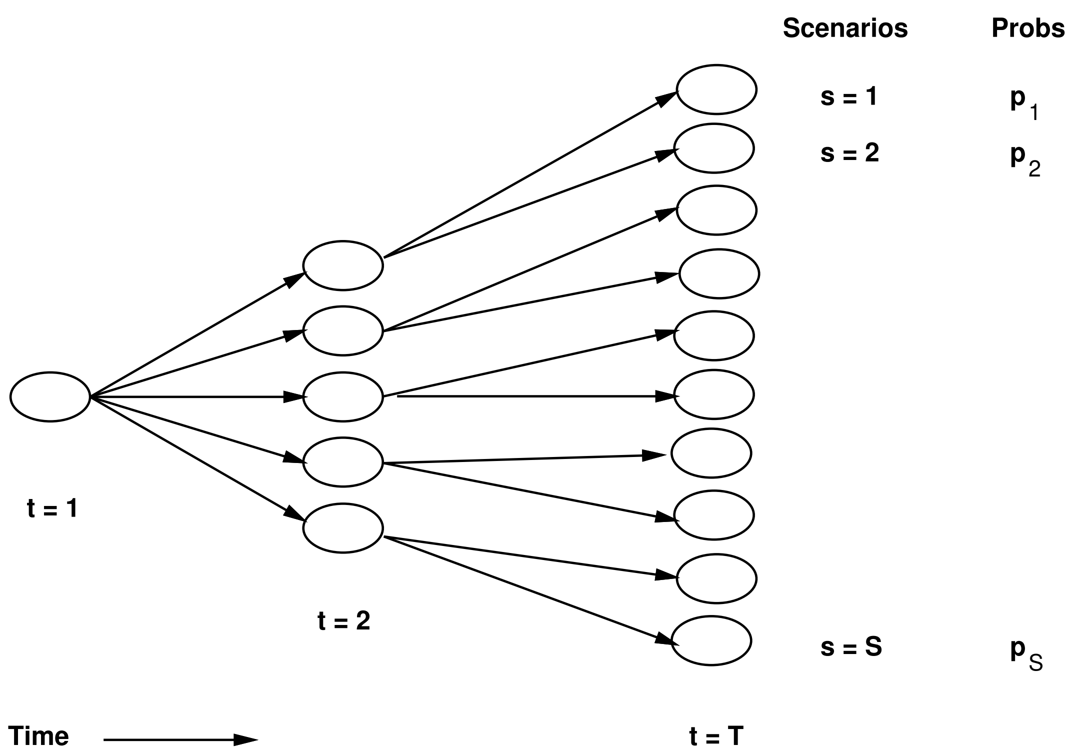
The structure of a two-stage stochastic program can be modeled by a tree. The root node represents the first-stage problem, and the leaf nodes represent the alternative second-stage problems for each scenario s. The arcs connecting the root to the leaves model the probability distribution as viewed from the root node; that is, from the root node there is only a probability that a given leaf node s will occur, namely p_s. The tree models the essential property of decision-making under uncertainty; information about scenarios is revealed over time.



2.2 Multistage Stochastic Programming

The information structure of a multistage stochastic program is modeled by a multi-level tree, with nodes arranged in layers corresponding to time stages $t = 1, \dots, T$. Every node except the root has a single "parent" node in the previous stage; all nodes except leaves have child nodes representing the conditional distribution of possible events occuring in the next stage. A scenario corresponds to a single path from root to leaf; each scenario terminates in a unique leaf node.

In a given collection of scenarios, nodes may be identified with the "bundle" B_t of scenarios passing through them. Let the symbol $\mathcal{B}_t$ denote the collection of all bundles B_t at each time stage $t = 1, \dots, T$. The collection $\mathcal{B}_1$ at the first-time stage contains a single member B_1 which consists of all the scenarios $s = 1, \dots, S$; all scenarios pass through the root node. The collection $\mathcal{B}_T$ at the final time stage contains S members each of which contains a single scenario, since only one scenario "passes through" a leaf node. Of course, every scenario must pass through some node at stage t, so the union of the bundles B_t in $\mathcal{B}_t$ is the set of all scenarios.



In the tree depicted above, for example, the collection $\mathcal{B}_2$ of stage 2 bundles consists of 5 members. The first member bundle contains scenarios 1 and 2, the second contains scenarios 3 and 4, and so forth.

Scenarios passing through the same node-bundle are indistinguishable at that time stage. Thus, the data for the decision problem at a given node, including the history of past decisions, are *identical* for each scenario passing through the node. Furthermore, since the decisionmaker cannot distinguish between scenarios in a node-bundle, the feasible solutions $x_t(s)$ to a multi-stage stochastic program must satisfy

$$x_t(s) = x_t(s') \quad \forall s, s' \in B_t, \quad \forall B_t \in \mathcal{B}_t, \quad \forall t = 1, \ldots, T$$

These conditions are known as the *nonanticipativity* constraints, and are collectively denoted by: $x \in \mathcal{N}$. They express the intuitive notion that decisions cannot anticipate the future. Another way of expressing the nonanticipativity constraints is

$$x_t(s) = E\{x_t(s) \,|\, \mathcal{B}_t\}(B_t) \quad \forall s \in B_t, \quad \forall B_t \in \mathcal{B}_t, \quad \forall t = 1, \ldots, T$$

where for a random variable $Y = (Y_1, \ldots, Y_T)$, the conditional expectation $E\{Y_t(s) \,|\, \mathcal{B}_t\}$ is a quantity which has a distinct value for each bundle B_t

$$E\{Y_t(s) \,|\, \mathcal{B}_t\}(B_t) = \sum_{s \in B_t} \left(\frac{p_s}{\sum_{s \in B_t} p_s} \right) Y_t(s)$$

i.e, the average over B_t of $Y_t(s)$ conditioned on being in B_t.

The stochastic program (2.1) is extended to multiple stages by defining the recourse function Q_t for $t = 1, \ldots, T-1$ recursively as follows:

$$\begin{aligned} Q_t(x_1, \ldots, x_t; s) = \min \;\; & c_{t+1}(s)\, x_{t+1} \\ & \qquad + E\{Q_{t+1}(x_1, \ldots, x_t; s) \,|\, \mathcal{B}_{t+1}\} \\ \text{s.t.} \;\; & A_{t+1\,t+1}(s)\, x_{t+1} \\ & \qquad = b_{t+1}(s) - \sum_{t' < t+1} A_{t+1\,t'}(s)\, x_{t'} \\ & x_{t+1} \geq 0 \end{aligned} \tag{2.2}$$

Note that the recourse term Q_{t+1} is evaluated at nodes in stage $t+1$ by averaging the recourse cost over all scenarios in each node-bundle $B_{t+1} \in \mathcal{B}_{t+1}$.

The final-stage problem Q_T is similarly defined, except it has no recourse terms:

$$
\begin{aligned}
Q_T(x_1, \ldots, x_T; s) = \min \;\; & c_T(s)\, x_T \\
\text{s.t.} \;\; & A_{T\,T}(s)\, x_T = b_T(s) - \sum_{t' < T} A_{T\,t'}(s)\, x_{t'} \\
& x_T \geq 0
\end{aligned}
\tag{2.3}
$$

The formulation (2.1-3) highlights the role of the recourse function. It is the basis for algorithms based on a cutting plane approach (Benders decomposition, or the L-shaped method), where a sublinear representation of the recourse function Q_1 is generated by accumulating cuts. See the articles in [3] for details.

The stochastic program (2.1-3) is equivalent to the following extended linear program:

$$
\begin{aligned}
\min \quad & \sum_{s=1}^{S} \sum_{t=1}^{T} c_t(s)\, x_t(s)\, p_s \\
\text{s.t.} \quad & A_{tt}(s)\, x_t(s) = b_t(s) - \sum_{t' < t} A_{tt'}(s)\, x_{t'}(s) \\
& x_t(s) \geq 0 \\
& x \in \mathcal{N}
\end{aligned}
\tag{2.4}
$$

where the first and second groups of constraints hold for every stage $t = 1, \ldots, T$ and every scenario $s = 1, \ldots, S$. This version highlights the role of the nonanticipativity constraints, and is the starting point for algorithms based on relaxing nonanticipativity (see the description of the Progressive Hedging algorithm in [8]).

The data of a stochastic program is structured according to node bundles. This structure is only implicit in (2.1-3) and (2.4). The version of the stochastic program implemented in SP/OSL, called a *stochastic model*, is explicitly organized by node:

$$
\begin{aligned}
\min \quad & \sum_{t=1}^{T} \sum_{B_t \in \mathcal{B}_t} c_t(B_t)\, x_t(B_t)\, p(B_t) \\
\text{s.t.} \quad & A_{tt}(B_t)\, x_t(B_t) = b_t(B_t) - \sum_{t' < t;\; B_{t'} \supset B_t} A_{tt'}(B_t)\, x_{t'}(B_{t'}) \\
& x_t(B_t) \geq 0
\end{aligned}
\tag{2.5}
$$

where the "node probability" $p(B_t)$ is the sum of the probabilities p_s passing through the node-bundle B_t. The statement $B_{t'} \supset B_t$ for $t' < t$, asserts that the node-bundle $B_{t'}$ is a direct *ancestor* of the node-bundle B_t. Note that the nonanticipativity constraints are implicitly satisfied.

To implement (2.5), SP/OSL explicitly organizes the stochastic data by node-bundle:

$$[c_t(B_t), b_t(B_t), A_{t1}(B_t), \ldots, A_{tt}(B_t)]$$

The diagonal matrix A_{tt} is called the node matrix, and the off-diagonal matrices $A_{t1}, \ldots, A_{t\,t-1}$ are called "arc" matrices (they bring the ancestor decisions forward to the stage t problem).

This form of the stochastic program is most convenient for the sorts of manipulations required for algorithm development in stochastic programming. Subproblems that might be needed in an implementation of the L-shaped or Progressive Hedging algorithms, for instance, are generated in SP/OSL by specifying nodelists. A linear program in the form of (2.5) is then generated, with special care taken in the calculation of the node probabilities and the ancestor relationships.

The stochastic program (2.5) makes no assumptions concerning the form of the distributions. Thus (2.5) and the associated data stuctures serve as a general starting point for any implementation of a stochastic programming algorithm.

2.3 Building Scenario Trees from Sample Paths

In any realistic application, the potential number of nodes will be huge. A simple scheme has been implemented in SP/OSL for combining sample paths from a stochastic process into an empirical scenario tree (see section 3.6). This section presents a brief mathematical description of this procedure. (A paper on this work is under preparation [7].)

Denote the data process $[c_t, b_t, A_{t1}, \ldots, A_{tt}]$ by $\boldsymbol{\xi}_t$ — this is a discrete-time multidimensional stochastic process over $t = 1, \ldots, T$. Suppose that one can generate ν sample paths of $\boldsymbol{\xi}$, say $\{\boldsymbol{\xi}^\iota\}_{\iota=1}^{\nu}$.

For each time stage, partition the range of $\boldsymbol{\xi}_t$ into a finite number of sets, say B_t. Let $\mathcal{B}_t$ be the collection of all the sets in the partition. Of course, B_t is a node-bundle for the stochastic process, in the sense that every sample path of $\boldsymbol{\xi}$ must pass through one of the nodes B_t in $\mathcal{B}_t$.

Each of the sample paths $\{\boldsymbol{\xi}^\iota\}_{\iota=1}^{\nu}$, in particular, must pass through one of the nodes. For each sample path $\boldsymbol{\xi}^\iota$ add probability $1/\nu$ to the node probability $p(B_t)$ for every node B_t, $t = 2, \ldots, T$ that it passes through. (The node probabilities are initialized at zero.) The resulting tree is an empirical approximation to the bundled stochastic process, and may be used as a consistent approximation to the original stochastic program.

3. OVERVIEW OF SP/OSL

The motivating principle of SP/OSL is to free the user from the complex task of managing multistage stochastic data structures common to all stochastic programs, while making minimal assumptions concerning the style of modeling and no assumptions for preference of solution method.

To implement this aim, SP/OSL contains several independent clusters of subroutines corresponding roughly to initialization and input, tree generation, model formation and data transfer, solution via Benders decomposition, and user-access to data and information.

All information concerning a particular instance of a stochastic program is organized as a structured data type called `Stoch_Data`. This structure is initialized by a call to subroutine `scninit`, which returns a pointer to the `Stoch_Data` data type. Every SP/OSL call uses this pointer to access the data of the stochastic program. Multiple instances of stochastic programs can appear in the same program module, each with their own unique `Stoch_Data` pointer.

The generic SP/OSL driver consists of five subroutine calls:

```
#include "usrstoc.h"
Stoch_Data *stoch_data;
scninit(&rc,&stoch_data,...);
scncore(&rc,stoch_data,... );
for(s=0;s<max_s;s++)
{
    scnadsc(&rc,stoch_data,s,...);
}
scnsolv(&rc,stoch_data,osl_dspace,...);
for(s=0;s<max_s;s++)
{
    scnsoln(&rc,stoch_data,s,soln,...);
    printf( ... ,soln);
}
```

The call to `scninit` initializes the `stoch_data` pointer to the `Stoch_Data` object. The calls to `scncore` (declare core file) and `scnadsc` (add a scenario) pass the problem data to the `Stoch_Data` object. A call to `scnsolv` declares and solves the stochastic problem with a Benders decomposition method implemented in OSL. The subroutine `scnsoln` accesses scenario solutions one-by-one after which they may be processed by the user (for example, printed).

3.1 Scenarios Data Structure.

SP/OSL Version 1.0 implements the "scenarios" version of the standard input format published in [1]. The standard input format views a stochastic program as being defined through deviations from a core linear program. The core problem outlines the dimensions of the *decision stages* of the stochastic program: how many rows/columns belong to each stage. Scenarios provide the information about the probability distribution and the data needed to define the actual stochastic program (see section 2).

In the standard format, scenarios are declared one at a time by specifying the node from which it branches, the probability to be attached to the leaf, and the scenario data for the time stages between the branch and the leaf.

Scenario data is defined relative to the core data, by the convention that *all scenario data is added to the corresponding core data.* For details of this convention, see the documentation of the `scnadsc` subroutine. The scenarios standard is the only stochastic programming data format implemented in SP/OSL at this time.

