stocext

Table of Contents

IBM Stochastic Extentions 1
Stochastic Extensions Front Matter 3
OSL Stochastic Extensions G & R Roadmap 5
Table of Contents 6
IBM Stochastic Extensions Users Guide 8
IBM Stochastic Extensions Reference 23
Example Problems and Code 51
Bibliography 83

[User's Guide | Download | Bottom of page]

IBM Optimization Library Stochastic Extensions

Introduction

Stochastic programming supports decision making under uncertainty. It is a methodology for bringing uncertain future scenarios into the traditional decision making framework of linear programming. Just as linear programming models the optimal allocation of constrained resources to meet known demands, stochastic programming models the allocation of today's resources to meet tomorrow's unknown demands in such a way that the user can explore the trade offs with respect to expected risks and rewards and make informed decisions.

Users seeking the greater modeling power and flexibility of stochastic programming come up against high hurdles: there is more input data to be managed, the optimization problems are very large, and the solution arrays are larger and more difficult to analyze.

IBM Stochastic Extensions offers advanced new technology to help the user meet the demands of modeling very large, multi-stage stochastic programs. While the IBM Stochastic Solutions product has an easy-to-use solver that operates from the command line, as well as a starter set of callable modules, Stochastic Extensions provides advanced programming capability in conjunction with the IBM Optimization Library. It supports the Stochastic Mathematical Programming System (SMPS) input format for multistage stochastic programs as well as accepting model data from the application through a set of friendly yet robust stochastic data structures. The flexible nested decomposition solver is robust and fast, and allows the developer total control over partitioning large master problems into smaller sub-problems.

The IBM Stochastic Extensions requires the IBM Optimization Library to provide the underlying solver capabilities. To use the flexible nested decomposition solver, TCP/IP must also be installed and running on the user's system.

The Stochastic Extensions Subroutine Modules

Highly advanced modeling development is supported by the IBM Stochastic Extension Subroutine Modules. These are designed to facilitate the integration of IBM Stochastic Extensions with IBM Optimization Library applications. These subroutine modules are accessed by constructing specialized driver programs. These functions support advanced model generation, control of solver operation through callbacks, and analysis of solution data. Input modules support in-core modeling of stochastic programs, user callbacks monitor solver progress, and data access modules retrieve the solution for use in application processing.

Guide, Reference and Examples

The Guide and Reference is divided into three parts. The User's Guide contains basic information to help users formulate, solve, and analyze stochastic programming problems. The Reference Section contains detailed information about calling sequences, data structures, and messages. The Examples contains example problems and several driver programs.

Readers seeking a fuller introduction to the ideas and techniques of stochastic programming should consult the recently published texts by Birge and Louveaux, and Kall and Wallace .

If you would like to have softcopy of the html user guide on your PC, click here to download the zipped postscript HTML (179 KB)

If you prefer hardcopy documentation, click here to download a postscript (1100KB) or zipped postscript (294KB) version.

Notes:	1. Viewing these postcript files is problematical. An inability to view them satisfactorily does not necessarily mean that they will not print correctly. 2. There are no hypertext links in the postcript files, and so the cross references are undetectable.

[**Top of Page | Previous Page | Stochastic Extensions User's Guide**]

IBM HOME ORDER PRIVACY LEGAL CONTACT IBM

IBM Optimization Library
Stochastic Extensions
Guide and Reference

© IBM Corp., 1998. All rights reserved.

Notices
Roadmap
Stochastic Extensions Users Guide
Stochastic Extensions Reference
Example Problems and Code
Bibliography
Complete Table of Contents

Notices

Before using this information and the product it supports, be sure to read the following.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights or other legally protectible rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

A separate license is required for each machine on which the licensed program materials will be used. That is, a separate license for the IBM Optimization Library is required for each machine on which an application program incorporating subroutines from the library is to be run.

Portions of the GENQMD subroutine developed by J. Alan George and Joseph W.H. Liu of the University of Waterloo are incorporated into the Optimization Library and are used with the permission of the authors.

[**Previous Page** | **Road Map** | **Top of Page**]

IBM HOME ORDER PRIVACY LEGAL CONTACT IBM

Stochastic Extensions Guide and Reference Roadmap

- Stochastic Extensions User's Guide
  Overview SMPS Format for Stochastic LPs
  Flexible Nested Decomposition Solver Quadratic Solver Parametric Analysis
- Stochastic Extensions Reference
  Data Structures: usrCoreData usrStoch usrNode usrNodeData
  The Stochastic Library Extensions Modules
  Messages History and Acknowledgements
- Example Problems and Code
  Sample Problems: KandW3 PROD_MIX LandS HYDRO CAP
  Guide to the Driver Source Codes - Illustrating: Tree Construction
  SMPS Input Direct Solution by exsolv Post Solution Processing
  Drivers: exsmps exsgbds exsolv exnest exsdst exsgbdt
- Bibliography Complete Table of Contents

[Previous (Front Matter) | Next (Users Guide)]

Complete Table of Contents

[User's Guide | Reference Section | Sample Problems | Sample Code | Bibliography]

- Stochastic Extensions User's Guide
  - Overview of IBM Stochastic Extensions
    - The Stochastic Extensions Modules
    - Scenarios Data Structure
    - Organization of Storage: the SPL-file
    - Stochastic Models
    - Building Scenario Trees from Simulations
    - User Access to Stochastic Extensions Data Structures
  - Stochastic Mathematical Programming System (SMPS) Format for Stochastic Linear Programs
    - Problem Statement SMPS Header Line.
    - SMPS Format for Linear Programs
      - Core File Data line File layout Example
      - Time File Example
  - Stoch File
    - Stoch file - header lines
    - Independent Discrete Distributions Format
    - Uniform Distribution Format
    - Normal Distribution Format
    - Block Independent Distributions Format
    - Linear Transformations Format
    - Scenarios Example Event Tree Format
  - The Flexible Decomposition Solver
    - The Quadratic Solver
    - Parametric Analysis
- Reference Section
  - Data Structures
    - usrCoreData usrStoch usrNode usrNodeData
  - The Stochastic Extensions Modules
    - Select module:
  - Messages
  - History and Acknowledgements
- Example Problems and Code
  - A collection of sample problems
    - Problem KandW3 Statement Formulation Data Solution
    - Problem PROD_MIX Statement Formulation Data Solution
    - Problem LandS Statement Formulation Data Solution
    - Problem HYDRO Statement Formulation Data Solution
    - Problem CAP Statement Formulation Data Solution
  - Guide to the Driver Source Codes
    - Using the Tree Construction Subroutines
    - Using the SMPS Input format
    - Direct Solution by exsolv
    - Post Solution Processing

- Sample Driver Coding
  exsmps.c exsgbds.c exsolv.c exnest.c exsdst.c exsgbdt.c
- Bibliography

[**Top of Page | Previous (Bibliography) | There is no Next File | Roadmap**]

Stochastic Extensions Users Guide

- Overview of Stochastic Extensions

  The Stochastic Extensions Solver The Stochastic Extensions Modules

  Formulating the Problems
  Scenarios Data Structure

  Organization of Storage: the SPL-file
  Stochastic Models

  Building Scenario Trees from Simulations

  User Access to internal Data Structures
- Stochastic Mathematical Programming System (SMPS) Format

  Header Line
  Problem Statement

  SMPS Format for LPs: Core File Time File
  Stoch File Header lines
  Distributions Linear Transformations Scenarios
- The Flexible Nested Decomposition Solver
  The Quadratic Solver Parametric Analysis
- Bottom of Page

Overview of Stochastic Extensions

Stochastic Extensions provides functional support in modeling, solving, and analysing solution data for multiperiod stochastic programs. Key new features in the Stochastic Extensions are:

- A flexible nested decomposition solver, allowing the solution of much larger stochastic programs.
- Solution of quadratic stochastic programs.
- Parametric solution with perturbed bounds or costs vectors.

The modeling phase includes:

- **Core Problem.** Specification of a *Core* linear or quadratic program.
- **Time Stages.** Identification of *Time Stages*.
- **Stochastics.** Definition of the *Stochastic* information for the distribution of the random variables.

The solver phase includes:

- **Model Size.** Identification of the size of the model.
- **Decomposition.** Decomposition of the model into subproblems.
- **Algorithm.** Invocation of the *Nested L-shaped* (Benders) decomposition algorithm.
- **Termination.** Interpretation of the algorithm's termination criteria.
- **Parametric.** Invocation of the parametric procedure to generate a frontier of solutions.

The analysis of solution data is a complex issue. The number of ways the user can implement the functions of the IBM Optimization Library Stochastic Extensions is so great that it is difficult to anticipate all possible user requirements. The Solver provides access to:

- **Activities.** column and row solution activity levels by scenario and stage, and
- **Distribution.** distributions of problem data items (such as the value of the objective).

The callable modules provide complete access to all solution data, including dual information.

The Stochastic Extensions Solver

Most users will find that the easiest way to use the IBM Stochastic Extensions is through the Solver. This Solver can perform many different functions, selected by values supplied as command line arguments. The Solver is documented in the Solver Reference.

The Stochastic Extensions Modules

The motivating principle of IBM Stochastic Extensions is to free the user from the complex task of managing multistage stochastic data structures common to all stochastic programs. To implement this aim, IBM Stochastic Extensions provides several independent clusters of user callable modules corresponding roughly to initialization and input, tree generation, model formation and data transfer, solution via Benders decomposition, and user-access to data and information. These modules are accessed by writing a driver program. Example drivers are provided. Here we give an overview of the basic structure of a stochastic extensions driver program.

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 ekks_InitializeApplication, which returns a pointer to the Stoch_Data data type. Every Callable Module 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 IBM Stochastic Extensions driver consists of five subroutine calls:

```
#include "ekkstoch.h"
Stoch_Data *stoch_data;
    ekk_Initialize();
ekks_InitializeApplication(&rc,&stoch_data,...);
ekks_CreateCore(&rc,stoch_data,... );
for(s=0;s<max_s;s++)
{
    ekks_AddScenario(&rc,stoch_data,s,...);
}
```

```
ekks_BendersLSolve(&rc,stoch_data,osl_dspace,...);
for(s=0;s<max_s;s++)
{
    ekks_GetSolution(&rc,stoch_data,s,soln,...);
    printf( ... , soln);
}
```

The call to ekks_InitializeApplication initializes the stoch_data pointer to the Stoch_Data object. The calls to ekks_CreateCore (declare core file) and ekks_AddScenario (add a scenario) pass the problem data to the Stoch_Data object. A call to ekks_BendersLSolve declares and solves the stochastic problem with a Benders decomposition method implemented in the IBM Optimization Library subroutines. The subroutine ekks_GetSolution accesses scenario solutions one-by-one after which they may be processed by the user (for example, printed).

Formulating Stochastic Programming Problems

The formulation of stochastic programs is a sophisticated undertaking that may require expert advice from application domains, probability and statistics, and the numerical modeling of decision systems. Problem examples are provided that indicate various types of formulations and analyses. These may be useful as templates.

Scenarios Data Structure

Scenarios provide the information about the probability distribution and the data needed to define the actual stochastic program in the SMPS 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. For details of this convention, see the documentation of the ekks_AddScenario subroutine.

Organization of Storage: the SPL-file

When stochastic program core and scenario data is passed, the data, consisting of matrix, row and column bounds, and objective function, is sorted into stages and stored in a specially-structured *SPL-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 SPL-file until needed by an optimization method.

SPL-file storage is managed by IBM Stochastic Extensions I/O routines. Users have access to this information through special purpose data-access routines. The SP-file may be saved (ekks_PutStochasticData), in which case it will contain all information necessary to restore (ekks_GetStochasticData) the stochastic program.

Stochastic Models

The data corresponding to a stochastic program is organized by nodes and arcs of a tree structure and stored in an SPL-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.

The data required for the definition of a stochastic program is organized 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 IBM Stochastic Extensions subroutine ekks_DescribeModel. By default, a Stoch_Data structure is considered to be itself a stochastic model containing all possible nodes.

Building Scenario Trees from Simulations

The IBM Stochastic Extensions includes a set of routines for building a scenario tree out of a sequence of simulations. Each simulation is viewed as an array of *events*, labeled as integers. When a new simulation is added (ekks_AddToTree), 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 ekks_GetScenarioFromTree). The example driver exsgbdt illustrates the use of these routines.

User Access to Internal Data Structures

The user interfaces, data structures and subroutine declarations, are declared in the header file ekkstoch.h. ekk-stoch This file should be included in every program module that calls IBM Stochastic Extensions subroutines.

The data pointed to by the structures Stoch_Data and Stoch_Model is internal and protected from the user. But all stochastic programming data is available to the user via the structures usrCoreData, usrStoch, usrNode, and usrNodeData, as referenced in the Data Structures Reference. These must be filled by special subroutine calls using: ekks_GetCoreData, ekks_GetScenarioTree and/or ekks_GetNodeData. The sample driver programs exsdst.c, and exsolv.c show examples of user data access.

Calls to user 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 ekks_GetNodeData contains only the stochastic data. It must be combined with the appropriate core data obtained by ekks_GetCoreData. 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.

Stochastic Mathematical Programming System (SMPS) Format for Stochastic Linear Programs

This section contains material from the paper "A standard input format for multiperiod stochastic linear programs", by *J.R. Birge, M.A.H. Dempster, H.I. Gassmann, E.A. Gunn, A.J. King, and S.W. Wallace*, which introduced a data convention for the automatic input of multiperiod stochastic linear programs based on the MPS standard. This material is reproduced here, with slight updating, by permission of the authors.

SMPS Header Line.

- columns 1 through 14: indicator code
- columns 15 through 24: name

Most sections contain *data lines*. Data lines are distinguished by a blank in the first column; the first word of a header line must, therefore, begin with a non-blank character. Comment lines are indicated by an asterisk (*) in the first column and may appear anywhere.

Problem Statement

A general form of the multiperiod stochastic linear program is as follows.

$$\begin{aligned} &\text{minimize}_{x1} && \boldsymbol{c}_1^T \boldsymbol{x}_1 + E_2\, Q_2(\boldsymbol{x}_1)\,, \\ &\text{subject to:} && \boldsymbol{x}_1 \in \mathbb{R}_{n_1}\,, \\ & && \boldsymbol{l}_1 \le \boldsymbol{x}_1 \le \boldsymbol{u}_1\,, \\ & && \boldsymbol{A}_1\boldsymbol{x}_1 = \boldsymbol{b}\,; \end{aligned}$$

where the functions Q_2, Q_3, **...** , Q_T are defined recursively by:

$$\begin{aligned} Q_2(\boldsymbol{x}_1) = \min_{x2} \quad & \boldsymbol{c}_2^T \boldsymbol{x}_2 + E_3\, Q_3(\boldsymbol{x}_1, \boldsymbol{x}_2)\,, \\ \text{subject to:} \quad & \boldsymbol{x}_2 \in \mathbb{R}_{n_2}\,, \\ & \boldsymbol{l}_2 \le \boldsymbol{x}_2 \le \boldsymbol{u}_2\,, \\ & \boldsymbol{A}_{2,1}\boldsymbol{x}_1 + \boldsymbol{A}_{2,2}\boldsymbol{x}_2 = \boldsymbol{b}_2\,, \end{aligned}$$

$$\begin{aligned} Q_3(\boldsymbol{x}_1, \boldsymbol{x}_2) = \min_{x3} \quad & \boldsymbol{c}_3^T \boldsymbol{x}_3 + E_4\, Q_4(\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3)\,, \\ \text{subject to:} \quad & \boldsymbol{x}_2 \in \mathbb{R}_{n_3}\,, \\ & \boldsymbol{l}_3 \le \boldsymbol{x}_3 \le \boldsymbol{u}_3\,, \\ & \boldsymbol{A}_{3,1}\boldsymbol{x}_1 + \boldsymbol{A}_{3,2}\boldsymbol{x}_2 + \boldsymbol{A}_{3,3}\boldsymbol{x}_3 = \boldsymbol{b}_3\,, \end{aligned}$$

and so on, for Q_t , $t = 4$, **...** , T - 1. In each of the forgoing equations, "E_t" represents the expectation with respect to the random variables in period number t . Finally, Q_T is defined by:

$$\begin{aligned} Q_T(\boldsymbol{x}_1, \ldots, x_{T-1}) = \min_{xT} \quad & \boldsymbol{c}_T^T \boldsymbol{x}_T\,, \\ \text{subject to:} \quad & \boldsymbol{x}_2 \in \mathbb{R}_{n_T}\,, \\ & \boldsymbol{l}_T \le \boldsymbol{x}_T \le \boldsymbol{u}_T\,, \\ & \boldsymbol{A}_{T,1}\boldsymbol{x}_1 + \boldsymbol{A}_{T,2}\boldsymbol{x}_2 + \ldots + \boldsymbol{A}_{T,T}\boldsymbol{x}_T = \boldsymbol{b}_T\,. \end{aligned}$$

The data defining this problem may be arranged in a LP formulation for a single realization of the random variables:

$$\text{objective:} \quad \sum_{j=1}^{T} \boldsymbol{c}_t^T \boldsymbol{x}_t ,$$

$$\text{constraints:} \quad \boldsymbol{x}_t \in \mathbb{R}^{n_t} \qquad \text{for } t = 1, \ldots, T ,$$

$$\boldsymbol{l}_t \le \boldsymbol{x}_t \le \boldsymbol{u}_t \qquad \text{for } t = 1, \ldots, T ,$$

$$\boldsymbol{A}_1 \boldsymbol{x}_1 = b_1 \in \mathbb{R}^{m_1} , \text{ and}$$

$$\sum_{j=1}^{t} \boldsymbol{A}_{t,j} \boldsymbol{x}_j = b_t \in \mathbb{R}^{m_t} , \text{ for } t = 2, \ldots, T .$$

All entries of the matrices $\boldsymbol{A}_{t,j}$ and vectors: $\boldsymbol{c}_t$, $\boldsymbol{l}_t$, $\boldsymbol{u}_t$, $\boldsymbol{b}_t$ may be random (although in practice all but a few entries will be deterministic). The indices $t = 1, \ldots, T$ signify the periods of the problem; to each period t there corresponds a decision vector $x_t \in \mathbb{R}^{n_t}$. The lower block-triangular constraint system expresses the typical feature of these problems--the decisions of the prior periods constrain the decision of the current period explicitly (since those decisions are known) but the decision of the current period is affected only implicitly by the feasibility and costs of possible future (recourse) decisions.