3.2 Organization of Storage: the SP-file.

When stochastic program core and scenario data is passed to SP/OSL, the data, consisting of matrix, row and column bounds, and objective function, is sorted into stages and stored in a specially-structured *SP-file.* Core data is stored first in the file, then the scenario data. A copy of the core data is retained in memory, but all scenario data is left in the SP-file until needed by an optimization method. Binary file data storage and retrieval is remarkably fast in the AIX 3.2 operating system; the user will not notice much delay attributable to data movement between this file and core memory. For communicating stochastic program data between processors, or for repetitive numerical work, the use of this storage/retrieval facility is strongly recommended.

SP-file storage is managed by SP/OSL routines. Users may have access to this information through special purpose user-access routines. The SP-file may be saved (`scnptsd`), in which case it will contain all information necessary to restore (`scngtsd`) the stochastic program.

3.3 Stochastic Models.

The data corresponding to a stochastic program is organized by nodes and arcs of a tree structure and stored in an SP-file. Many stochastic programs could be defined from the same data—for example, the data defined by one single scenario is itself a linear program complete with objective and bounds.

SP/OSL organizes the data required for the definition of a stochastic program into a structure of type `Stoch_Model`. A `Stoch_Model` can be thought of as a selection of a subset of nodes from a `Stoch_Data`. (They are not the same sort of structure, since the node data for the submodel is obtained by reference to the original `Stoch_Data` structure.) A stochastic model is created by passing a *list of nodes* to the appropriates SP/OSL subroutine `scnlmdl` or `scnemdl`. By default, a `Stoch_Data` structure is considered to be itself a stochastic model containing all possible nodes. The `Stoch_Data` pointer is substitutable for the `Stoch_Model` pointer in many SP/OSL subroutine calls.

The user can define any possible subtree of the original stochastic data by specifying an appropriate nodelist. The conventions for forming the subtree from a nodelist are as follows:

1. All incoming arcs for every node in the list are included in the stochastic model.
2. All nodes that originate arcs in the model but that do not appear in the original nodelist are appended to the original nodelist and marked as "virtual".

The treatment of *virtual* nodes depends on the method of solution. In some cases, for example, if the nodelist defines an entire scenario subset, there will be no virtual nodes. In Benders decomposition, for another example, virtual nodes are the source of proposals and the destinations of cuts. The treatment of virtual nodes will be further discussed below in the section on defining models in OSL. The user may access information about the `Stoch_Model` structure, and use this information to implement their own solution methods.

SP/OSL provides some special subroutines for creating stochastic models of certain types, for example: given a list of probabilities, some of which may be zero, adjust probabilities in the model (`scncprb`); given a time stage, create the list of nodelists describing the subtrees generated by cutting all arcs incoming to nodes belonging to that stage (`scnpcut`).

3.4 Solution using `scnsolv`.

For those interested in using the modeling facilities of SP/OSL to construct and solve a stochastic program, SP/OSL provides a special subroutine `scnsolv()` to solve a stochastic model using a simple version of Benders decomposition. (See the example drivers `solv.c`, `sosl.c`, and `sdst.c` in the appendix.)

The user may set various parameters controlling the performance of the `scnsolv` procedure by editing the file `scnsolv.defaults`, to be described below.

The solution procedure consists of a Benders decomposition phase, and if needed, a Simplex method phase. (Benders decomposition often does not terminate in a reasonable number of iterations, for this reason it is often useful to follow Benders with the Simplex method.) In the Benders decomposition phase, the problem is decomposed into master and subproblems at a user-specified stage, the `Cut_Stage`. All nodes belonging to stages strictly less than the `Cut_Stage` are put into the master problem. (For this reason, the `Cut_Stage` should not be less than 2.) The nodes belonging to the `Cut_Stage` become "root" nodes of subtrees extending to the leaf nodes. These subtrees form the nodelists for the subproblems of the Benders decomposition method.

If there are more subtrees than the maximum number of subproblems allowed, `Maximum_Subproblems`, then the subtrees are divided among the subproblems as evenly as possible. The first subproblem is joined to the master problem, thus the master consists of the nodes less than `Cut_Stage` together with the subtrees collected into the first subproblem. This "augmented master" tends to produce better proposals; for example, the master nodes might be unbounded without the influence of some downstream nodes.

Benders cuts are generated for each remaining subproblem. Changing the number `Maximum_Subproblems` can have a big effect on the performance of Benders decomposition. For example, 100 subproblems will generate 100 different cuts, whereas 1 subproblem will generate 1 cut from the same proposal. The difference is the level of aggregation: the 1 cut from 1 subproblem is in fact the aggregation of the 100 cuts from the 100 subproblems. Generally speaking, aggregated cuts contain less information than the collection of disaggregated cuts, but disaggregated cuts make the master problem grow much faster.

Major iterations of Benders decomposition are performed until either a relative optimality gap of size less than `Optimality_Gap` is attained, or the maximum number of iterations `Benders_Major_Iterations` is performed. A major iteration consists of:

(1) Solving the master with all optimality and feasibility cuts so far generated, and posting the proposed master solution and master objective value, if one exists.

(2) Solving all subproblems with the master proposal and generating subproblem optimality or feasibility cuts. If all subproblems generate optimality cuts, then the subproblem optimal values are summed to form the subproblem objective value.

(3) Testing the relative gap between the sum of the subproblem objective values (the true value at the given master proposal) and the master objective value (a lower bound) for convergence to within `Optimality_Gap`.

At the termination of the Benders decomposition phase, the solution is placed into the OSL model for the full stochastic program. (The OSL model is automatically declared by a call to `scnemdl()`, but no data is generated.) If optimality was not attained, then the Simplex phase begins.

Experience suggests that the solution progress is most affected by the maximum number of major interations, the cut stage, and the maximum number of subproblems. It seems that it is best to run about 3 major iterations of Benders decomposition and then pass to the Simplex phase. The setting of the `Cut_Stage` controls the size of the subproblems and the size of the master. Large subproblems and masters take a long time to solve. A large number of small subproblems generating disaggregated cuts for the master does not seem to present much difficulty for OSL. The form of the `scnsolv.defaults` file is as follows:

```
Maximum_Subproblems          10
Cut_Stage                    3
Benders_Major_Iterations    60
Optimality_Gap               1.0e-6
EKKNWMT_Type                 1
Perform_EKKSCAL              0
EKKCRSH_Method               3
Perform_EKKPRSL              0
EKKPRSL_Type                 0
EKKPRSL_Unit                 0
EKKSSLV_Algorithm            1
```

Those parameters not yet described affect to the solution procedures applied within the Benders decomposition. These procedures are OSL subroutines, and the parameters refer to the calling sequences documented in the OSL manual. A method or type setting of zero means that OSL will not be called to invoke the procedure. For example, the parameter `EKKCRSH_method` refers to the parameter passed to the OSL routine `EKKCRSH`, which is called at the initial solution of any subproblem, master or subproblem. If this parameter is set to zero, then `EKKCRSH` would not be called (not recommended!). The actual instantiation is `if (icrsh) ekkcrsh(iret,osl_dspace,icrsh);`, and similarly for the other procedures.

3.5 Defining Stochastic Models to OSL

SP/OSL subroutines provide a simple interface to the Optimization Subroutine Library (OSL). Stochastic models may be declared as models in OSL, and then solved by OSL routines.

The user must write a driver that combines SP/OSL and OSL calls to accomplish this. Defining a stochastic model to OSL requires two steps. First, a call to `ekkdsca` causes OSL to initialize its working array, here called `osl_dspace`. Then the SP/OSL routine `scnlmdl` declares a model in OSL from a given nodelist and passes the node and arc data over to `osl_dspace`. The user has the option of declaring or not declaring the matrix block to OSL. In the latter case, the user would have invoked `scnemdl` instead of `scnlmdl`. The block management facility of OSL is used to place the nodes and arcs into the OSL model. In particular, core matrix blocks are replicated using the `ekkrptb` subroutine. As explained above, every node (except the root) consists of core and stochastic data. This data is added together as it is placed into the OSL data structures (except the matrix blocks, whose addition is implied by OSL's convention of adding matrix elements with the same row and column offsets). Node probabilities for the stochastic model are computed and the node objective is multiplied by this probability prior to being placed in OSL.

Virtual nodes have no LP data present in the model, but columns for them are declared. The columns belonging to the virtual nodes are those from arcs that originate at the virtual node and terminate at some "real" node in the model. The column bounds and objective values associated with the virtual nodes are all set to zero.

The SP/OSL structure keeps track of where nodes were placed by storing the row and column *offset* given to the node block as it was placed in the model. Once a solution has been obtained, SP/OSL routines can retrieve the solution. The OSL model so created is accessible to the user, as are the node offsets. The user can, if desired, take control of the optimization process at this point. (It is important to note that the rows and columns may have been sorted in the input phase, so the original locations of particular variables may have changed. The sort keys are accessible to the user. The user can employ the subroutine `scnsoln` to retrieve scenario solutions resorted back to the original order.)

3.6 Building Scenario Trees from Simulations.

SP/OSL includes an independent set of routines for the purpose of building a scenario tree out of a sequence of simulations. These subroutines have the prefix "`tree_`", and are require the inclusion of the header file `usrtree.h` for their use.

Each simulation is viewed as an array of *events*, labelled as integers. When a new simulation is added (`tree_add`), its initial sequence is compared against all initial sequences so far received. The longest match found is recorded, and the new simulation is made to branch from the scenario it matched. Weights of unmatched nodes are initialized at one, and weights of the matched nodes are incremented by one. The weights are eventually used to calculate the empirical probability of a node. The result is a scenario tree that describes the empirical distribution of the sequence of simulations. The scenario tree data, together with scenario numbers, branching node, and probability weights can be retrieved (subroutine `tree_get`); and the resulting conditional distributions can be examined (subroutine `tree_dist`). (The first version of these subroutines was written by Chris Donohue, of the University of Michigan.)

3.7 User Access to SP/OSL Data Structures.

The SP/OSL user interfaces, data structures and subroutine declarations, are declared in the header file `usrstoc.h`. This file should be included in every program module that calls SP/OSL subroutines.

The data pointed to by the structures `Stoch_Data` and `Stoch_Model` is internal to SP/OSL and protected from the user. But all stochastic programming data is available to the user via the structures `usrCoreData`, `usrNodeData`, `usrNode`, and `usrStoch`, etc, as defined in the `usrstoch.h` header file. These must be filled by special subroutine calls: `scncdat`, `scnsget` and `scnndat`. The drivers `sdst.c`, `solv.c`, and `sosl.c` contain examples of user data access.

Calls to SP/OSL usr access subroutines can have a `Stoch_Data` or a `Stoch_Model` pointer in the argument list. In the latter case, the list of nodes returned will be those that appeared in the model's original list, plus the list of appropriately marked virtual nodes. Offsets and probabilities refer to the relevant model.

The node data retrieved by a call to `scnndat` contains only the stochastic data. It must be combined with the appropriate core data obtained by `scncdat`. It is important to note is that the data may have been sorted during the input phase. Particular items may have been moved out of their original positions. The `usrCoreData` structure has pointers to the sort keys used to translate the row and column indices from external (i.e. user) order to internal, and back.

3.8 Standard Format: Stochastic MPS Input Files

SP/OSL provides subroutines to read the stochastic MPS format defined in [1]. In this format, the core problem is first read in as an MPS formatted file (for example, by the OSL subroutine `ekkmps`). The *time* file is processed to partition the rows and columns into stages by SP/OSL subroutine `scntmps`. (The row and column indices in the core file must be ordered by stage.) Scenario data is provided in the *stochastic* file; this is read by SP/OSL subroutine `scnsmps`.

Scenario data may also be provided by `scnadsc`. Multiple calls to `scnsmps` are possible, just as for `scnadsc`.

3.9 Parametric Solution

SP/OSL includes facilities for defining a parametrized perturbation vector and for successively solving a stochastic program as the parameter is incremented. The SP/OSL routines are the counterparts of the OSL subroutines `ekkpmdl` and `ekkspar`. Subroutine `scnpmdl` declares the perturbation vector and parameter interval/increments and subroutine `scnspar` performs the parametric optimization.

The perturbation vector is defined on the core model row bounds and objective. Subroutine `scnpmdl` accepts the perturbation vector, sorts it by stage, and replicates it for every node of the problem, with node probabilities multiplying the objective portion of the perturbation. Subroutine `scnspar` adds the parameter increment times the perturbation to each node and resolves the problem starting from the previously obtained solution.

The user may write a user-callout function `report` to process the solutions obtained as `scnspar` progresses. `Report` is called at the conclusion of each optimization in `scnspar`.

4. EXAMPLE DRIVERS

The examples in the following sections illustrate the use of some of the options for input, solution and data access available in the SP/OSL library. The C-language coding for the examples may be found in the `./examples` directory. Follow the Readme instructions to make the examples. This code is also reproduced in the appendix.

All the input examples create an SP-file `<file>.spl`, where the string `<file>` is input from the command line (except `sgbd`, which creates a file called `sgbd.spl`). The solver examples restore the stochastic program from `<file>.spl`, then proceed to solve the problem. For example, to solve the problem `app0110` found in the `./data` directory in SMPS format, execute the commands:

```
% smps data/app0110
% solv data/app0110
```

4.1 Generic Driver with scnadsc

The driver `sgbd` implements the stochastic Aircraft Allocation problem published by G.B. Dantzig in [2]. (See also the exposition of this example and many other examples in [5].) It is a simple recourse problem with a total of 750 scenarios derived from 5 independent random variables, representing demand for aircraft seats. The coding demonstrates the use of `scnadsc`. To generate the problem and solve it, execute the commands:

```
% sgbd
% solv sgbd
```

The call to `sgbd` produces the SP file `sgbd.spl`, and the call to `solv` solves the stochastic program.

4.2 SMPS Input

The driver `smps` reads the "scenarios" input format [1]. To use the driver, invoke it as follows:

```
% smps data/app0110
```

The application name `app0110` is one of the applications found in the `./data` directory. The `smps` call produces the SP file `../data/app0110.spl`.

4.3 Direct Solution by `scnsolv`

Driver `solv` restores the stochastic program from the SP-file `<file>.spl` and solves it using `scnsolv`. For instance,

```
% solv data/app0110
```

solves the problem generated by the invocation of `smps` in the example given in section 4.2.