The proposed format is most easily understood by considering the above multistage LP in stages of gradually increasing levels of detail. First we regard it as a purely deterministic problem and create an input file following the MPS convention, ignoring (non-random) zero entries. This file will identify an objective vector, upper and lower bounds, a right hand side, and a block lower-triangular matrix, all expressed in the usual column-row format; we call this the *core file*.

Next we note the period structure of the problem, that certain decisions are made at certain times. Here it is necessary only to specify which rows and columns from the core file correspond to which periods. It is most simply done by indicating the beginning column and row for each period. This is done in the *time file*. Such a system relies on the proper sequencing of the core file--we require that the list of row names is in order from first period to last period and that the block lower-triangular matrix has been entered in column order: first period columns first, and last period columns last.

Finally there remains the specification of the distributions of the random entries; this takes place in the *stoch file*. The simplest case occurs when all random entries are mutually *independent*. We consider two general types of dependency among the random entries: *blocks* and *scenarios*. A block is a random vector whose realizations are observed in a single, fixed period. A *scenario* is a more general type of stochastic structure which models dependencies across periods.

The organization of the data files is similar to the MPS format. Each data file contains a number of *sections*, some of which are optional. A *header line*, or *header*, marks the beginning of each section and may contain *keywords* to inform the user that the data to follow should be treated in a special way. Each header line is divided into two fields delimited by specific columns.

SMPS Format for Linear Programs

Core File

The *core file* is sketched only briefly since it closely follows the MPS format. The core file consists of sections introduced by header lines. Data lines follow the headers.

MPS Data line for LP

A data line is divided into six fields: three name fields, two numeric fields and one indicator field.

- columns 2 and 3: indicator field
- columns 5-12: name field
- columns 15-22: first data name field
- columns 25-36: first numeric field
- columns 40-47: second data name field
- columns 50-61: second numeric field

A *name* is treated as a character string and may contain any ASCII symbol, except that data name fields must not start with a ($). Numbers with decimal point, or in scientific notation, may appear in numeric fields.

Core file layout in SMPS

1. NAME Required. The second word field is used to identify the problem.
2. ROWS Required. Each data line specifies the names of the objective and rows of the matrix in the first name field, and the type of constraint (E, L, G, N) in the code field. The list of row names must be in order from first period to last, preceded by the objective name(s).
3. COLUMNS Required. Each data line specifies the column names and the nonzero values. This must be done in column order.
4. RHS Required. Starts the RHS section. Data lines specify nonzero entries of the right hand side.
5. BOUNDS Optional. Data lines specify the column bounds using the codes LO or UP in the code field.
6. RANGES Optional. Refer to the MPS standard.
7. ENDATA Required. End of problem data.

Core file - example

```
1...5.........15........25.............40........50.............
NAME          problem name
* this is a comment line
ROWS
 N  OBJ
 L  ROW1
 E  ROW2
    -etc-
COLUMNS
1...5.........15........25.............40........50.............
    COL1      ROW1      value          ROW2      value
    COL1      ROW3      value          ROW4      value
```

```
    -etc-
    RHS
    RHS       ROW1      value          ROW2      value
    -etc-
BOUNDS
 LO BND       COL1      value
    -etc-
RANGES
    -etc-
ENDATA
```

Integer variables may be indicated by using delimiters as described in the MPS/MIP standard. for those users who want to use mixed integer programming techniques.[1]

Time File

The *time file* contains the information needed to specify the dynamic structure of the problem. It indicates which rows and columns are to be identified with which period. The first line identifies it as a time file and gives a name to the problem. The next header line consists of a single word PERIODS in the first name field, and contains the keyword LP in the second name field to identify the problem as a pure LP. The first two name fields of the data lines identify the beginning row and column names for each period with the corresponding period name in the third name field.

Time file - example

```
1...5.........15........25.............40........50.............
TIME          problem name
PERIODS
    COL1      ROW1                     PERIOD1
    COL6      ROW3                     PERIOD2
    COL8      ROW19                    PERIOD3
ENDATA
```

In this example: Columns COL1 through COL5 are PERIOD1 decision variables and COL6 and COL7 are PERIOD2 variables; rows ROW1 and ROW2 are PERIOD1 constraints and ROW3 through ROW18 are PERIOD2 constraints. All remaining rows and columns belong to PERIOD3.

Stoch File

In the *stoch file* the distributions of the random variables are specified. As mentioned, we consider three varieties of distributions: *independent*, *blocks*, and *scenarios*. Each type will be treated in separate sections of this file; each section consists of a header line followed by data lines.

Stoch file - header lines

1. STOCH - informative. Identifies a new problem with a give name in the second word field.
2. INDEP section - specifies the distribution of all independent random entries in separate sections for each type.
3.

BLOCKS section - specifies the joint distribution of all dependent random entries in separate sections for each type.

4. SCENARIOS section - specifies the scenarios.

5. ENDATA - informative. End of problem data.

Each distribution specifies the values that a core file entry takes with a certain probability. This value *replaces* the corresponding core file entry.(2)

Distributions

- Independent Discrete Distributions Format
- Uniform Distribution Format
- Normal Distribution Format
- Block Independent Distributions Format

Independent Discrete Distributions

For each discretely distributed entry one must specify the values and corresponding probabilities. The first two name fields identify the entry, and the first two numeric fields are the value field and probability field respectively. The intervening third name field contains the name of the period in which the random variable is realized. (This information is ignored by the input routine, but it is useful to have it made explicit for data consistency checking).

Independent Discrete format

```
1...5.........15........25.............40........50.............
INDEP          DISCRETE
    COL1       ROW8       6.0             PERIOD2    0.5
    COL1       ROW8       8.0             PERIOD2    0.5
    RHS        ROW8       1.0             PERIOD2    0.1
    RHS        ROW8       2.0             PERIOD2    0.5
    RHS        ROW8       3.0             PERIOD2    0.4
```

In this example the entry COL1/ROW8 takes value 6.0 with probability 0.5 and 8.0 with probability 0.5; and the right hand side of ROW8 takes values (1.0,2.0,3.0) with probabilities (0.1,0.5,0.4) respectively. Of course the probabilities associated with an entry must total one.

Uniform Distribution

The endpoints of the interval are the only relevant parameters for uniformly distributed entries. These are entered into the first two numeric fields; the third name field is blank.

Uniform Distribution Format

```
1...5.........15........25.............40........50.............
INDEP          UNIFORM
    COL1       ROW8       8.0             PERIOD2    9.0
```

In this example the random entry COL1/ROW8 is uniformly distributed over the interval [8.0, 9.0].

Normal Distribution

The normal distribution is specified by mean and variance in the first two numeric fields.

Normal Distribution Format

```
1...5.........15........25.............40........50.............
INDEP          NORMAL
    COL1       ROW8       0.0             PERIOD2    1.0
```

In this example the random variable is normal with mean 0.0 and variance 1.0.

Block Independent Distributions

The "values" of a block are actually vectors of values of the entries that make up the block, and to each value of a block there corresponds a probability. We need two sorts of data lines to describe a block. The first line, distinguished by a BL in the code field, gives the name of the block, the name of the period in which the block is realized, and the probability that the block assumes a given vector value; the following lines identify which entries of the block assume which value.

Independent Block format

```
1...5.........15........25.............40........50.............
BLOCKS         DISCRETE
 BL BLOCK1     PERIOD2    0.5
    COL1       ROW6       83.0
    COL2       ROW8       1.2
 BL BLOCK1     PERIOD2    0.2
    COL1       ROW6       83.0
    COL2       ROW8       1.3
 BL BLOCK1     PERIOD2    0.3
    COL1       ROW6       84.0
    COL2       ROW8       1.2
ENDATA
```

In this example the block, called BLOCK1, is the vector made up of the entries COL1/ROW6 and COL2/ROW8. It takes values (83.0,1.2) with probability 0.5, (83.0,1.3) with probability 0.2, and (84.0,1.2) with probability 0.3.

Linear Transformations

These are blocks whose distribution is computed as a linear transformation of another random vector with independent components, i.e., The vector v is the block whose distribution is desired; the vector u has independently distributed components of standard type, e.g., normal. We first identify the block as in the subroutine case, then the (marginal) distribution of each (independent) component of the random vector followed immediately by the corresponding column of the matrix H .

Linear Transformation format

```
1...5.........15........25.............40........50.............
BLOCKS         LINTR
 BL V_BLOCK    PERIOD2
    COL1       ROW8
    COL3       ROW6
 RV U1         NORMAL     0.0                       1.0
    COL1       ROW8       11.0
    COL3       ROW6       21.0
 RV U2         UNIFORM    -1.0                      1.0
    COL1       ROW8       12.0
    COL3       ROW6       22.0
 RV U3         CONSTANT
    COL3       ROW6       24.0
```

This example illustrates the file structure. In this case

Note that the "names" U1, U2 and U3 are irrelevant and may be left blank.

Scenarios, An Example - event tree, and the Scenarios Format

To describe *scenarios* one needs a data structure that expresses inter-period dependencies. This is best developed as a description of the distribution of a *process* vector just as one may describe a stochastic process in probability theory. We consider the random entries of the objective, bounds, and matrix as states of a process vector. Given the corresponding (finite dimensional) joint distributions of this process, Kolmogorov's construction yields a probability measure, termed the *process distribution*, on the space of trajectories. Thus, in general, we have the alternatives of describing the distribution of the stochastic process in joint or conditional state distribution form, or as a process distribution over trajectories. In the case under consideration, all this may be given a much simpler, more graphic, characterization.

To describe the process paths in the discrete state case, note that process can assume only finitely many values. A path of the process up to a given time stage may be followed by a finite collection of possible values of the process in the next time period. We think of these as *nodes* in the next period. Following Lane and Hutchinson and Raiffa, we construct an *event tree* representation of the trajectories: Represent the (unique) value of the process in the first period by a single node connected by *oriented arcs* from that node to nodes representing the values of the process in period 2. These are the *descendant* nodes of the first period node in the terminology of Birge. Each node of period 2 is connected to its descendant nodes at period 3 by individual arcs oriented in the direction of the period 3 nodes. This construction is continued by connecting each *ancestor* with its descendant nodes in the next period. Each node of this tree has a single entering arc and multiple departing arcs representing the possible next period events.

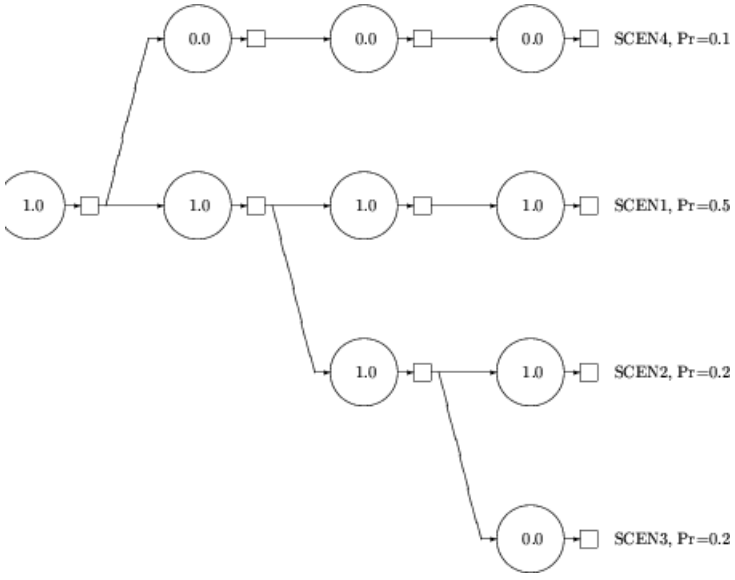
A single trajectory of the process thus corresponds to a *path* from the period 1 node to a single period T node composed of arcs oriented in the direction of increasing time, and moreover, corresponds uniquely to a single node in the last period T. There are only finitely many paths linking nodes in period 1 to nodes in period T, and to specify the distribution on the paths one needs only to attach a definite probability to each path. Hence specifying the process distribution in terms of path probabilities can be effected in this context by assigning probabilities to period T nodes of the hyperreferent tree event tree, see Figure 1.1 below.

The topology of this event tree represents the information about the process. To each arc we attach the probability that the terminal node occurs given the initial node has occurred--that is, the conditional probability of the process distribution over a period conditional on the history over all previous periods. These *arc probabilities* can be computed from the path probabilities by summing the probabilities of the paths visiting the terminal node, and then dividing by the sum of the probabilities of the paths visiting the initial node.

Conversely, if to each arc in the tree we know the conditional probability that its terminal node occurs given that its initial node has occurred, then the probability of any given path is simply the product of the arc probabilities along the path.

A decision at a given node is made only on the basis of information collected up to and including that node. The uncertainty faced by the decision maker is represented by the collection of paths that branch from this node. Thus in the following Figure, the decisions occur at the nodes and the scenarios branch after the node.

Figure 1.1: Scenarios example - event tree



In a language more specific to our application, a "path" in the tree analogy is a single "scenario". The nodes visited by the path correspond to certain values assumed by certain entries of the matrices in the core file. Thus a scenario is completely specified by a list of column/row names and values, and a probability value. Once a single given scenario is described, then other scenarios that branch from it may be described by indicating in which period the branch has occurred, and then listing the subsequent column/row names and values. It is best to work through the example shown in the preceding Figure.

There are two types of data lines. The first, signified by SC in the code field, gives the name of the scenario in the first name field and its probability in the first numeric field, and then gives the name of the scenario from which the branch occurred and the name of the period in which the branch occurred

--i.e., the first period in which the two scenarios *differ*-- in the second and third name fields, respectively. A scenario that originates in period one is indicated by ROOT in the name field. The next data lines give the column/row values assumed by the scenario.

Scenarios Format

```
1...5.........15........25.............40........50.............
SCENARIOS     DISCRETE
 SC SCEN1     ROOT      0.5            PERIOD1
    COL1      ROW2      1.0
    COL2      ROW3      1.0
    COL3      ROW4      1.0
    COL4      ROW5      1.0
 SC SCEN2     SCEN1     0.2            PERIOD3
    COL3      ROW4      1.0
    COL4      ROW5      1.0
 SC SCEN3     SCEN2     0.2            PERIOD4
    COL4      ROW5      0.0
 SC SCEN4     SCEN1     0.1            PERIOD2
    COL2      ROW3      0.0
    COL3      ROW4      0.0
    COL4      ROW5      0.0
ENDATA
```

This is a description of the distribution of four entries: COL1/ROW2, COL2/ROW3, COL3/ROW4, COL4/ROW5, which for convenience we denote here as d1, d2, d3, and d4, respectively. Note that in PERIOD4 there are two nodes for the "state" d4 equals 0.0 and two for d4 equals 1.0, and similarly in PERIOD3 two nodes for d3 equals 1.0. This is because a node is distinguished by the information that one has collected concerning the path up to and including that node. Thus in PERIOD3 the two nodes are distinguished because in scenario SCEN1 one *knows* that the final state is d4 equals 1.0, whereas in SCEN2 the outcome of d4 is in doubt.

The Flexible Nested Decomposition Solver

The decomposition method implemented in the IBM Stochastic Extensions is the L-shaped method of van Slyke and Wets. The nested decomposition solver is a very powerful method for solving large stochastic programs.

The solution procedure consists of a nested L-shaped, or Benders, decomposition phase, and if needed, a Simplex method phase. The decomposition may not terminate in a reasonable number of iterations, for this reason it is often useful to follow Benders with the Simplex method. This is automatically done if the number of major cycles exceeds Benders_Major_Iterations.

In the decomposition phase, the problem is decomposed into subproblems according to one of several user-specified criteria: Problem_Size, Maximum_Subproblems, or Cut_Stage.

- If Problem_Size is greater than zero, then subtrees are generated by adding nodes from leaf to root until the number of rows in the subtree is just larger than Problem_Size. The nodes are collected in such a way as to create a submodel tree with one root and no cycles.
- Otherwise, if Cut_Stage is nonzero then the node tree is cut at Cut_Stage to form a subproblem tree with one master and one or more subtrees.

- If both Problem_Size and Cut_Stage are zero, then exactly Maximum_Subproblems are created as if Problem_Size was equal to 0.9 times the number of rows divided by Maximum_Subproblems.

If there are more subproblems than the maximum number of subproblems allowed, Maximum_Subproblems, then the subtrees are divided among the subproblems as evenly as possible. In the case of a cut at a time stage Cut_Stage, 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. If the tree is not decomposed by time stage, then no augmented master is generated.

In nested decomposition, subproblems can generate both cuts and proposals, depending on their position in the subproblem tree. The implementation requires that all child subproblems report OPTIMAL before the subproblem will generate a cut for its parent, and that a subproblem generates a proposal for its children only when it has received a new proposal from its parent. (This is the so-called Fast Forward, Fast Back protocol.) Only optimal cuts and proposals are generated.