Subroutine `scnsolv` reads options from a file `scnsolv.defaults`. Depending on the options settings given, `scnsolv` calls `scnemdl` to create an *empty* full model stochastic program. Then the Benders decomposition routine `scnbndr` decomposes the problem at a given stage into a given number of subproblems and iterates until a given number of iterations or a given tolerance is reached; `scnsolv` then loads this solution into the appropriate OSL arrays. If `scnbndr` terminates with an optimal solution, then `scnsolv` returns. Otherwise, `scnsolv` calls `scndsmt` to load the full model matrix blocks into OSL arrays and calls `ekksslv` to finish. (The user should note that `scndsmt` is not called if `scnbndr` terminates with an optimal solution.)

All output of `solv` is directed to standard output.

4.4 Direct Solution by OSL Routines

Driver `sosl` restores the stochastic program from the SP-file `<file>.spl` and then solves the problem using `ekksslv`. For instance,

```
% sosl data/app0110
```

solves the problem generated by the invocation of `smps` in the example given in section 4.2. The subroutine `scnlmdl` performs the two actions `scnemdl` (declare OSL model) and `scndsmt` (pass model data to OSL). Then the OSL model is solved.

All output of `sosl` is directed to standard output.

4.5 Post Solution Processing

Driver `sdst` restores the stochastic program from the SP-file `<file>.spl` and solves the problem using `scnsolv`. For instance,

```
% sdst data/app0110
```

solves the problem generated by the invocation of `smps` in the example given in section 4.2. The optimal value along each scenario is then computed by using the user-access routines `scncdat`, `scnsget` and `scnndat`, and the results are displayed as a distribution.

All output of `sdst` is directed to standard output.

4.6 Simple Decomposition

This example is not yet ready.

4.7 Efficient Frontier using `scnspar`

This example is not yet ready.

REFERENCES

[1] J.R. Birge, M.A.H. Dempster, H.I. Gassmann, E.A. Gunn, A.J. King, and S.W. Wallace, "A standard input format for multiperiod stochastic programs", *Math. Prog. Soc. Committee on Algorithms Newsletter* **17** (1987) 1–20.

[2] G.B. Dantzig, *Linear Programming and Extensions*, (Princeton University Press, 1963).

[3] Yu. Ermoliev and R. Wets, editors, *Numerical Techniques for Stochastic Optimization* (Springer Verlag, 1988).

[4] *Optimization Subroutine Library Guide and Reference* (International Business Machines Corporation, SC23-0519-02, 1991).

[5] A.J. King, "Stochastic programming problems: examples from the literature", in Yu. Ermoliev and R. Wets, *op cit.*

[6] A.J. King, "Asymmetric risk measures and tracking models for portfolio optimization under uncertainty", *Annals of O.R.* **45** (1993) 165–177.

[7] A.J. King and R.J-B Wets, "Epiconsistency for multistage stochastic programs", (in preparation; IBM Thomas J. Watson Research Center, 1994).

[8] R.T. Rockafellar and R.J-B Wets, "Scenarios and policy aggregation in optimization under uncertainty", *Math. of O.R.* **16** (1991) 119–147.

APPENDIX A Description of Data Structures

SP/OSL internal data structures are not accessible to the user. Instead, the user can gain access to copies of the structures in the format described in the file `usrstoc.h`. The function calls to access the data are `scncdat` (get usrCoreData structure), `scnsget` (get usrStoch structure) and `scnndat` (get usrNode data). This section describes the data structures.

usrCoreData. The usrCoreData structure contains all information about the core model and its time stage decomposition. It is accessed by a call to the subroutine `scncdat`.

```
typedef struct {
     long nrow;           /* number of rows */
     long ncol;           /* number of columns */
     long nels;           /* number of matrix elements */
     long nstg;           /* number of stages */
     long nblocks;        /* number of blocks in matrix */
     double *drlo,*drup;  /* row bounds */
     double *dclo,*dcup;  /* column bounds */
     double *dobj;        /* objective */
     long *mrow;          /* matrix row indices */
     long *mcol;          /* matrix column indices */
     double *dels;        /* matrix elements */
     long *rowstart;      /* starts for stage partition of rows */
     long *colstart;      /* starts for stage partition of cols */
     long *elemstart;     /* starts for stage partition of matrix
                             Matrix is partitioned by row blocks
                             and then by column blocks
                             thus t*(t+1)/2-1 is the matrix index for
                             the start of the t-th stage node block
                             (as would be obtained by numbering blocks
                             starting from the top and moving left to
                             right).  The matrix is assumed full lower
                             block triangular. */
     long *sortin2exr;    /* in-to-external row index translation */
     long *sortin2exc;    /* in-to-external col index translation */
} usrCoreData;
```

The three arrays `rowstart`, `colstart` and `elemstart` identify the starting indices for each time stage in the list of row, column, and matrix entry indices respectively. Index numbering for matrix elements always begins from one (for compatibility with OSL).

For example, if

$$\texttt{rowstart} \quad = [1, 5, 11]$$

then the row indices $1, 2, 3, 4$ are first stage rows, and $5, 6, 7, 8, 9, 10$ are second stage rows. The first four entries in the row bounds arrays `drlo` and `drup` are the bounds for the first stage rows, and the second six entries are the second stage row bounds. Of course, the ordering here may not correspond to the user's original ordering in the calling sequence for the `scncore` subroutine that created the core data structure. The array `sortin2exr` provides the original row indices. Thus, the first element of `sortin2exr` gives the original index for the first row, etc. Similar comments hold for the column arrays `colstart` and `sortin2exc`.

The `elemstart` array has a more complex structure. The core matrix is sorted first into row blocks, within which each row belongs to the same stage, and then each row block sorted into column blocks, within which each column belongs to the same stage. For example, if the problem has 4 stages, then this method sorts the matrix into lower block triangular form with up to ten blocks in the following order:

$$\begin{pmatrix} 1 & \cdot & \cdot & \cdot \\ 2 & 3 & \cdot & \cdot \\ 4 & 5 & 6 & \cdot \\ 7 & 8 & 9 & 10 \end{pmatrix}$$

In this case the `elemstart` array has 11 entries. For example, if

$$\texttt{elemstart} \quad = [1, 6, 11, 20, 20, 26, 35, 35, 35, 41, 50]$$

then the diagonal block 6 has 9 entries starting at the 26th entry of the matrix triple `(mrow, mcol, dels)`, and the off-diagonal blocks 4, 7 and 8 have no entries at all. The matrix triple is stored in "by indices" format; thus the 26th entry has value `dels[25]` (using the indexing convention of the C programming language) row index `mrow[25]` and column index `mcol[25]`.

usrStoch. The `usrStoch` structure identifies the nodes of a stochastic model. The `usrStoch` and `usrNode` structures are filled by a call to `scnsget`.

```
typedef struct {
     long nnode;        /* number of nodes */
     usrNode *node;     /* array of node structures */
     long nleaf;        /* number of leaves */
```

```
    long *leaf;       /* list of leaf positions in node array (beg=1) */
    void *user;       /* (optional) pointer to user structure */
} usrStoch;
```

The nodes are described by an array of `usrNode` structures. The leaves of the tree are identified by the `leaf` array. For example, if one has

$$\texttt{leaf} \quad = [4, 8, 9]$$

then there are three scenarios in the model and the terminal nodes of the scenarios are described by the 4th, 8th, and 9th entries in the `node` array. A pointer is provided to point to a user structure, if needed (this information is for the user's purposes only; at this time it cannot be stored or retrieved by SP/OSL).

usrNode. The `usrNode` structure describes all the information relating to a node in a scenario tree.

```
typedef struct {
    double prob;     /* probability of node in model */
    long orig;       /* original node name (numbered from 1) */
    long par;        /* parent's position in list (beg=1) */
    long child;      /* next-stage child (end=0) */
    long sib_cous;   /* same-stage sibling or cousin (end=0) */
    long stag;       /* original time stage */
    long scen;       /* original scenario number */
    long rbas;       /* row offset in matrix */
    long cbas;       /* column offset in matrix */
    long nrow;       /* number of rows */
    long ncol;       /* number of columns */
    long virtual;    /* = 1 if node not actually in model */
    void *user;      /* (optional) pointer to user structure */
} usrNode;
```

A node points to all information and data corresponding to a given time stage along a given scenario. The `prob` field contains the node's probability in the stochastic model. This probability multiplies the node's objective in the stochastic model. The `orig` field identifies the node's original numbering in the `Stoch_Data` structure, and must be used when accessing the LP data for the node.

The next three fields allow the user to move around the tree. The numbers refer to the ordering in the `node` array of the `usrStoch` structure. The `par` field identifies the node's parent, that is, the node from which it branched in the stage immediately preceding. If the node happens to be a root, then its `par` field is set equal to zero. The `child` field identifies one of the children of the node. If there are no children (i.e. this is a leaf node) the field is set to zero. The `sib_cous` field identifies one sibling (that is, sharing a common parent) or cousin (merely share the same time stage). The `sib_cous` fields of nodes sharing a common time stage are arranged in a *ring*. All children of a node `node` can be found by following the `sib_cous` fields of the `child`, checking to see that they all share the same parent. All nodes of a given stage can be found by following the `sib_cous` fields around until the starting point is reached again.

The `stag` field identifies the node's stage and the `scen` field the original scenario number. The two fields `rbas` and `cbas` are the row and column bases that give the node's offset in the (deterministic equivalent) stochastic model. Finally, the `virtual` field identifies whether the node was (=0) or was not (=1) included in the nodelist which generated the stochastic model. Nodes whose descendents are in a model but that do not themselves appear in the nodelist generating the model need to be identified as such when constructing the LP for the stochastic model, so that offsets may be allocated for the virtual node columns. A pointer is provided to point to a user structure, if needed (this information is for the user's purposes only; at this time it cannot be stored or retrieved by SP/OSL).

usrNodeData. The `usrNodeData` structure describes the *stochastic* LP data of a node. The data is accessed by *original* node number, as found in the `usrNode` structure describing the node (in the field `orig`), by the subroutine `scnndat`.

```
typedef struct {
     long stg;               /* stage of node ( = number of partitions) */
     long nrow;              /* number of rows */
     long ncol;              /* number of columns */
     long nels;              /* number of node block elements */
     double *drlo,*drup;     /* row bounds */
     double *dclo,*dcup;     /* column bounds */
     double *dobj;           /* objective */
     long *mrow;             /* block row indices */
     long *mcol;             /* block column indices */
     double *dels;           /* block elements */
     long *elemstart;        /* starts for stage partition of node block:
```

```
                                     thus t-1 is the index for the start of the
                                     t-th stage arc block and stg-1 is the
                                     index for the start of the node block.
                                     The node block has (poss.  empty) arc
                                     blocks for all stages t < istg. */
} usrNodeData;
```

This data must be combined with the core data to construct the actual node LP data. The convention in SP/OSL is to *add* the core and stochastic data.

When a user accesses node data, the incoming arc matrices are copied over in the same subroutine call. This should be taken into consideration when transferring large amounts of data from the SP-file into memory.

The matrix associated with a node is the stochastic *row-block*; that is, all the stochastic row data in the given time stage in the given scenario are found in this matrix. Of course, there may not be any elements in the stochastic row-block. The matrix is structured in similar fashion as that of the matrix in the `usrCoreData` stucture, that is, it has been sorted into column blocks, by stage. The column indices in each of the column-blocks have been normalized to begin with index 1. The actual column index of an off-diagonal block is found by locating the node's ancestor for that stage and adding the ancestor's column offset `cbas` to the matrix column index. The actual column index of the diagonal block is found by adding the current node's column offset `cbas` to the column index. The actual row index of all entries is found by adding the current node's row offset `rbas`.

APPENDIX B Description of Scn Subroutines

SCNADSC. Add a scenario to the stochastic data structure.

```
extern void scnadsc( long *rc, Stoch_Data *stoch,
                     long nscn, long ianc, long istg,
                     double dprob, long itype,
                     long nrow,long ncol,long nels,
                     double *dobj, double *drlo, double *drup,
                     double *dclo, double *dcup,
                     long *mrow, long *mcol, double *dels );
/* scnadsc:
 *     Adds a scenario to stochastic problem.  Sets the scenario
 *     probability, the branching information, and stores the scenario data
 *     by node in the SP-file associated to the Stoch_Data structure.
 *
 *     LP data should be the "difference" of the intended scenario LP data
 *     and the core LP data.  The LP matrix must be in "indices" sparse
 *     format (see the OSL manual for a discussion of sparse formats).
 *
 *     The LP data is NOT kept in core memory.  It may be retrieved for
 *     the user by a call to scnndat.
 *
 *     Errors occur when branching information is inconsistent
 *     or when dimensions are inconsistent with the core.
 *
 *     If this call is repeated for the same scenario, the scenario
 *     probability is incremented by the amount of the new probability.
 *     No other data is changed.
 *
 *     All scenario LP data is ADDED to the corresponding core data when
 *     forming an LP.  This requires a convention for dealing with infinite
 *     bounds in the core.  If the core value is infinite and the incoming
 *     scenario value is nonzero, then a warning message is issued and the
 *     incoming scenario value is set to zero.
 *
 *     CORE VALUE | SCENARIO VALUE | ACTION             | MESSAGE | RC
 *     finite     | infinite       |  none              | none    |  0
 *     infinite   | nonzero        |  scen value zeroed | warning |  1
 * INPUT:
 *    Stoch_Data *stoch  - pointer initialized by scninit
 *    long nscn          - scenario number (1,2,...)
 *    long ianc          - ancestor scenario number
 *    long istg          - branching stage number
 *    double dprob       - probability of scenario
```

```
 *    long itype         - matrix type (MUST be 1 - by indices format)
 *    long nrow          - number of rows (must match core model)
 *    long ncol          - number of columns (must match core model)
 *    long nels          - number of elements in scenario matrix
 *    double *dobj       - array of objective entries
 *    double *drlo       - array of row lower bound entries
 *    double *drup       - array of row upper bound entries
 *    double *dclo       - array of column lower bound entries
 *    double *drup       - array of column upper bound entries
 *    long *mrow         - array of matrix row indices
 *    long *mcol         - array of matrix column indices
 *    double *dels       - array of matrix entries
 *
 * OUTPUT:
 *  long *rc           - return code (2 is fatal)
 *
 * RETURNS:
 *    void
 */
```