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 root subproblem with all optimality and feasibility cuts so far generated, and posting the proposed root solution and objective value, if one exists.
2. Solving the full subproblem tree with the root proposal (possibly iterating many times between intermediate subproblems) and generating subproblem optimality or feasibility cuts to the root. 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 root proposal) and the root objective value (a lower bound) for convergence to within Optimality_Gap.

At the termination of the nested decomposition phase, if optimality was not attained, then the Simplex phase begins.

Experience suggests that the solution progress is most affected by the Problem_Size and the maximum number of major iterations. Large subproblems take a long time to solve. A large number of small subproblems will spend a long time resolving root proposals.

Those parameters not yet described affect to the solution procedures applied within the Benders decomposition. These procedures are implemented in IBM Optimization Library subroutines, and the parameters refer to the calling sequences of these subroutines, which are documented in the Optimization Library Users Guide and Reference. A method or type setting of zero means that the procedure will not be called. For example, the parameter EKKCRSH_method refers to the parameter passed to 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.

The Quadratic Solver

Quadratic stochastic programming problems may be solved using the L-shaped method, just like any other convex stochastic program. The Stochastic Extensions includes the ability to specify a diagonal quadratic matrix in invocation of the ekks_CreateCore subroutine, and to change the problem type to quadratic. This causes the flexible nested decomposition solver to use the Optimization Library quadratic solver.

Parametric Analysis

Once a problem has been solved with the L-shaped method, it can be quickly resolved with perturbed bounds or objectives. Invoking the parallel solver will cause the program to take equal steps from the initial bounds and objective toward the perturbed bounds and objective. The user specifies the perturbation and the number of steps. At each step, the bounds and objective are updated and the Optimization Library simplex solver is called with the old solution as a starting basis. Parametric performs quite well even for very large problems, once the nested decomposition solver has found the initial solution.

Footnotes

1. mixed integer programming techniques.[(1)]
   This is not supported in the present version of the IBM Stochastic Extensions. [August 1, 1998]
2. corresponding core file entry.[(2)]
   The IBM Stochastic Extensions Stoch file format allows for a keyword ADD in the third name field indicating that the values for the random variables are added to the corresponding core file entry.

[**Top of Page | Previous (Roadmap) | Next (Reference Section) | Table of Contents**]

Stochastic Extensions Reference

- Data Structures
  usrCoreData usrStoch usrNode usrNodeData
- The Stochastic Extensions Modules
- Messages
- History and Acknowledgements
- Bottom of Page

Data Structures

IBM Stochastic Extensions internal data structures are not accessible to the user. The user gains access to copies of the structures in the format described in the file ekkstoch.h by the functions ekks_GetCoreData (which fetches a copy of usrCoreData struct), ekks_GetScenarioTree (which fetches a copy of usrStoch struct) and ekks_GetNodeData (get usrNode struct). This section describes the data structures.

usrCoreData

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

```
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 the IBM Optimization Library).

If, for example, the rows with indices 1, 2, 3, and 4 are first stage rows, and the rows with indices 5, 6, 7, 8, 9, and 10 are second stage rows, then the first four entries in the row bounds arrays drlo and drup are bounds for the first stage rows, and the next six entries are bounds for the second stage rows. Of course, the ordering here may not correspond to the original ordering in the calling sequence for the ekks_CreateCore 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: In this case the elemstart array has 11 entries. For example, if 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 "by indices," and 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 Stoch and Node structures are filled by a call to ekks_GetScenarioTree.

```
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 (first=1) */
   void *user;     /* (optional) pointer to user structure */
} usrStoch;
```

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

usrNode

The Node 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 (first=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 Stoch 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).

usrNodeData

The NodeData 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 ekks_GetNodeData.

```
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 IBM Stochastic Extensions 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 structure of the matrix is similar to that of the matrix in the CoreData structure, 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 row offset rbas for the current node.

The Stochastic Extensions Modules

Select module:

ekks_AddEvent

```
  Takes an event for a INDEP_DISCRETE distribution and adds it as
  a scenario.

  FUNCTION PROTOTYPE:
    void ekks_AddEvent (long *rc, Stoch_Data *stoch, Tree **treep,
         long *index, double dp, long replace );

  INPUT:
    Stoch_Data *stoch  pointer to a Stoch_Data struct
    Tree **treep       address of pointer to event tree under
                         construction (must be NULL in first call)
    long *index        pointer to event indices
    double dp          probability
    long replace       0=>add to, 1=>replace core values
```

```
OUTPUT:
  long *rc           pointer to return code (2 is fatal)

RETURNS:
  void
```

ekks_AddScenario

```
Adds a scenario to stochastic problem, sets the scenario probability,
the branching information, and stores the scenario data by node in
an SPL file.

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          scenario value zeroed    warning   1

The LP data is NOT kept in core memory. (It may be retrieved
for the user by a call to ekks_GetNodeData.)  The LP matrix
must be in "indices" sparse format.

If replace = 0:
    LP data should be the "difference" of the intended scenario
    LP data and the core LP data.
If replace = 1:
    Scenario LP data should "replace" the core LP data. In this
    case, the subroutine calculates the difference for you.

FUNCTION PROTOTYPE:
  void ekks_AddScenario (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, long replace );

INPUT:
  Stoch_Data *stoch   pointer to Stoch_Data struct initialized
                         by ekks_InitializeApplication
  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 *dcup        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
  long replace        0=>add to, 1=>replace core values

Output:
  long *rc            pointer to return code (2 is fatal)

RETURNS:
  void
```

ekks_AddToTree

```
Updates scenario tree with data values in arr[].

If arr[] is already in tree, then ekks_AddToTree increments
counters and returns the original scenario number, otherwise
it creates a new scenario from data values in arr[], and returns
the newly created scenario number.

FUNCTION PROTOTYPE:
  long ekks_AddToTree (long *rc, Tree *tree, long *arr, long narr,
       double dp);

INPUT:
  Tree *tree       pointer to tree initialized by
                      ekks_InitializeTree
  long *arr        point to array of integer values
  long narr        number of data values (stages) in arr[]
  double dp        probability of events

OUTPUT:
  long *rc        pointer to return code:
                     1 if maximum number of samples exceeded
                     2 if tree not initialized or corrupted

RETURNS:
  scenario number as described above
```

ekks_BendersLSolve

```
Applies Benders L-shaped decomposition to the stochastic model,
with cut made at the stage number IPCUT.

METHODOLOGY:
  A single master problem is created and up to MAXMOD subproblems
  corresponding to a cut made at stage IPCUT; that is, the master
  problem contains all rows and columns for stages prior to IPCUT.
  If MAXMOD is not sufficient to hold all the subproblems, the
  subproblems are distributed as evenly as possible among the
  MAXMOD models.

  The program performs MAXITS iterations of Benders decomposition,
  terminating if optimality, infeasibility, or unboundedness
  is detected.

FUNCTION PROTOTYPE:
  void ekks_BendersLSolve (long *rc, Stoch_Model *smodel,
       long ipcut, long maxmod, long maxits, double rmaxmin,
       long inwmt, long icrsh, long ilpdc, long iscal,
       long iprsl, long iprsltype, long iprslunit, long isalg,
       long largebounds, long *status, double *optgap );

INPUT:

  Stoch_Model *smodel   pointer to stochastic model created by OSL
  long ipcut            period at which Benders cuts are made
  long maxmod           maximum number of subproblems
  long maxits           maximum number of Master iterations
  double rmaxmin        -1.0 => maximize, 1.0 =>  minimize
  long inwmt            compress model matrices into a new copy,
                           0 => do not compress
  long iscal            0=>don't, 1=>do  scale each submodel
  long icrsh            type of crash to perform before first
                           solution of submodels (0 => none)
  long ilpdc            0=>don't, 1=>do  do LP decomposition for
                            each submodel
  long iprsl            0=>don't, 1=>do presolve each submodel
  long iprsltype        type of presolve
  long iprslunit        presolve unit
  long isalg            specifies simplex algorithm to be used by
                          EKKSSLV  for first solution of submodels
  long largebounds      0=>don't, 1=>do add large bounds to master
  double *optgap        pointer to maximum optimality gap before
                           stopping BendersLSolve

OUTPUT:
  long *rc             pointer to return code (code >1 => fatal error)
  long *status         pointer to status; value = 0=>optimal,
                       1=>infeasible, 2=>unbounded, 3=>unfinished
  double *optgap       pointer to Optimality gap at termination

RETURNS:
  void
```

ekks_ChangeProbability

```
Updates leaf node probabilities with new scenario probabilities.

Note: The original scenario probabilities are unaffected.
      These can be restored by a call to ekks_ChangeProbability
      with a NULL usrProb parameter.

FUNCTION PROTOTYPE:
  void ekks_ChangeProbability ( long *rc, Stoch_Data *stoch,
       double *usrProb );

INPUT:
  Stoch_Data *stoch    pointer to Stoch_Data struct initialized
                          by ekks_InitializeApplication
  double *usrProb      pointer to array of new probabilities

OUTPUT:
  long *rc             pointer return code (2 is fatal)

RETURNS:
  void
```