SCNBNDR. Solve the problem using Benders decomposition.

```
extern void scnbndr( long *rc, Stoch_Model *oModel, long ipcut,
                     long maxmod, long maxits,
                     long inwmt, long icrsh, long ilpdc,
                     long iscal,
                     long iprsl, long iprsltype, long iprslunit,
                     long isalg, long *optimal, double optgap );
/* scnbndr:
 *
 *  Applies Benders decomposition to the stochastic model oModel.
 *  The cut is made at the stage number IPCUT.
 *
 * Methodology:
 *
 *  A single master problem and up to MAXMOD subproblems are created,
 *  corresponding to a partition of the scenario tree made at stage IPCUT.
 *  The master problem contains all rows and columns for stages prior
 *  to IPCUT.
 *
 *  These problems are created in the osl_dspace of oModel.  If MAXMOD
 *  is not sufficient to hold all the subproblems, the subproblems are
 *  distributed as evenly as possible among MAXMOD models.
 *
```

```
*  If large_bounds is set, then bounds of the order of 10Î6 are added
*  to the master.
*
*  The program performs at most MAXITS iterations of Benders decomposition,
*  terminating if optimality, infeasibility, or unboundedness is
*  detected.
*
*  At termination, the master and subproblem solution data is
*  transferred to the OSL model for oModel as appropriate, and the
*  local structures are freed.
*
*  EKKSSLV (. ,. ,1,3) is the appropriate continuation.
*
* INPUT:
*
*  Stoch_Model *oModel  - stochastic model created by scnlmdl or scnemdl.
*  long ipcut           - period at which Benders cuts are made
*  long maxmod          - maximum number of subproblems
*  long maxits          - maximum number of master iterations
*  long inwmt           - EKKNWMT type (0 = none EKKNWMT) for all problems
*  long iscal           - (0,1) means (don't, do) call EKKSCAL for submodels
*  long icrsh           - EKKCRSH type (0 = none) for 1st solution of submodels
*  long ilpdc           - (0,1) means (don't, do) call EKKLPDC for submodels
*  long iprsl           - (0,1) means (don't, do) call EKKPRSL for submodels
*  long iprsltype       - type of presolve
*  long iprslunit       - presolve unit
*  long isalg           - EKKSSLV algorithm for first solution of submodels
*  long large_bounds    - (0,1) means (don't, do) add large bounds to master
*  double optgap        - Benders terminates when gap is less than optgap
*
* OUTPUT:
*
*  long *rc          - return code >= 2 indicates fatal error.
*  long *optimal       - (0,1,2) means (nonterminated, optimal, non-optimal).
*
*/
```

SCNCDAT. Get the core data into `usrCoreData` structure.

```
extern void scncdat( long *rc, Stoch_Data *stoch, usrCoreData **pcore );
/* scncdat
 *
 *   Copy the core data pointers of the Stoch_Data structure "stoch"
 *   into the usrCoreData structure.
```

```
 *   If *pcore is NULL, then a new usrCoreData structure is malloc'ed
 *   and the pointer returned.
 *
 *   The data pointed to by usrCoreData pointers is NOT a copy of the
 *   core data.  It is the original data as formed by scncore.
 *
 *  INPUT:
 *   Stoch_Data *stoch     - pointer initialized by scninit and scncore
 *
 *  OUTPUT:
 *   usrCoreData **pcore   - address of pointer to type usrCoreData.
 *
 */
```

SCNCORE. Create core model.

```
extern void scncore( long *rc, Stoch_Data *stoch,
                     long nstg, long *Rstag, long *Cstag,
                     long irow, long icol, long iels,
                     double *dobj, double *drlo, double *drup,
                     double *dclo, double *dcup,
                     long *mrow, long *mcol, double *dels );
/* scncore
 *
 * Sorts core model row and column indices by stages, and groups
 * core model data into blocks corresponding to stages.  Stores
 * all model information, sort keys, and grouped data into a
 * SP file.
 *
 * The data is held in core memory, and may be accessed by a
 * scncdat call.
 *
 * INPUT:
 *
 *     Stoch_Data stoch   - pointer initialized by scninit
 *     long nstg          - number of stages in the core model
 *     long *Rstag        - array of stage numbers for rows
 *     long *Cstag        - array of stage numbers for columns
 *     long irow          - number of rows in core model
 *     long icol          - number of columns in core model
 *     long iels          - number of matrix elements in core model
 *     double *dobj       - array of objective entries
 *     double *drlo       - array of row lower bound entries
 *     double *drup       - array of row upper bound entries
```

```
 *     double *dclo        - array of column lower bound entries
 *     double *drup        - array of column upper bound entries
 *     long *mrow          - array of matrix row indices
 *     long *mcol          - array of matrix column indices
 *     double *dels        - array of matrix entries
 *
 * OUTPUT:
 *
 *     long *rc      - return code (2 is fatal).
 *
 * RETURNS:
 *     void
 */
```

SCNCPRB. Change scenario probabilities.

```
extern void scncprb( long *rc, Stoch_Data *stoch, double *usrProb );
/* scncprb:
 *
 *    Updates leaf node probabilities with new scenario probabilities.
 *
 *    Note: The original scenario probabilities are unaffected.
 *          These can be restored by a call to scncprb with a NULL
 *         usrProb parameter.
 *
 * INPUT:
 *     Stoch_Data *stoch   - pointer initialized by scninit.
 *     double *usrProb     - points to array of new probabilities.
 *
 * OUTPUT:
 *     long *rc            - return code (2 is fatal)
 *
 * RETURNS:
 *      void
 *
 */
```

SCNDSCM. Declare stochastic model.

```
extern void scndscm( long *rc, Stoch_Data *stoch,
          long nnodes, long *usrndlist,
          Stoch_Model **smodel, long *nblocks, long calcprob );
/* scndscm
```

```
 *
 *     Creates model node information structures and offsets from
 *     user-supplied nodelist.  If usrndlist is empty (nnodes=0)
 *     then all nodes in the Stoch_Data structure are included.
 *
 *     Node probabilities are inherited from Stoch_Data structure.
 *     Zero probability nodes (and their incoming arcs) are removed.
 *
 *     Virtual nodes (ancestors not in model one of whose children are)
 *     are recorded and are offset by the total number of real node columns.
 *
 *     If calcprob is set, the probabilities for each node are
 *     recomputed and normalized so that the probability totals to one.
 *
 *     No matrix data is passed.  (This is done by scnlmdl or scndsmt)
 *     The number of blocks to be passed is returned.
 *
 * INPUT:
 *     Stoch_Data *stoch      - stochastic data
 *     long nnodes            - number of nodes
 *     long *usrndlist        - list of nodes to be included in model
 *     long calcprob          - (0,1) means (don't, do) recompute
 *                               node probabilities.
 *
 * OUTPUT:
 *     Stoch_Model **smodel   - pointer to new stochastic model
 *     long *nblocks          - number of blocks in model
 *
 * RETURNS:
 *     void
 *
 */
```

SCNDSMT. Define data for stochastic program in OSL.

```
extern void scndsmt ( long *rc, Stoch_Model *smodel );
/* scndsmt
 *
 *   Passes matrix, costs and bounds data to OSL.
 *   SCNEMDL must already have been called.
 *
 *   This subroutine is called by SCNLMDL.
 *
 * INPUT:
```

```
 *   Stoch_Model *smodel   - stochastic model declared by scndscm.
 *
 * OUTPUT:
 *   long *rc              - return code (2 is fatal).
 *
 * RETURNS:
 *   void
 *
 */
```

SCNEMDL. Declare empty stochastic model to OSL.

```
extern void scnemdl( long *rc, Stoch_Model *smodel, double *dspace,
                     long imodel );
/* scnemdl:
 *
 *   Link OSL array "dspace" with Stoch_Model object.
 *   Create OSL model "imodel", but do not pass data.
 *
 * INPUT:
 *   Stoch_Model *smodel   - stochastic model declared by scndscm
 *   double *dspace        - array initialized by OSL.
 *   long imodel           - OSL model initialized by EKKDSCM
 *
 * OUTPUT:
 *   long *rc              - return code (2 is fatal)
 *
 * RETURNS:
 *   void
 *
 */
```

SCNFREE. Free stochastic data structures.

```
extern void scnfree( Stoch *stoch);
/* scnfree:
 *
 *  Free all data structures and temporary files created by SP/OSL routines.
 *  (The OSL data structures are unaffected by this call.)
 *
 *  If a Stoch_Model structure is passed, then only structures created
 *  for this model are deleted.
 *
```

```
 * INPUT:
 *  Stoch *stoch        - Stoch_Model or Stoch_Data pointer to be deleted
 *
 */
```

SCNGTHR. Gather submodel solutions.

```
extern void scngthr( long *rc, Stoch_Model *oModel,
                  Stoch_Model *sModels[], long nmodels );
/* scngthr:
 *
 *   Gathers solution vectors and status vectors from OSL submodels
 *   created by scnpcut into original OSL model.
 *
 * INPUT:
 *   Stoch_Model *oModel     - model declared to OSL by scnemdl or scnlmdl.
 *   Stoch_Model *sModels[]  - submodels generated by scnpcut and
 *                             declared to OSL by scnemdl or scnlmdl
 *   long nmodel             - number of submodels.
 *
 * OUTPUT:
 *   long rc                 - return code (2 is fatal)
 *
 * RETURNS:
 *   void
 *
 */
```

SCNGTSD. Get stochastic data from SP file.

```
extern void scngtsd( long *rc, Stoch_Data *stoch, const char *filename );
/* scngtsd:
 *
 *   Copies Stoch_Data structures from SP file "filename" into core
 *   memory pointed to by "stoch".
 *
 * INPUT:
 *   Stoch_Data *stoch   - stochastic data pointer initialized by scninit.
 *   char *filename      - name of SP file.
 *
 * OUTPUT:
 *   long *rc            - return code (2 is fatal)
 *
```

```
 * RETURNS:
 *   void
 *
 */
```

SCNINIT. Initialize stochastic data structure.

```
extern void scninit( long *rc, Stoch_Data **stoch, long maxscn );
/*  scninit:
 *
 *     Allocates pointers for stochastic data structures,
 *     opens SP file, returns Stoch_Data pointer into **stoch.
 *     The SP file is created with the tmpnam() utility.
 *
 *  INPUT:
 *     long maxscn              - maximum number of scenarios
 *     Stoch_Data **stochdata    - pointer to stochastic data structure
 *
 *  OUTPUT:
 *     Stoch_Data *stochdata     - stochastic data structure
 *     long *rc                 - return code (2 is fatal)
 *
 *  RETURNS:
 *     void
 *
 */
```

SCNLMDL. Declare and describe stochastic model to OSL.

```
extern void scnlmdl( long *rc, Stoch_Model *smodel, double *dspace,
                     long imodel );
/* scnlmdl:
 *
 *  Declare stochastic model to OSL and pass data.
 *  Combines the operations of scnemdl and scndsmt.
 *
 * INPUT:
 *   Stoch_Model *smodel  - stochastic model declared by scndscm
 *   double *dspace       - array initialized by OSL.
 *   long imodel          - OSL model initialized by EKKDSCM
 *
 * OUTPUT:
 *   long *rc             - return code (2 is fatal)
```

```
 *
 * RETURNS:
 *   void
 *
 */
```

SCNMSET. Set SP/OSL messages on/off.

```
extern void scnmset( long *rc, long msgid, long onoff);
/* scnmset.c
 *
 *  Turn SP/OSL messages on/off.
 *  Message msgid (use symbolic names from message table) is turned off
 *  if onoff is 0, turned on if onoff is 1.
 *
 * INPUT:
 *   long msgid      - Message identifier (from message table).
 *   long onoff      - 0 turns off printing, 1 turns on printing
 *
 * OUTPUT:
 *   long *rc        - return code (2 is fatal)
 *
 * RETURNS:
 *   void
 */
```

SCNNDAT. Access node data.

```
extern void scnndat( long *rc, Stoch_Data *stoch, long node,
                     usrNodeData **punode );
/* scnndat:
 *
 *   Get stochastic node data into usrNodeData structure.
 *
 *   Allocates new usrNodeData structure.
 *   Accesses stochastic matrix and costs/bounds data from SP file
 *   and places them in volatile working memory.  Costs and bounds are
 *   expanded to their dense form.  usrNodeData pointers are appropriately
 *   set.
 *
 *   NB: ALL matrix and costs/bounds arrays are volatile!  The next call
 *       to an SP/OSL routine WILL OVERWRITE THEM!  (To protect
 *       yourself, copy them to a memory region that you maintain.)
```

```
 *   NB: The matrix and costs/bounds arrays contain only the stochastic
 *       part of the data.  They must be ADDED to the relevant core data.
 *   NB: returns NULL pointers for root node (node=1)
 *
 * INPUT:
 *   Stoch_Data *stoch   - stochastic data pointer initialized by scninit
 *   long node           - node number to be accessed
 *   usrNodeData **punode  - pointer to usrNodeData
 *
 * OUTPUT:
 *  usrNodeData *punode   - usrNodeData structure
 *  long *rc              - return code (2 is fatal)
 *
 * RETURNS:
 *  void
 */
```

SCNPCUT. Cut model at given stage for Benders.

```
extern void scnpcut( long *rc, Stoch_Model *oModel,
                     long ipcut, long maxmodels,
                     long ***nodlst, long **nnod, long *nmodels );
/* scnpcut:
 *
 *   Performs a Benders cut on smodel before stage ipcut, creating node
 *   lists for at most maxmodels problems.  The first list is for the root
 *   subproblem, and consists of all (nonvirtual) nodes having stage less
 *   than ipcut.  The rest of the nodes are distributed among the leaf
 *   subproblems in such a way that the difference in the number of branches
 *   allocated between any two subproblems is no more than one.
 *
 *   To Do:  Should allocate so that the number of leaves in each
 *           subproblem are almost equal (not always possible).
 *
 *   The lists (nodelists, numbers of nodes in each nodelist) are
 *   malloc'ed in permanent storage.
 *
 *
 * INPUT:
 *
 *     Stoch_Model oModel   - Stoch_Model to be cut
 *     long ipcut           - period at which model is cut
 *     long maxmodels       - maximum number of models (master + subs)
 *
```

```
 * OUTPUT:
 *     long ***nodlst       - 2-d integer array of nodelists for each submodel
 *     long **nnod          - number of nodes in each submodel
 *     long *nmodels        - number of submodels generated
 *     long *rc             - return code (2 is fatal)
 *
 * RETURNS:
 *     void
 */
```