ekkse_CloseCallBack

```
Clears registration of a call back function previously registered
by a call to ekkse_RegisterCallBack

FUNCTION PROTOTYPE:
  void ekkse_CloseCallBack (int *cbid);

INPUT:
  int *cbid    pointer to user exit id number

RETURNS:
  void
```

ekks_CreateCore

```
Sorts core model row and column indices by stages, groups core
model data into blocks corresponding to stages, and stores all
model information, sort keys, and grouped data into an SPL file.

The data may be accessed by a call to ekks_GetCoreData.

FUNCTION PROTOTYPE:
  void ekks_CreateCore (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,
       double *dqdg);
```

```
INPUT:
  Stoch_Data stoch     pointer to Stoch_Data struct initialized
                          by ekks_InitializeApplication
  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 *dcup         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
  double *dqdg         array of quadratic matrix diagonal entries

OUTPUT:
  long *rc             pointer to return code (2 is fatal)

RETURNS:
  void
```

ekkse_CutModelAtStage

```
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.

FUNCTION PROTOTYPE:
  void ekkse_CutModelAtStage (long *rc, Stoch_Model *smodel,
       long *ipcut, long maxmodels);

INPUT:
  Stoch_Model *smodel      pointer to Stoch_Model to be cut
  long ipcut               period at which model is cut
  long maxmodels           maximum number of models (Master + Subs)

OUTPUT:
  long *rc                 pointer to return code (2 is fatal)

RETURNS:
  void
```

ekkse_CutModelByNode

Generates "depth-first" partition of the model. Number of nodes is controlled by the value of *maxNodes*. The minimum size of a model is determined by dividing 0.9 times the value of *maxNodes* into the number of rows of the problem.

The method adds a scenario to a model, then starting from the leaf and going back to the root, all children of the parent of the node are added until the minimum size is reached.

Then models with smaller than minimum are added to their masters, until *maxNodes* models remain.

```
FUNCTION PROTOTYPE:
  void ekkse_CutModelByNode (long *rc, Stoch_Model *smodel,
       long maxNodes, long **masters);

INPUT:
  Stoch_Model *smodel     pointer to Stoch_Model to be cut
  long maxNodes            maximum number of nodes
                              to be generated

MODIFIED:
  long **masters           list of master nodes for submodels

OUTPUT:
  long *rc                 pointer to return code (2 is fatal)

RETURNS:
  void
```

ekkse_CutModelByRow

Generates "depth-first" partition of the model. Size of models is controlled by the value of *minNumRows*.

The method adds a scenario to a model, then starting from the leaf and going back to the root, all children of the parent of the node are added until *minNumRows* is reached.

```
FUNCTION PROTOTYPE:
  void ekkse_CutModelByRow (long *rc, Stoch_Model *smodel,
       long minNumRows, long **masters);

INPUT:
  Stoch_Model *smodel      pointer to Stoch_Model to be cut
  long minNumRows          minimum number of rows per model

OUTPUT:
  long *rc                 pointer to return code (2 is fatal)
  long *masters            list of master nodes for each submodel

RETURNS:
  void
```

ekkse_DefineParametricModel

```
Create parametric stochastic model

The user supplies arrays which are 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.

FUNCTION PROTOTYPE:
   void ekkse_DefineParametricModel (long *rc, Stoch_Model *smodel,
        double *cdelta, double *lrdelta, double *urdelta,
        double *lcdelta, double *ucdelta, long mask );

INPUT:
  Stoch_Model *smodel  pointer to stochastic model initialized by OSL
  double *cdelta       array of objective coefficient perturbations
  double *lrdelta      array of lower row bound perturbations
  double *urdelta      array of upper row bound perturbations
  double *lcdelta      array of lower column bound perturbations
  double *ucdelta      array of upper column bound perturbations
  long mask            determines which perturbations are used:
                        mask = C*1 + LR*2 + UR*4 + LC*8 + UC*16
                        where C, LR, UR, LC, and UC are 1 or 0
                        depending on whether a perturbation of the
                        corresponding objective coefficient, row, or
                        column bound is (1) or is not (0) to be made

OUTPUT:
  long *rc             pointer to return code (2 is fatal)

RETURNS:
  void
```

ekks_DescribeModel

```
Creates model node information structures and offsets from
user-supplied nodelist.

If nodelist is empty (nnodes=0) then all nodes in the Stoch_Data
struct 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 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.  Number of blocks to be passed is returned.

FUNCTION PROTOTYPE:
  void ekks_DescribeModel (long *rc, Stoch_Data *stoch, long nnodes,
       long *usrndlist, Stoch_Model **smodel,
       long *nblocks, long calcprob );

INPUT:
  Stoch_Data *stoch      pointer to Stoch_Data structure
  long nnodes            number of nodes
  long *usrndlist        pointer to list of nodes to be included
                            in model
  long calcprob          0=>don't, 1=>do recompute node probabilities

OUTPUT:
  Stoch_Model **smodel   pointer to new stochastic model
  long *nblocks          number of blocks in model

RETURNS:
  void
```

ekkse_DistributionAtTreeNode

```
Finds the node corresponding to arr[0, ..., nstg-1], and reports
the conditional distribution to standard output.

FUNCTION PROTOTYPE:
  void ekkse_DistributionAtTreeNode (long *rc, Tree *tree,
       long *arr, long narr);

INPUT:
  Tree *tree       pointer to Tree struct initialized by
                      ekks_InitializeTree
  long *arr        pointer to array of integer values
  long narr        number of data values (stages) in arr[]

OUTPUT:
  long *rc         return codes:
                      1 if maximum number of samples exceeded
                      2 if tree not initialized or corrupted

RETURNS:
  void
```

ekks_FreeStructure

```
Free all data structures and temporary files associated with
the Stoch_Data or Stoch_Model structure.

FUNCTION PROTOTYPE:
```

```
  void ekks_FreeStructure (Stoch *stoch);

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

RETURNS:
  void
```

ekks_GetCoreData

```
Copy core data pointers into the usrCoreData struct.

If *pcore is NULL, then a new usrCoreData struct is malloc'ed
and the pointer returned.

The data pointed to by usrCoreData pointers is stable.
(It is the original data as formed by ekks_CreateCore.)

FUNCTION PROTOTYPE:
  void ekks_GetCoreData (long *rc, Stoch_Data *stoch,
       usrCoreData **pcore );

INPUT:
  Stoch_Data *stoch      pointer initialized by ekks_CreateCore
                             and ekks_InitializeApplication

OUTPUT:
  usrCoreData **pcore    address of pointer to type usrCoreData

RETURNS:
  void
```

ekkse_GetDualSolution

```
Put scenario duals at location pointed to by dualsout[].
(Must be allocated by user.)

FUNCTION PROTOTYPE:
  void ekkse_GetDualSolution (long *rc, Stoch_Model *smodel, long s,
       long colrowtype, double *solution, long *indx);

INPUT:
  Stoch *stoch       pointer to the Stoch_Model or Stoch_Data struct
  long s             scenario number for which solution
                       info is desired
  long colrowtype    type of solution values desired
                       (1 => rows, else columns)

OUTPUT:
  long *rc           pointer to return code (2 is fatal)
```

```
    double *solution   pointer to the solution
    long *indx         pointer to the indices

  RETURNS:
    void
```

ekks_GetNodeData

```
  Get stochastic node data into usrNodeData struct

  Allocates a new usrNodeData struct.  Accesses stochastic matrix
  and costs/bounds data from SPL file and places them in volatile
  working memory.

  Costs and bounds are expanded to their dense form.
  usrNodeData pointers are appropriately set.

  Note: ALL matrix and costs/bounds arrays are volatile!
        The next call to a Stochastic Solutions/Extensions
        routine WILL OVERWRITE THEM! (To protect yourself,
        copy them to a memory region that you maintain.)

  Note: The matrix and costs/bounds arrays contain only the
        stochastic part of the data. They must be ADDED to
        the relevant core data.

  FUNCTION PROTOTYPE:
    void ekks_GetNodeData (long *rc, Stoch_Data *stoch, long node,
         usrNodeData **punode );

  INPUT:
    Stoch_Data *stoch     pointer to a Stoch_Data struct initialized
                             by ekks_InitializeApplication
    long node             node number to be accessed

  OUTPUT:
    long *rc              pointer to return code (2 is fatal)
    usrNodeData **punode  pointer to usrNodeData

  RETURNS:
    void
```

ekks_GetPointerToDistribution

```
  Mallocs an independent data struct and sets it to point to the
  information collected in a ekks_ReadMPS reading of an
  INDEPENDENT/DISCRETE format.

  FUNCTION PROTOTYPE:
    void ekks_GetPointerToDistribution ( long *rc, Stoch_Data *stoch,
```

```
        Independent **s_indep );

  INPUT:
    Stoch_Data *sdata          pointer to a Stoch_Data struct

  OUTPUT:
    long *rc                   pointer to return code (2 is fatal)
    Independent **s_indep      pointer to an Independent struct

  RETURNS:
    void
```

ekks_GetPointerToReducedCosts

```
  Gets a pointer to the solution region for a stochastic model.

  FUNCTION PROTOTYPE:
    void ekks_GetPointerToReducedCosts (long *rc, Stoch *stoch,
         long *nmod, double **redcosts, long *ncol);

  INPUT:
    Stoch *stoch        pointer to Stoch_Model or Stoch_Data struct
    long *nmod          pointer to the number of the model

  OUTPUT:
    double **redcosts   pointer to solutions region
    long *rc            pointer to return code (2 is fatal)
    long *ncol          pointer to the number of columns

 RETURNS:
      void
```

ekks_GetPointerToRowDuals

```
  Gets a pointer to the solution region for a stochastic model.

  FUNCTION PROTOTYPE:
    void ekks_GetPointerToRowDuals (long *rc, Stoch *stoch, long *nmod,
         double **rowduals, long *nrow);

  INPUT:
    Stoch *stoch        pointer to Stoch_Model or Stoch_Data struct
    long *nmod          pointer to the number of the model

  OUTPUT:
    double **rowduals   pointer to the solutions region
    long *rc            pointer to return code (2 is fatal)
    long *nrow          pointer to the number of rows

 RETURNS:
```

```
     void
```

ekks_GetPointerToRowSolution

```
  Gets a pointer to the solution region for a stochastic model.

  FUNCTION PROTOTYPE:
    void ekks_GetPointerToRowSolution (long *rc, Stoch *stoch,
         long *nmod, double **rsoln, long *nrow);

  INPUT:
    Stoch *stoch     pointer to Stoch_Model or Stoch_Data struct
    long *nmod       pointer to the number of the model

  OUTPUT:
    double **rsoln   pointer to the solutions region
    long *rc         pointer to return code (2 is fatal)
    long *nrow       pointer to the number of rows

 RETURNS:
     void
```

ekks_GetPointerToSolution

```
  Gets a pointer to the solution region for a stochastic model.

  FUNCTION PROTOTYPE:
    void ekks_GetPointerToSolution (long *rc, Stoch *stoch, long *nmod,
         double **dsoln);

  INPUT:
    Stoch *stoch     pointer to Stoch_Model or Stoch_Data struct
    long *nmod       pointer to the number of the model for which
                       the solution pointer is desired

  OUTPUT:
    long *rc         pointer to return code (2 is fatal)
    double **dsoln   pointer to the solutions region

  RETURNS:
    void
```

ekks_GetScenarioFromTree

```
  Finds scenario information for sample number nsmp.
```

```
Note: if branching stage, "istg", has value 1, this means
    that the scenario branches from the root (ianc=0).

FUNCTION PROTOTYPE:
  void ekks_GetScenarioFromTree (long *rc, Tree *tree, long nsmp,
       long *ianc, long *istg, double *wgt, long *nscn, long **arr);

INPUT:
  Tree *tree     pointer to Tree struct initialized by
                    ekks_InitializeTree
  long nsmp      sample number (first sample is 0, etc)

OUTPUT:
  long *rc       pointer to return code:
                    1 if maximum number of samples exceeded
                    2 if tree not initialized or corrupted
  long *ianc     pointer to ancestor scenario number
  long *istg     pointer to branching stage number
  double *wgt    pointer to scenario weight
  long *nscn     pointer to scenario number
  long **arr     pointer to data array for scenario

RETURNS:
  void
```

ekks_GetScenarioTree

```
Creates a user accessible stochastic struct

The user needs to copy over the information to
preserve it beyond the next Stochastic Solution call.

The input can be either a Stoch_Data struct or a
Stoch_Model structure, in which case information for
only those nodes in the model is compiled.

FUNCTION PROTOTYPE:
  void ekks_GetScenarioTree (long *rc, Stoch *stoch,
       usrStoch **pstree );

INPUT:
  Stoch *stoch         pointer to a Stoch_Data or Stoch_Model struct

OUTPUT:
  long *rc             pointer to return code (2 is fatal)
  usrStoch **pstree    pointer to a usrStoch structure
                       Note: arrays pointed to by pstree are volatile!

RETURNS:
  void
```

ekks_GetSolution

```
Gets a solution for a scenario.

FUNCTION PROTOTYPE:
  void ekks_GetSolution (long *rc, Stoch *stoch, long s,
       long colrowtype, double *solution, long *indx);

INPUT:
  Stoch *stoch        pointer to Stoch_Model or Stoch_Data struct
  long s              scenario number for which solution
                        info is desired
  long colrowtype     type of solution values desired
                        (1 => rows, else columns)

OUTPUT:
  long *rc            pointer to return code (2 is fatal)
  double *solution    pointer to solution data
  long *indx          pointer to indices

RETURNS:
  void
```

ekks_GetStochasticData

```
Copies Stoch_Data structures from SPL file.

FUNCTION PROTOTYPE:
  void ekks_GetStochasticData (long *rc, Stoch_Data *stoch,
       const char *filename );

INPUT:
  Stoch_Data *stoch     pointer to Stoch_Data struct initialized by
                           ekks_InitializeApplication
  char *filename        name of SPL file

OUTPUT:
  long *rc              return code (2 is fatal)

RETURNS:
  void
```

ekks_Initialize

```
Initializes Stochastic data structures

FUNCTION PROTOTYPE:
  int ekks_Initialize();
```

```
RETURNS:
  integer return code
```

ekks_InitializeApplication

```
Allocates pointers for stochastic data structures,
opens SPL file, returns Stoch_Data pointer

FUNCTION PROTOTYPE:
  void ekks_InitializeApplication (long *rc, Stoch_Data **stoch,
       long maxscn );

INPUT:
  long maxscn                maximum number of scenarios

OUTPUT:
  long *rc                   pointer to return code (2 is fatal)
  Stoch_Data **stochdata     pointer to Stoch_Data struct

RETURNS:
  void
```

ekks_InitializeTree

```
Creates tree and initializes structures.
Creates first scenario from data values in arr[].

FUNCTION PROTOTYPE:
  void ekks_InitializeTree (long *rc, Tree **tree, long *arr,
       long narr, long *mxsmp, double dp );

INPUT:
  long *arr            pointer to an array of integer values
  long narr            number of data values (stages) in arr[]
  long mxsmp           maximum number of samples to be stored
  long dp              probability of events

OUTPUT:
  long *rc             pointer to return code 0
  Tree *tree           pointer to tree structure

RETURNS:
  void
```

ekkse_NestedLSolve

```
Applies Nested decomposition to the stochastic model smodel.

METHODOLOGY:
  Subproblems are created according to the assignment of nodes
  to subproblems. The logic is as follows:
     if minNumRows > 0, nodes are accumulated into subproblems
             until minNumRows is exceeded
     else if ipcut > 0, subproblems are generated by cutting
             at period ipcut
     else if maxmod > 0, use minNumRows = 0.9 * totalrows/maxmod
             and distribute any excess evenly among subproblems

  The program performs MAXCYCLES iterations of Nested
  decomposition, terminating if optimality, infeasibility,
  or unboundedness is detected.

FUNCTION PROTOTYPE:
  void ekkse_NestedLSolve (long *rc, Stoch_Model *smodel,
       long minNumRows, long ipcut, long maxmod, long maxcycles,
       double rmaxmin, long inwmt, long icrsh, long iscal,
       long isslvalg, long isslvinit, long largebounds,
       long *status, double *optgap );

INPUT:
  Stoch_Model *smodel  stochastic model created by OSL
  long minNumRows      min number of rows for size of subproblems
  long ipcut           period at which Nested cuts are made
  long maxmod          maximum number of subproblems
  long maxcycles       maximum number of Master iterations
  double rmaxmin       1.0 => maximize, -1.0 => minimize
  long inwmt           EKKNWMT type (0 => none EKKNWMT)
                         for all problems
  long iscal           1=>do, 0=>don't perform scaling for
                         each submodel
  long icrsh           crash type (0 => none) for first
                         first solution of submodels
  long ilpdc           1=>call, 0=>don't perform LP decomposition
                         for each submodel
  long iprsl           1=>call, 0=>don't perform presolve for
                         each submodel
  long iprsltype       type of presolve to be done
  long iprslunit       presolve unit
  long isalg           EKKSSLV algorithm for first solution
                         of submodels
  long largebounds     1=>add, 0=>don't add large bounds to master
  long *optgap         pointer to Target optimality gap

OUTPUT:
  long *rc             pointer to return code (code >1 => fatal error)
  long *status         pointer to status; value =  0=>optimal,
                         1=>infeasible, 2=>unbounded, 3=>unfinished
  long *optgap         pointer to relative accuracy of attained
                         optimality gap

RETURNS:
  void
```