SCNPMDL. Define parametric stochastic model in OSL.

```
extern void scnpmdl( long *rc, Stoch_Model *smodel,
                     double *cdelta, double *lrdelta, double *urdelta,
                     double *lcdelta, double *ucdelta, long mask );
/*
 * scnpmdl.c  Create parametric stochastic model
 *
 *  A cover for EKKPMDL.  The user supplies arrays which are to be
 *  used to perturb the objective and bounds of the core model.
 *  These then are sorted and replicated to produce perturbations
 *  for the stochastic model (objective perturbations are multiplied
 *  by the node probabilities).  The core perturbation is replicated
 *  for each node of the stochastic model.
 *
 *  The user can access the perturbation arrays from OSL and alter them
 *  node-by-node if perturbations differ by scenario.
 *  See the OSL documentation on EKKPMDL for details on implementation
 *  of perturbations.
 *
 *  INPUT:
 *
 *  Stoch_Model *smodel  - pointer to stochastic model initialized by scnlmdl.
 *  double *cdelta       - array of objective perturbations.
 *  double *lrdelta      - array of row lower bound perturbations.
 *  double *urdelta      - array of row upper bound perturbations.
 *  double *lcdelta      - array of column lower bound perturbations.
 *  double *ucdelta      - array of column upper bound perturbations.
 *  long mask            - says which perturbations are used:
 *                         mask = C*1 + LR*2 + UR*4 + LC*8 + UC*16
 *                         where C,LR,UR,LC,UC are 1 or 0 depending on
 *                         whether there is or is not a perturbation of
 *                         the corresponding objective or bound.
 *
```

```
 * OUTPUT:
 *  long *rc            - return code (2 is fatal)
 *
 * RETURNS:
 *  void
 */
```

SCNPTSD. Put stochastic data/model into SP file.

```
extern void scnptsd( long *rc, Stoch *stoch, char *filename );
/* scnptsd.c
 *
 *  Puts the Stoch_Data or Stoch_Model structure into an SP file to be
 *  called "filename".  If a Stoch_Model structure, then only that data
 *  relevant to the Stoch_Model is stored.
 *
 *  Opens the file as a buffered direct access file, writes the data and
 *  then closes it.
 *
 *  File may be read by a call to scngtsd().
 *
 * INPUT:
 *   Stoch_Data *stoch   - stochastic data/model pointer
 *   char *filename      - name of SP file.
 *
 * OUTPUT:
 *   long *rc           - return code (2 is fatal)
 *
 * RETURNS:
 *   void
 */
```

SCNSGET. Get scenario tree information.

```
extern void scnsget( long *rc, Stoch *stoch, usrStoch **pstree );
/* scnsget.c
 *
 *  Mallocs a usrStoch structure and compiles stochastic tree
 *  information into the usrStoch arrays.
 *
 *  The input can be either a Stoch_Data structure or a Stoch_Model
 *  structure.  If a Stoch_Model then information for only those nodes in the
 *  model is compiled.
```

```
 *
 * INPUT:
 *  Stoch *stoch       - pointer to a Stoch_Data or a Stoch_Model structure.
 * OUTPUT:
 *  usrStoch **pstree  - pointer to a usrStoch structure.
 *
 * RETURNS:
 *  long *rc           - return code (2 is fatal)
 */
```

SCNSMPS. Read SMPS stoch file.

```
extern void scnsmps( long *rc, Stoch_Data *stoch, long smpsunit );
/* scnsmps.c
 *
 * Accepts a Stoch_Data structure processed by scntmps(),
 * and a fortran unit number of a STOCH file.
 * Processes STOCH file, creates scenario data and stores it in the
 * Stoch_Data structure.
 *
 * INPUT:
 *
 *   Stoch_Data *stoch  - pointer to Stoch_Data structure processed by scntmps.
 *   long smpsunit      - integer identifying FORTRAN unit no of STOCH file.
 *
 * OUTPUT:
 *   rc                 - return code (2 is fatal).
 *
 * RETURNS:
 *   void
 */
```

SCNSOLN. Return scenario solution sorted into user's original order.

```
extern void scnsoln( long *rc, Stoch_Model *smodel, long s, long ncol,
                     double *solout, long *inv );
/* scnsoln.c
 *     Put scenario solution at location pointed to by solout[].
 *      (Must be malloced by user.)
 *
 * INPUT:
 *  Stoch_Model *smodel  - pointer to Stoch_Model initialized by scndscm.
 *  long s               - scenario number for which solution is desired
```

```
 *  long ncol           - number of columns in core model
 *  double *solout      - pointer to ncol doublewords
 *  long *inv           - pointer to ncol long integers
 *
 * OUTPUT:
 *  long *rc          - return code (2 is fatal)
 *  double *solout    - array containing solution entries ordered according to
 *                      the original ordering (before sorting).
 *  long *inv         - array containing internal (sorted) indices
 *                      which entry in the original ordering maps to which
 *                      entry in the ordering after sorting.
 *
 * RETURNS:
 *     void
 */
```

SCNSOLV. Solve stochastic model.

```
extern void scnsolv( long *rc, Stoch_Model *smodel, double *dspace,
                     long nmod, double maxmin, long defaultsubs );
/* scnsolv.c
 *
 *     Solve stochastic model using Benders decomposition subroutine
 *     scnbndr(), then simplex (EKKSSLV) after Benders terminates.
 *
 *     Various defaults can be overridden through the file scnsolv.defaults.
 *     See scnbndr() for documentation on parameters.
 *
 * scnsolv.defaults file.....
 *     Maximum_Subproblems         10
 *     Cut_Stage                   3
 *     Benders_Major_Iterations    60
 *     Optimality_Gap              1.0e-6
 *     EKKNWMT_Type                1
 *     Perform_EKKSCAL             0
 *     EKKCRSH_Method              3
 *     Perform_EKKPRSL             0
 *     EKKPRSL_Type                0
 *     EKKPRSL_Unit                0
 *     EKKSSLV_Algorithm           1
 *
 *...supplies parameters to the scnbndr() call:
 *
 *     Maximum_Subproblems      - scnbndr:maxmod
 *     Cut_Stage                - scnbndr:ipcut
```

```
 *     Benders_Major_Iterations - scnbndr:maxits
 *     Optimality_Gap           - scnbndr:optgap
 *     EKKNWMT_Type             - scnbndr:inwmt
 *     Perform_EKKSCAL          - scnbndr:iscal
 *     EKKCRSH_Method           - scnbndr:icrsh
 *     Perform_EKKPRSL          - scnbndr:iprsl
 *     EKKPRSL_Type             - scnbndr:iprsltype
 *     EKKPRSL_Unit             - scnbndr:iprslunit
 *     EKKSSLV_Algorithm        - scnbndr:isalg
 *
 * INPUT:
 *
 * Stoch_Model *smodel - pointer to Stoch_Model structure initialized by scndscm
 * double *dspace      - array initialized by OSL subroutine ekkdsca.
 * long nmod           - first available OSL model number for use by scnsolv
 * double maxmin       - (1.0, -1.0) if problem is (minimize, maximize)
 * long defaultsubs    - default maximum number of subproblems
 *                       (overridden by the file scnsolv.defaults, if available)
 *
 * OUTPUT:
 * long *rc            - return code (2 is fatal)
 *
 * RETURNS:
 * void.
 *
 */
```

SCNSPAR. Solve parametric stochastic model.

```
extern void scnspar( long *rc, Stoch_Model *smodel,
                     double lambda, double lambdadelta, double lambdalim,
                     long mask,
                     long (*report) (double *) ) ;
/* scnspar.c
 *
 * Solve parametric model. If lambda is non-zero, then lambda times
 * the perturbation vector is added to the original and the resulting
 * problem is solved with EKKSSLV.
 * Thereafter, the perturbation is added in increments multiplied by
 * lambdadelta up to the maximum lambdalim.
 *
 * It is more efficient to set up the problem so that the original
 * is solved with scnsolv(), then call scnspar() with lambda=0.
 *
```

```
 * Each time the incremented problem is solved, the function pointed to
 * by report() is called with argument the OSL dspace attached to smodel.
 *
 * The mask argument is used in the same fashion as in scnpmdl().
 *
 * INPUT:
 *   Stoch_Model *smodel       - model pointer initialized by scndscm()
 *   double lambda             - amount of perturbation added at first
 *   double lambdadelta        - incremental amount of perturbation added
 *   double lambdalim          - maximum perturbation added
 *   long mask                 - see scnpmdl:mask
 *   long (*report(double *))  - called after every solution with argument
 *                               OSL dspace
 *
 * OUTPUT:
 *   long *rc                  - return code (2 is fatal)
 *
 * RETURNS:
 *   void
 */
```

SCNTMPS. Process SMPS time file.

```
extern void scntmps( long *rc, Stoch_Data *stoch, double *dspace,
                     long tmpsunit );
/* scntmps.c
 *
 * Makes the core model from a TIME file and an OSL model generated from
 * an MPS file.  The TIME file identifies which rows and columns belong
 * to which time stage.  The rows and columns must be in stage order.
 *
 * The core model is generated by scncore().
 *
 * INPUT:
 * double *dspace     - doubleword array initialized by OSL
 * Stoch_Data *stoch  - pointer to Stoch_Data structure initialized by scninit
 * long tmpsunit      - integer identifying FORTRAN unit no of time file.
 *
 * OUTPUT:
 *
 * long *rc           - return code (2 is fatal).
 *
 * RETURNS:
 * void
```

```
*/
```

SCNVRDL. Delete SP file.

```
extern void scnvrdl( long *rc, Stoch_Data *stoch );
/* scnvrdl.c
 *
 * Closes and deletes temporary SP file associated to the Stoch_Data
 * structure.  (The SP file was created with the tmpnam() utility.)
 *
 * This might be done, for example, after the stochastic model has been
 * declared in OSL, or after a call to scnptsd() (put stochastic data).
 *
 *
 * INPUT:
 *  Stoch_Data *stoch - stochastic data structure initialized by scninit()
 *
 * OUTPUT:
 *  long *rc          - return code (2 is fatal)
 *
 * RETURNS:
 *   void
 */
```

APPENDIX C Description of Tree Subroutines

usrtree. The suite of subroutines declared in the header file `usrtree.h` can be used to generate an empirical tree out of simulations.

A sample event in a given time stage *must be represented by an integer*. The subroutines process a sample path in time order, beginning at the first stage (arbitrarily called time-stage 1). Each event is compared against the events already stored.

If the event does not match any already stored, a new scenario is created that branches from the last-matched node in the current time stage.

If the event matches a stored event, the time stage is increased and the processing is repeated. If all events find a match, then the weight of the matched scenario is increased.

There are also facilities for obtaining scenario information for a given sample number, and for obtaining distribution information for a given node.

TREEINIT. Initialize a tree structure.

```
extern long tree_init (long *rc, Tree **tree, long *arr, long narr, long mxsmp);
/*
 * tree_init:
 *
 * Creates tree and initializes structures.
 * Creates first scenario from event data values in arr[].
 *
 * Input:
 * arr[]  - integer array of event data values
 * narr - number of events (stages) in arr[]
 * mxsmp - maximum number of samples to be stored
 *
 * Output:
 * *tree - pointer to tree structure
 * rc - return code 0
 *
 * Returns:
 * return value = 1
 ********************************************************************/
```

TREEADD. Add a simulation to tree.

```
extern long tree_add (long *rc, Tree *tree, long *arr, long narr);
/*
 * tree_add:
 *
 * Updates scenario tree with event data values in arr[].
 * If arr[] is already in tree, then it increments counters
 * and returns the original scenario number.
 * Otherwise it creates new scenario from event data values in arr[]
 * and returns the newly created scenario number.
 *
 * Input:
 * tree - pointer to tree initialize by tree_init.
 * arr[]  - integer array of event data values
 * narr - number of event data values (stages) in arr[]
 *
 * Output:
 * rc - return codes:
 *   1 if maximum number of samples exceeded
 *   2 if tree not initialized or corrupted
 *
 * Returns:
 * scenario number as described above,
 ***********************************************************************/
```

TREEGET. Get a sample/scenario from the tree.

```
extern void tree_get(long *rc, Tree *tree, long nsmp, long *ianc,
                     long *istg, long *wgt, long *nscn, long **arr);
/*
 * tree_get:
 *
 * Finds scenario information for sample number nsmp.
 *
 * NB: if branching stage, "istg", has value 1, this means
 *     that the scenario branches from the root (ianc=0).
 *
 * Input:
 * tree - pointer to tree initialized by tree_init.
 * nsmp - sample number (first is 0, etc)
 *
 * Output:
 * *ianc  - ancestor scenario number.
```

```
 * *istg - stage number at which scenario branches
 * *wgt - scenario weight
 * *nscn - scenario number
 * **arr   - pointer to data array for scenario
 * *rc - return codes:
 *   1 if maximum number of samples exceeded
 *   2 if tree not initialized or corrupted
 *
 ***********************************************************************/
```

TREEDISTN. Generate distribution at given node.

```
extern long tree_distn(long *rc, Tree *tree, long *arr, long narr);
/*
 * tree_distn:
 *
 * Finds the node corresponding to outcomes in arr[0..nstg-1]
 * and reports the conditional distribution to standard output
 *
 * Input:
 * tree - pointer to tree initialized by tree_init.
 * arr[]  - integer array of data values
 * narr - number of data values (stages) in arr[]
 *
 * Output:
 * rc - return code:
 * 2 if tree not initialized, or corrupted
 *
 * Returns:
 * weight of node if found, or 0 if not found.
 ***********************************************************************/
```

APPENDIX D Messages

An invocation of one of the SP/OSL drivers, for example smps, produces output like this:

```
1EKK0006I Optimization Subroutine Library Version 1.2 (32111) Page 1
EKK0050I                                 Entering OSL subroutine EKKDSCA
EKK0081I This application has 1 model(s)
SCN0500I: Stochastic Programming/OSL Interface Library, Version 1.0
               (c) Copyright IBM Corp. 1993
               All rights reserved
               Licensed materials - property of IBM
SCN0502I:                                Entering SP/OSL subroutine scninit.
EKK0050I                                 Entering OSL subroutine EKKMPS
EKK0053I EKKDSCM is being called with default settings for model number 1
EKK0078I          1 NAME          APP
EKK0078I          2 ROWS
EKK0078I         29 COLUMNS
EKK0078I        106 RHS
EKK0078I        132 ENDATA
EKK0014I         Using Matrix APP
EKK0014I         Using Objective COST
EKK0014I         Using RHS RHS
EKK0016I         Matrix has 25 rows, 60 columns and 96 entries
EKK0019I         There are 40 objective row entries
SCN0502I:                                Entering SP/OSL subroutine scntmps.
SCN0504I: Core model has 25 rows, 60 columns, and 3 stages.
SCN0502I:                                Entering SP/OSL subroutine scnsmps.
SCN0701I: Scenario 1  Ancestor 0  Stage 2 Probability 0.111
SCN0701I: Scenario 2  Ancestor 1  Stage 3 Probability 0.111
SCN0701I: Scenario 3  Ancestor 1  Stage 3 Probability 0.111
SCN0701I: Scenario 4  Ancestor 0  Stage 2 Probability 0.111
SCN0701I: Scenario 5  Ancestor 4  Stage 3 Probability 0.111
SCN0701I: Scenario 6  Ancestor 4  Stage 3 Probability 0.111
SCN0701I: Scenario 7  Ancestor 0  Stage 2 Probability 0.111
SCN0701I: Scenario 8  Ancestor 7  Stage 3 Probability 0.111
SCN0701I: Scenario 9  Ancestor 7  Stage 3 Probability 0.111
SCN0502I:                                Entering SP/OSL subroutine scnptsd.
SCN0804I: Copying Stoch_Data structure to ../data/app0110.spl ...
```

The output is a mixture of OSL and SP/OSL messages. Messages beginning with EKK are OSL messages and those beginning with SCN are SP/OSL messages. For instance, in this example, the EKK messages are generated by the calls to the OSL subroutines ekkdsca, and ekkmps. The explanation of OSL messages is given in the OSL documentation. The `SCN` messages were generated by the calls to the SP/OSL subroutines `scninit`, `scntmps`, `scnsmps` and `scnptsd`.

The SP/OSL message begins with a string in the form `SCN0xxxy`. The string `xxx` is the message number. The letter `y` indicates whether the message is informational (`I`), a warning (`W`), or a serious error (`E`). The rest of the message contains information concerning the event that prompted it. In the descriptions below, the message is given in the format style of the C programming language. The letters preceded by the percent sign `%` indicate that a string (`%s`), or integer (`%d`), or real number (`%f` or `%g`) meaningful to the event is to be substituted.

SCN0500I. *Informational.*

Text: `Stochastic Programming/OSL Interface Library, Version 1.0`

Effect: None.

Meaning: Initialization and copyright announcement.

SCN0501E. *Fatal Error.*

Text: `Malloc failed in subroutine %s. Check maximum memory allocation for user.`

Effect: Subroutine exits immediately with error code 2.

Meaning: SP/OSL memory management is primarily through the C library routine `malloc`. This routine has returned a `NULL` pointer, indicating a problem with available memory. Check the load command syntax to see if available memory can be increased. Also, check the integrity of the dimensions (size of core problem, number of time steps, number of scenarios, number of nodes) to see if the memory requirements are reasonable.

SCN0502I. *Informational.*

Text: `Entering SP/OSL subroutine %s.`

Effect: None.

Meaning: Subroutine initiation announcement.

SCN0504I. *Informational.*

Text: `Core model has %d rows, %d columns, and %d stages.`

Effect: None.

Meaning: Dimensions of core model.

SCN0507E. *Fatal Error.*

Text: `Zero allocation attempted in subroutine %s. Check subroutine arguments.`

Effect: Subroutine exits immediately with error code 2.

Meaning: The subroutine has attempted to allocate a zero length memory region. This could indicate a problem with incoming dimensions. Check integrity of subroutine arguments.

SCN0607E. *Fatal Error.*

Text: `Error detected while trying to open SPL file %s in subroutine %s.`

Effect: Subroutine exits immediately with error code 2.

Meaning: A NULL pointer was returned from the C library `fopen` function on the named file.

SCN0608E. *Fatal Error.*

Text: `Error detected while trying to close SPL file %s in subroutine %s.`

Effect: Subroutine exits immediately with error code 2.

Meaning: A nonzero code was returned from the C library `fclose` function on the named file.

SCN0609E. *Fatal Error.*

Text: `Attempt to reposition within SPL file %s failed in subroutine %s.`

Effect: Subroutine exits immediately with error code 2.

Meaning: An invalid SPL record number was encountered. This may indicate that an SP/OSL data structure was overwritten before the record was written or read.

SCN0610E. *Fatal Error.*

Text: `Error detected in subroutine %s while writing to SPL file %s.`

Effect: Subroutine exits immediately with error code 2.

Meaning: The return code from the C library function `fwrite` to the named file did not match the size of the written record.

SCN0611E. *Fatal Error.*

Text: `Error detected in subroutine %s while reading from SPL file %s.`

Effect: Subroutine exits immediately with error code 2.

Meaning: The return code from the C library function `fread` to the named file did not match the size of the read record.

SCN0614E. *Fatal Error.*

Text: `Unable to allocate file pointer for SPL file %s in subroutine %s.`

Effect: Subroutine exits immediately with error code 2.

Meaning: The C library subroutine `malloc` was unable to allocate enough memory for the SPL file data stucture. Check the maximum memory available to the user. Check also the loader syntax to determine if more memory segments can be allocated.

SCN0615E. *Fatal Error.*

Text: `Unable to allocate filename storage for SPL file %s in subroutine %s.`

Effect: Subroutine exits immediately with error code 2.

Meaning: The C library subroutine `malloc` was unable to allocate enough memory for the SPL file name. Check the size of the file name. Check the maximum memory available to the user. Check also the loader syntax to determine if more memory segments can be allocated.

SCN0616E. *Fatal Error.*

Text: `Invalid SPL file pointer specified in subroutine %s.`

Effect: Subroutine exits immediately with error code 2.

Meaning: The SPL file data structure has been corrupted, perhaps overwritten.

SCN0619E. *Fatal Error.*

Text: `Invalid virtual record specified in subroutine %s for SPL file %s.`

Effect: Subroutine exits immediately with error code 2.

Meaning: The SPL record pointer is out of bounds. Probably the `Stoch_Data` structure has been corrupted.

SCN0620E. *Fatal Error.*

Text: `Negative virtual record size specified in subroutine %s for SPL file %s.`

Effect: Subroutine exits immediately with error code 2.

Meaning: The subroutine has attempted to write a record of negative size to the SPL file. This may indicate improper arguments or dimensions being supplied to the subroutine.

SCN0621E. *Fatal Error.*

Text: `Empty string specified for SPL filename in subroutine %s.`

Effect: Subroutine exits immediately with error code 2.

Meaning: Check the filename argument to the subroutine.

SCN0622E. *Fatal Error.*

Text: `Existing SPL file %s opened in subroutine %s is improperly initialized.`

Effect: Subroutine exits immediately with error code 2.

Meaning: The file is either not an SPL file, or the file has been corrupted.

SCN0623E. *Fatal Error.*

Text: `Error detected in subroutine %s while deleting SPL file %s.`

Effect: Subroutine exits immediately with error code 2.

Meaning: The C library routine `remove` failed to delete the named file. Check the file's permissions.

SCN0626E. *Fatal Error.*

Text: `Unable to initialize %s for SPL file %s in subroutine %s.`

Effect: Subroutine exits immediately with error code 2.

Meaning: The named record in the SPL file has failed an integrity check. This may mean that the file has been corrupted.

SCN0627E. *Fatal Error.*

Text: `Error detected while attempting to save SPL file with name %s.`

Effect: Subroutine exits immediately with error code 2.

Meaning: The system `cp` command failed to copy the temporary SPL file. Check your filesystem (these files can be large!).

SCN0630E. *Fatal Error.*

Text: `Data for current model-block is not block lower-triangular.`

Effect: Subroutine exits immediately with error code 2.

Meaning: The subroutine has received a matrix block that cannot be sorted into block lower-triangular form with the row/column stage partition information provided. Check the arguments to `scncore` and or `scnadsc` to be sure that the data is consistent.

SCN0631W. *Warning.*

Text: `Data for current model-block contains elements for earlier stages.`

Effect: Elements from earlier stages are ignored. Program continues.

Meaning: Subroutine `scnadsc` has encountered data with entries that belong to time stages earlier than that indicated by the branching stage information. Check the arguments to `scnadsc` verify the integrity of the data.

SCN0701I. *Informational.*

Text: `Scenario %i Ancestor %i Stage %i Probability %g`

Effect: None.

Meaning: Scenario detail from `scnadsc` calling sequence.

SCN0702W. *Warning.*

Text: `Scenario %i has already been defined.  Probability is changed to %g.`

Effect: Probability of scenario is incremented by the incoming probability. Program continues.

Meaning: Subroutine `scnadsc` was called twice for the same scenario index. Check the subroutine call to be sure that this is intended.

SCN0703E. *Fatal Error.*

Text: `Number of %s requested, %i, exceeds declared maximum %i.`

Effect: Subroutine exits immediately with error code 2.

Meaning: Check integrity of number requested, reset maximums if necessary.

SCN0704E. *Fatal Error.*

Text: `Branching stage out of bounds.`

Effect: Subroutine exits immediately with error code 2.

Meaning: Subroutine `scnadsc` has been called with a branching stage that is greater than maximum or less than zero. Check integrity of arguments.

SCN0705E. *Fatal Error.*

Text: `Scenario has negative probability.`

Effect: Subroutine exits immediately with error code 2.

Meaning: Subroutine `scnadsc` has encountered a negative probability. Check integrity of arguments.

SCN0706W. *Warning.*

Text: `Scenarios that branch from the root have branching stage = 2`

Effect: Branching stage is changed to 2. Program continues.

Meaning: A branching stage less than 2 was encountered in the arguments to `scnadsc`. This can indicate a serious error — namely, that the branching stage indices are consistently incorrect. Check the arguments to `scnadsc`.

SCN0707E. *Fatal Error.*

Text: `Ancestor scenario %i does not exist.`

Effect: Subroutine exits immediately with error code 2.

Meaning: Subroutine `scnadsc` has been called with an ancestor argument that has not previously been entered into the model. Check the integrity of the argument. Possibly a change in sequencing of the scenarios list will be required.

SCN0708W. *Warning.*

Text: `Scenario %i branches prior to ancestor's branch stage: true ancestor is %i`

Effect: Scenario ancestor is reset. Program continues.

Meaning: Inconsistent branching information was encountered in the arguments to `scnadsc`. Check the integrity of the arguments to `scnadsc`.

SCN0709E. *Fatal Error.*

Text: `Number of %s in CORE model %i does not match number %i in scenario`

Effect: Subroutine exits immediately with error code 2.

Meaning: Subroutine `scnadsc` has encountered a scenario whose basic dimensions do not match that of the core model. Check the integrity of the arguments to `scnadsc` and `scncore`.

SCN0710W. *Warning.*

Text: `Core %s value at index %d is infinite. Nonzero value in scenario %d set to zero.`

Effect: Scenario value of bound is set to zero. Program continues.

Meaning: A core bound is infinite, but the incoming scenario bound had a nonzero value. Since the sum of an infinite and a finite value is infinite, the program just sets the scenario value to zero. Check the arguments to `scnadsc` to determine whether this was intended.

SCN0711E. *Fatal Error.*

Text: `Total probability in model is zero.`

Effect: Subroutine exits immediately with error code 2.

Meaning: The probabilities of the scenarios in the model are zero. So there is no point in going on.

SCN0712I. *Informational.*

Text: `Declaring stochastic model in OSL. Number of rows %d, number of columns %d.`

Effect: None.

Meaning: Dimensions of stochastic model declared in OSL by `scnlmdl` or `scnemdl`.

SCN0713I. *Informational.*

Text: `Total number of blocks declared to OSL is %d.`

Effect: None.

Meaning: Number of blocks passed over to OSL in subroutine `scnlmdl` or `scndsmt`.

SCN0714I. *Informational.*

Text: `Total number of %s is %d.`

Effect: None.

Meaning: Various totals reported in `scndscm`.

SCN0715E. *Fatal Error.*

Text: `Node list is empty. Cannot construct model.`

Effect: Subroutine exits immediately with error code 2.

Meaning: Subroutine `scndscm` has been provided with a nodelist that has no elements, but argument `nnode` is non-zero. Check the integrity of the arguments to `scndscm`.

SCN0716W. *Warning.*

Text: `Node list is non-empty. Ignored.`

Effect: Nodelist is ignored. Program continues.

Meaning: Subroutine `scndscm` was called with `nnodes = 0` (indicating that the full model is desired) but the incoming nodelist was nonnull. Check the arguments to `scndscm` to be sure that this was intended.

SCN0717I. *Informational.*

Text: `Constructing stochastic model from original node list.`

Effect: None.

Meaning: Subroutine `scndscm` reports this.

SCN0718W. *Warning.*

Text: `%d zero probability nodes have been removed from model node list.`

Effect: The zero probability nodes are ignored in subsequent model construction or solution. Program continues.

Meaning: Subroutine `scnlmdl` or `scnemdl` has detected zero probability nodes in an incoming nodelist. This usually means that some scenario probabilities have been set to zero, for example in subroutine `scncprb`.

SCN0719I. *Informational.*

Text: `Process time for %s is %g seconds.`

Effect: None.

Meaning: As read.

SCN0720I. *Informational.*

Text: `Beginning %s.`

Effect: None.

Meaning: The named phase of a procedure (for example, Benders decomposition) has begun. This usually appears at the beginning of a long sequence of OSL messages, which should be interpreted as pertaining to the procedure named.

SCN0755E. *Fatal Error.*

Text: `Error creating hash tables for %s names.`

Effect: Subroutine exits immediately with error code 2.

Meaning: While processing an SMPS file, a string has been encountered that cannot be hashed. Check the integrity of the data in the SMPS file.

SCN0756E. *Fatal Error.*

Text: `Not enough space in dspace for %s file processing.`

Effect: Subroutine exits immediately with error code 2.

Meaning: Processing of SMPS files uses temporary regions in OSL's dspace. The subroutine requires more space. Check the validity of the request, and increase it if necessary.

SCN0757E. *Fatal Error.*

Text: `Index error in %s file.`

Effect: Subroutine exits immediately with error code 2.

Meaning: An improper name has been found for an index in an SMPS file. Check the integrity of the data in the SMPS file.

SCN0758W. *Warning.*

Text: `No ENDATA card in %s file. Ignored.`

Effect: The end of an SMPS file has been reached, but no 'ENDATA' line was read. Program assumes one was intended and continues.

Meaning: Check the SMPS file to ensure that it has not been corrupted.

SCN0759E. *Fatal Error.*

Text: `Premature EOF reached in %s file.`

Effect: Subroutine exits immediately with error code 2.

Meaning: The end of an SMPS file has been reached, but not enough information has been received to complete the stochastic model. Check the SMPS file to ensure its integrity.

SCN0760E. *Fatal Error.*

Text: `Expecting %s card in %s file.`

Effect: Subroutine exits immediately with error code 2.

Meaning: The subroutine processing the SMPS file is expecting to find a certain string. Check the SMPS file to ensure its integrity.

SCN0762E. *Fatal Error.*

Text: `Too many elements in scenario %d. Consider revising CORE file, so that more of these elements appear there.`

Effect: Subroutine exits immediately with error code 2.

Meaning: A scenario defined in a STOCH file is asking for more matrix elements than was found in the CORE matrix. (The maximum space reserved for reading in the scenario matrix data from a CORE file is precisely the size of the CORE matrix.) In this case, consider either: revising the core file, as suggested, or using `scnadsc` directly to define the scenario in question.

SCN0763E. *Fatal Error.*

Text: `Cannot apply a Range to an equality row in MPS format.`

Effect: Subroutine exits immediately with error code 2.

Meaning: In the STOCH file you tried to apply a range to a row that was declared as an equality row in the CORE file. Sorry, no can do.

SCN0764E. *Fatal Error.*

Text: `Error reading %s file.`

Effect: Subroutine exits immediately with error code 2.

Meaning: A FORTRAN read error has been encountered. Check integrity of SMPS files.

SCN0801E. *Fatal Error.*

Text: `OSL not initialized or corrupted. This was detected in subroutine %s.`

Effect: Subroutine exits immediately with error code 2.

Meaning: The subroutine checks the incoming `osl_dspace` to see if OSL has initialized it. Check to see if `ekkdsca` or `ekkinit` has been called, or if the `osl_dspace` pointer has been overwritten.

SCN0802E. *Fatal Error.*

Text: `%s structures not initialized or corrupted. This was detected in subroutine %s.`

Effect: Subroutine exits immediately with error code 2.

Meaning: The subroutine checks the incoming pointer to see if it is a valid SP/OSL structure. Check to see if `scninit` or `scndscm` has been called, or if the pointer has been overwritten.

SCN0803I. *Informational.*

Text: Varies.

Effect: None.

Meaning: As read.

SCN0804I. *Informational.*

Text: Varies.

Effect: None.

Meaning: As read.

SCN0805I. *Informational.*

Text: Varies.

Effect: None.

Meaning: As read.

APPENDIX E   Driver Source Code

This section contains the source code for the SP/OSL drivers described in section 4, above.

SMPS. Read S-MPS formatted files and save SP file.

```
/*  @(#)smps.c      4.1 9/22/94
 *
 * smps.c:  Main driver for reading SMPS files.
 *
 *          Usage:   smps <file>
 *
 *          Reads CORE, TIME, and STOCH files from
 *          <file>.core, <file>.time, and <file>.stoch.
 *          Writes the SP-file as <file>.spl.
 */
#include <ekkc.h>
#include <usrstoc.h>
#include <string.h>
#define MXSPC 2000000
#define MXSCN 200
main( long argc, char *argv[] ) {
    Stoch_Data *s_data=NULL;                     /* Stoch_Data pointer */
    double *osl_dspace=NULL;                     /* space for OSL */
    long ucore=20,utime=21,ustoch=22;            /* fortran unit numbers */
    long rc=0;                                   /* local variables */
/* filenames from argument list */
    char *kspc,*corefile,*timefile,*stocfile,*splfile;
    long nsize,nchar;
    long len;
/* get application name from command line */
    if ( argc != 2 ) {
      printf(" Usage:  smps <file>  , where "
             "{file.core,file.time,file.stoch} is a valid "
             "SMPS triple ");
      return;
    }
    nchar = (long) strlen( argv[1] );
/*
 *  make file names:
 *      "app".core
 *      "app".time
 *      "app".stoch
 */
    nsize = 4*(nchar+1)+(5+5+6+3+5);
```

```
    kspc = (char *) malloc(sizeof(char)*nsize);
    if ( kspc==NULL ) {
      printf(" ERROR: smps is unable to store input file-names ");
      return;
    }
    corefile = kspc; kspc += (nchar+1)+5;
     strcpy( corefile,argv[1] ); strcat( corefile,".core" );
    timefile = kspc; kspc += (nchar+1)+5;
     strcpy( timefile,argv[1] ); strcat( timefile,".time" );
    stocfile = kspc; kspc += (nchar+1)+6;
     strcpy( stocfile,argv[1] ); strcat( stocfile,".stoch" );
    splfile  = kspc; kspc += (nchar+1)+3;
     strcpy( splfile,argv[1] ); strcat( splfile,".spl" );
/* open files on fortran unit numbers */
    len = strlen(corefile); openfseq(&rc,&ucore,corefile,&len);
    len = strlen(timefile); openfseq(&rc,&utime,timefile,&len);
    len = strlen(stocfile); openfseq(&rc,&ustoch,stocfile,&len);
/* allocate space for OSL */
    osl_dspace = (double *) malloc (MXSPC*sizeof(double));
    if (osl_dspace==NULL) {
      printf(" System failed to allocate memory for osl_dspace ");
      return;
    }
/* initialize OSL with one model */
    ekkdsca( &rc,osl_dspace,MXSPC,1 );        if ( rc>299 ) return;
/* initialize Stoch_Data with maximum MXSCN scenarios */
    scninit( &rc,&s_data,MXSCN );             if ( rc>1 ) return;
/* Read Core MPS file into OSL */
    ekkmps( &rc,osl_dspace,ucore,1,0 );       if ( rc>299 ) return;
/* Process TIME file */
    scntmps( &rc,s_data,osl_dspace,utime);    if ( rc>1 ) return;
/* Process STOCH file */
    scnsmps( &rc,s_data,ustoch);              if ( rc>1 ) return;
/* Save SP-file */
    scnptsd( &rc,s_data,splfile );
/* Free Stoch_Data structures */
    scnfree(s_data);
}
```

SGBD. Generate aircraft allocation problem using `scnadsc`.

```
/*  @(#)sgbd.c      3.3 9/22/94
 *
 * sgbd.c:  Generic driver using scnadsc.
 *
```

```
 *          Implements G.B. Dantzig's Aircraft Allocation model,
 *          in: G.B. Dantzig, Linear Programming and Extensions
 *          (Princeton U. Press, 1963) pp. 572-597.
 *
 *          See also: A.J. King, "Stochastic programming problems:
 *          examples from the literature",
 *           in: Yu. Ermoliev and R.J-B Wets, eds, Numerical Techniques for
 *          Stochastic Optimization (Springer-Verlag, 1988) pp. 543-567.
 */
#include <usrstoc.h>
#define INF 1.0e31
main( long argc, char *argv[] ) {
    Stoch_Data *s_data;                         /* Stoch_Data pointer */
/* Model dimensions */
    long ncol=27, nrow=9, nels=44;
/* Sparse matrix data...organized by row */
    long mrow[]={ 1, 1, 1, 1, 1,
                  2, 2, 2, 2,
                  3, 3, 3,
                  4, 4, 4, 4, 4,
                  5, 5, 5, 5,
                  6, 6, 6, 6, 6, 6,
                  7, 7, 7, 7, 7,
                  8, 8, 8, 8, 8, 8,
                  9, 9, 9, 9, 9, 9 };
    long mcol[]={ 1, 2, 3, 4, 5,
                  6, 7, 8, 9,
                  10, 11, 12,
                  13, 14, 15, 16, 17,
                  1,        13, 18, 19,
                  2, 6, 10, 14, 20, 21,
                  3, 7,     15, 22, 23,
                  4, 8, 11, 16, 24, 25,
                  5, 9, 12, 17, 26, 27 };
    double dels[] = { 1.0, 1.0, 1.0, 1.0, 1.0,
                      1.0, 1.0, 1.0, 1.0,
                      1.0, 1.0, 1.0,
                      1.0, 1.0, 1.0, 1.0, 1.0,
                      16.0,              9.0, -1.0, 1.0,
                      15.0, 10.0,  5.0, 11.0, -1.0, 1.0,
                      28.0, 14.0,       22.0, -1.0, 1.0,
                      23.0, 15.0,  7.0, 17.0, -1.0, 1.0,
                      81.0, 57.0, 29.0, 55.0, -1.0, 1.0 };
/* Objective */
    double dobj[]={ 18.0, 21.0, 18.0, 16.0, 10.0, 15.0, 16.0, 14.0, 9.0,
```

```
                  10.0,  9.0,  6.0, 17.0, 16.0, 17.0, 15.0, 10.0, 0.0,
                  13.0,  0.0, 13.0,  0.0,  7.0,  0.0,  7.0,  0.0, 1.0 };
/* Column bounds */
    double dclo[]={ 0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,
                    0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,
                    0.0,  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,  INF,
                    INF,  INF,  INF,  INF,  INF,  INF,  INF,  INF,  INF,
                    INF,  INF,  INF,  INF,  INF,  INF,  INF,  INF,  INF };
/* Row bounds */
    double drlo[]={ -INF, -INF, -INF, -INF,  0.0, 0.0, 0.0, 0.0, 0.0 };
    double drup[]={ 10.0, 19.0, 25.0, 15.0,  0.0, 0.0, 0.0, 0.0, 0.0 };
/* Stages */
    long nstg=2;
    long n_first_stg_rows=4;
    long n_first_stg_cols=17;
    long rstg[]={ 1,1,1,1,2,2,2,2,2 };
    long cstg[]={ 1,1,1,1,1,1,1,1,1,
                  1,1,1,1,1,1,1,1,2,
                  2,2,2,2,2,2,2,2,2 };
/* Stochastic data */
    long nindp=5;
    long nsamp[]={ 5, 2, 5, 5, 3 };
    double demand[]={ 200, 220, 250, 270, 300,
                      50, 150,
                      140, 160, 180, 200, 220,
                      10, 50, 80, 100, 340,
                      580, 600, 620 };
    double dprobs[]={ 0.2, 0.05, 0.35, 0.2, 0.2,
                      0.3, 0.7,
                      0.1, 0.2, 0.4, 0.2, 0.1,
                      0.2, 0.2, 0.3, 0.2, 0.1,
                      0.1, 0.8, 0.1 };
/* local variables */
    long rc,ns=1,ii,iii,jj,*indx,*incr;
    double dd,dp=1.0;
/* compute number of scenarios */
    for (ii=0;ii<nindp;ii++) ns *= nsamp[ii];
/* initialize Stoch_Data with ns scenarios */
    scninit( &rc,&s_data,ns );                   if ( rc>1 ) return;
/* Create Core */
    scncore( &rc,s_data,nstg,rstg,cstg,nrow,ncol,nels,
           dobj,drlo,drup,dclo,dcup,mrow,mcol,dels );
/* Generate Scenarios */
/* ...zero everything */
```

```
    memset( dobj,0,ncol*sizeof(double) );
    memset( drlo,0,nrow*sizeof(double) );
    memset( drup,0,nrow*sizeof(double) );
    memset( dclo,0,ncol*sizeof(double) );
    memset( dcup,0,ncol*sizeof(double) );
    nels = 0;
/* ...initialization -- indx points to first sample of each rv */
    indx = (long *) malloc( (1+nindp)*sizeof(long) );
    indx[0] = 0;
    for (jj=0;jj<nindp;jj++) {
      indx[jj+1] += indx[jj] + nsamp[jj];
      dp *= dprobs[ indx[jj] ];
      drlo[ n_first_stg_rows + jj ] = demand[ indx[jj] ];
      drup[ n_first_stg_rows + jj ] = demand[ indx[jj] ];
    }
/* ...add first sample */
    scnadsc( &rc,s_data,1,0,2,dp,1,
             nrow,ncol,nels,dobj,drlo,drup,dclo,dcup,
             mrow,mcol,dels );
/* ...initialize the sample space increment for each rv to 1 */
    incr = (long *) malloc( nindp*sizeof(long) );
    for (jj=0;jj<nindp;jj++) incr[jj] = 1;
/* ...main loop
      The logic of this loop is as follows:
        While the sample size nsamp[jj] divides the scenario index ii,
          reverse the increment direction incr[jj],
          and increase the random variable index jj by 1.
        Increment the jj'th random variable by incr[jj]
          and call scnadsc with the new sample.
*/
    for (ii=1;ii<ns;ii++) {
      iii=ii; jj=0;
      while ( !(iii%nsamp[jj]) ) {
          iii /= nsamp[jj];
          incr[jj] = -incr[jj];
          jj++;
      }
      dd = dprobs[ indx[jj] ];
      indx[jj] += incr[jj];
      dp = dp* dprobs[ indx[jj] ]/dd;
      drlo[ n_first_stg_rows + jj ] = demand[ indx[jj] ];
      drup[ n_first_stg_rows + jj ] = demand[ indx[jj] ];
      scnadsc( &rc,s_data,ii+1,0,2,dp,1,
               nrow,ncol,nels,dobj,drlo,drup,dclo,dcup,
               mrow,mcol,dels );
```

```
/* turn off most of the annoying messages */
      if ( ii == 1) { scnmset( &rc,502,0); scnmset( &rc,701,0);
               printf("...assembling scenarios .... ");}
      if ( ii == ns-2) { scnmset( &rc,502,1); scnmset( &rc,701,1);}
    }
/* Save SP-file as "sgbd.spl" */
    scnptsd( &rc,s_data,"sgbd.spl" );
/* Free Stoch_Data structures */
    scnfree(s_data);
}
```

SOSL. Read SP file and solve using OSL `ekksslv`.

```
/*  @(#)sosl.c      3.4 9/22/94
 *
 * solv.c:  Main driver for reading SP-files and solving with OSL.
 *
 *          Usage:  solv <file>
 *
 *          Reads problem from file <file>.spl and solves using
 *          ekksslv().
 */
#include <usrstoc.h>
#include <ekkc.h>
#include <string.h>
#include <oslr.h>
#define MXSCN 750                   /* maximum scenarios */
#define MXSPC 2000000               /* OSL solution space */
#define MXSUB 100                   /* maximum Benders subproblems */
#define OPT  1.0                    /* minimize: 1.0; maximize: -1.0 */
main( long argc, char *argv[] ) {
    Stoch_Data *s_data=NULL;              /* Stoch_Data pointer */
    Stoch_Model *s_model=NULL;            /* Stoch_Model pointer */
    usrCoreData *u_core=NULL;             /* usrCoreData pointer */
    double *osl_dspace= NULL;             /* space for OSL */
    double *solution=NULL;                /* space for solution */
    long *osl_indx;                       /* space for osl indices */
    long rc, nf, ii, nblks, nmod;         /* local variables */
      /* filenames */
      char *splfile;
      long nsize,nchar;
      long len;
      /* get application name from command line */
      if ( argc != 2 ) {
            printf(" Usage:  sosl <file>  , where "
```

```
                "{file.spl} is a valid SP-file. ");
            return;
      }
      nchar = (long) strlen( argv[1] );
      nsize = (nchar+1)+3;
      splfile = (char *) malloc(sizeof(char)*nsize);
      if ( splfile==NULL ) {
            printf(" ERROR: sosl is unable to store input file-name ");
            return;
      }
      strcpy( splfile,argv[1] );
      strcat( splfile,".spl" );
/* malloc space for OSL */
    osl_dspace = (double *) malloc (MXSPC*sizeof(double));
    if ( osl_dspace == NULL ) {
      printf("Cannot malloc requested space for OSL ");
      return;
      }
/* initialize OSL */
    ekkdsca( &rc,osl_dspace,MXSPC,1 );
      if ( rc>299 ) return;
/* initialize Stoch_Data with maximum MXSCN scenarios */
    scninit( &rc,&s_data,MXSCN );
      if ( rc>1 ) return;
/* Read SP-file */
    scngtsd( &rc,s_data,splfile );
      if ( rc>1 ) return;
/* Declare stochastic model (full model) */
    scndscm( &rc,s_data,0,0,&s_model,&nblks,1 );
      if ( rc>1 ) return;
/* Declare full model in OSL with room for nblocks */
    nmod = 1;
    ekkdscm( &rc,osl_dspace,nmod,nblks );
      if ( rc>299 ) return;
/* Pass data to OSL model */
    scnlmdl( &rc,s_model,osl_dspace,nmod );
      if ( rc>1 ) return;
/* Check OPT, revise RMAXMIN if necessary */
    if ( OPT < 0.0 ) {
      ekkrget( &rc,osl_dspace,oslr,OSLRLN );
      RMAXMIN = OPT;
      ekkrset( &rc,osl_dspace,oslr,OSLRLN );
    }
/* Solve using EKKCRSH and EKKSSLV */
    ekkscal( &rc,osl_dspace );
```

```
    ekkcrsh( &rc,osl_dspace,1 );
    ekksslv( &rc,osl_dspace,0,2 );
/* debug
    {
    long unit=99;
    ekkgtmi( &rc,osl_dspace,nmod );
    ekkbcdo( &rc,osl_dspace,unit,1,1 );
    closefort(&unit);
    }
*/
/* Get core data */
    scncdat( &rc,s_data,&u_core );
      if ( rc>1 ) return;
/* Get first scenario solution */
    solution = (double *) malloc( u_core->ncol*sizeof(double) );
    osl_indx = (long *) malloc( u_core->ncol*sizeof(long) );
    scnsoln( &rc,s_model,1,u_core->ncol,solution,osl_indx );
      if ( rc>1 ) return;
/* Print first stage solution */
    nf = u_core->colstart[1] - u_core->colstart[0];
    printf("   Number of first stage columns %d ",nf);
    for (ii=0;ii<nf;ii++)
      printf(" Solution for index %d is %g ",ii+1,solution[ii] );
/* Free data structures */
    scnfree( s_model ); scnfree( s_data );
}
```

SOLV. Read an SP file and solve using scnsolv.

```
/*  @(#)solv.c      3.4 7/7/94
 *
 * solv.c:  Main driver for reading SP-files and solving.
 *
 *          Reads problem from file <file>.spl and solves using
 *          scnsolv().
 */
#include <usrstoc.h>
#include <ekkc.h>
#include <string.h>
#define MXSCN 750                   /* maximum scenarios */
#define MXSPC 2000000               /* OSL solution space */
#define MXSUB 750                   /* maximum Benders subproblems */
#define OPT  1.0                    /* minimize: 1.0; maximize: -1.0 */
main( long argc, char *argv[] ) {
    Stoch_Data *s_data=NULL;              /* Stoch_Data pointer */
```

```
    Stoch_Model *s_model=NULL;           /* Stoch_Model pointer */
    usrCoreData *u_core=NULL;            /* usrCoreData pointer */
    double *osl_dspace=NULL;             /* space for OSL */
    double *solution;                    /* space for solution */
    long *osl_indx;                      /* space for osl indices */
    long rc, nf, ii, nblks, nmod;        /* local variables */
      /* filenames */
      char *splfile;
      long nchar,nsize;
      long len;
      /* get application name from command line */
      if ( argc != 2 ) {
            printf(" Usage:  solv <file>  , where "
                "{file.spl} is a valid SPL file. ");
            return;
      }
      nchar = (long) strlen( argv[1] );
      nsize = (nchar+1)+3;
      splfile = (char *) malloc(sizeof(char)*nsize);
      if ( splfile==NULL ) {
            printf(" ERROR: solv is unable to store input file-name ");
            return;
      }
      strcpy( splfile,argv[1] );
      strcat( splfile,".spl" );
/* malloc space for OSL */
    osl_dspace = (double *) malloc (MXSPC*sizeof(double));
    if ( osl_dspace == NULL ) {
      printf("Cannot malloc requested space for OSL ");
      return;
      }
/* initialize OSL with 1+MXSUB models */
    ekkdsca( &rc,osl_dspace,MXSPC,1+MXSUB );
      if ( rc>299 ) return;
/* initialize Stoch_Data with maximum MXSCN scenarios */
    scninit( &rc,&s_data,MXSCN );
      if ( rc>1 ) return;
/* Read SP-file */
    scngtsd( &rc,s_data,splfile );
      if ( rc>1 ) return;
/* Declare stochastic model (full model) */
    scndscm( &rc,s_data,0,0,&s_model,&nblks,1 );
      if ( rc>1 ) return;
/* Declare full model in OSL with room for nblocks */
    nmod = 1;
```

```
    ekkdscm( &rc,osl_dspace,nmod,nblks );
      if ( rc>299 ) return;
/* Solve -- user can set options in scnsolv.defaults */
    scnsolv( &rc,s_model,osl_dspace,nmod,OPT,MXSUB );
      if ( rc>1 ) return;
/* debug
    {
    long unit=99;
    ekkgtmi( &rc,osl_dspace,nmod );
    ekkbcdo( &rc,osl_dspace,unit,1,1 );
    closefort(&unit);
    }
*/
/* Get core data */
    scncdat( &rc,s_data,&u_core );
      if ( rc>1 ) return;
/* Get first scenario solution */
    solution = (double *) malloc( u_core->ncol*sizeof(double) );
    osl_indx = (long *) malloc( u_core->ncol*sizeof(long) );
    scnsoln( &rc,s_model,1,u_core->ncol,solution,osl_indx );
      if ( rc>1 ) return;
/* Print first stage solution */
    nf = u_core->colstart[1] - u_core->colstart[0];
    printf("   Number of first stage columns %d ",nf);
    for (ii=0;ii<nf;ii++)
      printf(" Solution for index %d is %g ",ii+1,solution[ii] );
/* Free data structures */
    scnfree( s_model ); scnfree( s_data );
}
```

SDST. Solve and compute distribution of optimal values.

```
/*  @(#)sdst.c      3.6 9/22/94
 *
 * sdst.c:  Present distribution of optimal values.
 *
 *          Usage:  sdst <file>
 *
 *          Reads problem from file <file>.spl and solves using
 *          scnsolv().  Then, as an example of user coding, computes
 *          the distribution of the optimal value.
 */
#include <math.h>
#include <usrstoc.h>
#include <ekkc.h>
```

```
#include <osln.h>
#include <string.h>
#define MXSCN 750                      /* maximum scenarios */
#define MXSPC 2000000                  /* OSL solution space */
#define MXSUB 100                      /* maximum Benders subproblems */
#define OPT  1.0                       /* minimize: 1.0; maximize: -1.0 */
main( long argc, char *argv[] ) {
    Stoch_Data *s_data=NULL;                /* Stoch_Data pointer */
    Stoch_Model *s_model=NULL;              /* Stoch_Model pointer */
    usrCoreData *u_core=NULL;               /* usrCoreData pointer */
    double *osl_dspace=NULL;                /* space for OSL */
    double *solution=NULL;                  /* space for solution */
    long *osl_indx=NULL;                    /* space for osl indices */
    long rc, nf, ii, nblks, nmod;           /* local variables */
      /* filenames */
      char *splfile;
      long nsize,nchar;
      long len;
      /* get application name from command line */
      if ( argc != 2 ) {
            printf(" Usage:  sdst <file>  , where "
                "{file.spl} is a valid SPL file. ");
            return;
      }
      nchar = (long) strlen( argv[1] );
      nsize = (nchar+1)+3;
      splfile = (char *) malloc(sizeof(char)*nsize);
      if ( splfile==NULL ) {
            printf(" ERROR: sdst is unable to store input file-name ");
            return;
      }
      strcpy( splfile,argv[1] );
      strcat( splfile,".spl" );
/* malloc space for OSL */
    osl_dspace = (double *) malloc (MXSPC*sizeof(double));
    if ( osl_dspace == NULL ) {
      printf("Cannot malloc requested space for OSL ");
      return;
      }
/* initialize OSL with 1+MXSUB models */
    ekkdsca( &rc,osl_dspace,MXSPC,1+MXSUB );
      if ( rc>299 ) return;
/* initialize Stoch_Data with maximum MXSCN scenarios */
    scninit( &rc,&s_data,MXSCN );
      if ( rc>1 ) return;
```

```
/* Read SP-file */
    scngtsd( &rc,s_data,splfile );
      if ( rc>1 ) return;
/* Declare stochastic model (full model) */
    scndscm( &rc,s_data,0,0,&s_model,&nblks,1 );
      if ( rc>1 ) return;
/* Declare full model in OSL with room for nblocks */
    nmod = 1;
    ekkdscm( &rc,osl_dspace,nmod,nblks );
      if ( rc>299 ) return;
/* Solve -- user can set options in scnsolv.defaults */
    scnsolv( &rc,s_model,osl_dspace,nmod,OPT,MXSUB );
      if ( rc>1 ) return;
/*
 * Example of User computation using scenario tree and OSL
 *
 */
      {
            usrStoch *stree;
            usrNodeData *ndata;
            usrCoreData *cdata;
            usrNode *pnode;
            double *node_obj;
            /* locals */
            long in, il, ibas, jj;
            double nodvalue, scenvalue, mean=0.0, variance=0.0;
            long ii,is, it;
            double *dnobj,*dcobj,*dsoln,dprob;
            /* get core data */
            scncdat( &rc,s_data,&cdata );
            /* get stochdata information */
            scnsget( &rc, s_model, &stree ); if ( rc>1 ) return;
            /* malloc node objective value array */
            node_obj = (double *) malloc ( stree->nnode*sizeof(double));
            memset(node_obj,0,stree->nnode*sizeof(double));
            /* get osl pointers to solution array */
            ekkgtmi( &rc,osl_dspace,nmod );
            ekknget( &rc,osl_dspace,osln,OSLNLN );
            if ( rc>299 ) return;
            dsoln = osl_dspace+NCOLSOL-1;
/*
 * Loop over nodes and compute objective value
 * once for each node.
 */
            for (ii=0;ii<stree->nnode;ii++) {
```

```
                pnode = stree->node + ii;
                /* get core objective */
                it = pnode->stag;
                dcobj = cdata->dobj + cdata->colstart[it-1]-1;
                /* node offset in OSL model */
                ibas = pnode->cbas;
                for(jj=0;jj<pnode->ncol;jj++){
                      node_obj[ii] += dsoln[ibas+jj] * dcobj[jj];
                }
                if ( in > 1 ) {
                   /* get node objective */
                   scnndat( &rc, s_data, in, &ndata );
                   dnobj = ndata->dobj;
                   for(jj=0;jj<pnode->ncol;jj++){
                      node_obj[ii] += dsoln[ibas+jj] * dnobj[jj];
                   }
                }
          }
/*
 * For each leaf node,
 *      go up the tree adding in node_obj value along the way.
 */
          printf("      Distribution of Optimal Values    ");
          for(ii=0;ii<stree->nleaf;ii++) {
                scenvalue = 0.0;
                in = stree->leaf[ii];         /* name of leaf node */
                pnode = stree->node + in-1;   /* pointer to leaf info */
                is = pnode->scen;             /* leaf scenario */
                dprob = pnode->prob;          /* leaf probability */
                while (in>0) {
                      scenvalue += node_obj[in-1];
                      in = pnode->par;       /* parent pos'n in list */
                      if (in>0) pnode = stree->node + in - 1;
                }
                printf("    Scenario %d value %f probability %f  ",
                    is,scenvalue,dprob);
                mean += dprob*scenvalue;
                variance += dprob*scenvalue*scenvalue;
          }
          variance -= mean*mean;
          if (variance < 0.0) variance=0.0;
          printf("       Mean Value:       %f  ",mean);
          printf("       Standard Deviation:    %f  ",sqrt(variance));
   }
/* Free data structures */
```

```
    scnfree( s_model ); scnfree( s_data );
}
```