ekks_PassModelToOSL

```
Passes matrix, costs and bounds data to the Optimization Library
model corresponding to smodel.

FUNCTION PROTOTYPE:
  void ekks_PassModelToOSL (long *rc, Stoch_Model *smodel );

INPUT:
  Stoch_Model *smodel      stochastic model declared by OSL

OUTPUT:
  long *rc                 pointer to return code (2 is fatal)

RETURNS:
  void
```

ekks_ProblemType

```
Sets type of core model. Valid arguments are: linear, quadratic.

FUNCTION PROTOTYPE:
  void ekks_ProblemType ( long *rc, Stoch_Data*stoch,
       CoreType type );

INPUT:
  CoreType type             type of core model
  Stoch_Data *stochdata     pointer to Stoch_Data struct

OUTPUT:
  long *rc                   pointer to return code (2 is fatal)

RETURNS:
  void
```

ekks_PutStochasticData

```
Puts the Stoch_Data struct into a named SPL file.

Opens the file as a buffered direct access file, writes
the data and then closes it.

File may be read by a call to ekks_GetStochasticData.

FUNCTION PROTOTYPE:
  void ekks_PutStochasticData (long *rc, Stoch *stoch,
       char *filename);
```

```
INPUT:
  Stoch_Data *sdata     stochastic data pointer initialized by
                           ekks_InitializeApplication
  char *filename        name of SPL file

OUTPUT:
  long *rc              pointer to return code (2 is fatal)

RETURNS:
  void
```

ekks_ReadMPS

```
Processes the Stochastic MPS files .core, .time
and .stoch, creates scenario data and stores it in
the StochData structure.

The value of replace is set to 0 if STOCH file contains
'ADD' at position 40 in the first line.

FUNCTION PROTOTYPE:
  void ekks_ReadMPS (long *rc, Stoch_Data *stoch, long*smpstype,
       long *replace, char *corefile, char *timefile,
       char *stochfile );

INPUT:
  Stoch_Data *sdata     pointer to Stoch_Data structure to be
                          filled in by ekks_ReadMPS
  char *corefile        pointer to the core MPS filename
  char *timefile        pointer to the time MPS filename
  char *stochfile       pointer to the stoch MPS filename

OUTPUT:
  long *rc              pointer to return code (2 is fatal)
  long *smpstype        integer identifying type of
                          SMPS distribution
  long *replace         0=>add to, 1=>replace core values

RETURNS:
  void
```

ekkse_RegisterCallBack

```
Registers by name a function to be called in place of an
Optimization Library user exit subroutine, identified by number.
The correspondence between values of the numerical identifier
and the names of the user exit subroutines is as follows.
   1 corresponds to MIP User Exit EKKBRNU
   2 corresponds to MIP User Exit EKKCHNU
   3 corresponds to MIP User Exit EKKCUTU
```

```
    4 corresponds to MIP User Exit EKKEVNU
    5 corresponds to MIP User Exit EKKHEUU
    6 corresponds to Iteration User Exit EKKITRU
    7 corresponds to Message User Exit EKKMSGU
    8 corresponds to MIP User Exit EKKNODU
    9 corresponds to Row Ordering User Exit EKKORDU
   10 corresponds to MIP User Exit EKKSLVU

FUNCTION PROTOTYPE:
  void ekkse_RegisterCallBack (int *cbid, void *ekkcbptr);

INPUT:
  int *cbid            pointer to user exit function id number
  void *ekkcbptr       name of call back function being registered

OUTPUT:
  none

RETURNS:
  void
```

ekkse_SetMessage

```
Turn Stochastic Extensions messages on/off.
Message msgid (use symbolic names from message table) is
turned off, if onoff is 0, and turned on, if onoff is 1.

FUNCTION PROTOTYPE:
  void ekkse_SetMessage (long *rc, long msgid, long onoff);

INPUT:
  long msgid             Message identifier (from message table)
  long onoff             0 turns off printing, 1 turns on printing

OUTPUT:
  long *rc               pointer to return code (2 is fatal)

RETURNS:
  void
```

ekkse_SolveParametricModel

```
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 the value of lambdadelta up
to the maximum value, lambdalim.

It is more efficient to set up the problem so that the original is
solved with ekkse_NestedLSolve, then call
```

```
ekkse_SolveParametricModel with lambda=0.

Each time the incremented problem is solved, the function pointed
to by report() is called with the OSL work space attached to smodel
as argument.

The mask argument is used in the same way as in
ekkse_DefineParametricModel

FUNCTION PROTOTYPE:
  void ekkse_SolveParametricModel (long *rc, Stoch_Model *smodel,
       double lambda, double lambdadelta, double lambdalim,
       long mask, long (*report) (double *) );

INPUT:
  Stoch_Model *smodel          pointer to Stoch_Model
  double lambda                initial perturbation to be added
  double lambdadelta           incremental perturbation to be added
  double lambdalim             maximum perturbation to be added
  long mask                    see ekkse_DefineParametricModel
  long (*report) (double *))   called after solution

OUTPUT:
  long *rc                     pointer to return code (2 is fatal)

RETURNS:
  void
```

Messages

EKK0402	*Informational.*	Entering Stochastic Solutions function M01.
EKK0403	*Informational.*	Core model has I01 rows, I02 columns, and I03 stages.
EKK0404	*Informational.*	Scenario I01 Ancestor I02 Stage I03 Probability D01.
EKK0405	*Informational.*	Declaring stochastic model in OSL. Number of rows I01, number of columns I02.
EKK0406	*Informational.*	Total number of blocks declared to OSL is I01.
EKK0407	*Informational.*	Total number of C01 is I01.
EKK0408	*Informational.*	Constructing stochastic model from original node list.
EKK0409	*Informational.*	Process time for C01 is D01 seconds.
EKK0410	*Informational.*	Beginning C01.
EKK0411	*Informational.*	C01 C02 I01 ...
EKK0412	*Informational.*	C01 C02 ...
EKK0413	*Informational.*	C01 D01 ...
EKK0414	*Informational.*	Beginning first pass of Benders decomposition.
EKK0415	*Informational.*	Subproblem I01 did not solve. Exiting.

EKK0416	*Informational.*	Master infeasible. Terminating.
EKK0417	*Informational.*	Master unbounded. Generating ray.
EKK0418	*Informational.*	Estimated subproblem value at new master proposal is E0126.
EKK0419	*Informational.*	Beginning cycle I01.
EKK0420	*Informational.*	Solving subproblem I01.
EKK0421	*Informational.*	Unboundedness detected in subproblem I01 in cycle I02.
EKK0422	*Informational.*	Subproblem actual value at master proposal E0126.
EKK0423	*Informational.*	Problem is unbounded. Terminating.
EKK0424	*Informational.*	Optimality gap for current master proposal is E0114.
EKK0425	*Informational.*	Relative optimality gap is E0114.
EKK0426	*Informational.*	Optimal solution found to within relative optimality gap E0114.
EKK0427	*Informational.*	No new cuts from subproblems. Terminating.
EKK0428	*Informational.*	Maximum cycles exceeded. Terminating.
EKK0429	*Informational.*	No quadratic terms found in model. Resetting type to linear.
EKK0430	*Informational.*	Master problem value is E0126.
EKK3401	*Warning.*	Scenario I01 has already been defined. Probability is changed to D01.
EKK3402	*Warning.*	Scenarios that branch from the root have branching stage = 2.
EKK3403	*Warning.*	Scenario I01 branches prior to ancestor's branch stage: true ancestor is I02.
EKK3404	*Warning.*	Data for current model-block contains elements for earlier stages.
EKK3405	*Warning.*	SPL file is already opened for this application. No action taken.
EKK3406	*Warning.*	No filename supplied. Opening temporary SPL file in /tmp.
EKK3407	*Warning.*	Core C01 value at index I01 is infinite. Nonzero value in scenario I02 ignored.
EKK3408	*Warning.*	Node list is non-empty. Ignored.
EKK3409	*Warning.*	I01 zero probability nodes have been removed from model node list.
EKK3410	*Warning.*	No ENDATA card in C01 file. Ignored.
EKK3416	*Warning.*	Warning! No scenario C01.
EKK3417	*Warning.*	Warning! No stage C01.
EKK3418	*Warning.*	Warning! No column C01.
EKK3419	*Warning.*	Warning! No row C01.
EKK3420	*Warning.*	Warning! Ignored LB of Col I01 Scn I02.
EKK3421	*Warning.*	Warning! Ignored UB of Col I01 Scn I02.
EKK3422	*Warning.*	Warning! Ignored UB of Row I01 Scn I02.
EKK3423	*Warning.*	Warning! End of File.
EKK7401	*Error.*	Malloc failed in subroutine M01. Check maximum memory allocation for user.

EKK7402	*Error.*	Zero allocation attempted in subroutine M01. Check subroutine arguments.
EKK7403	*Error.*	C01 structures not initialized or corrupted. This was detected in subroutine M01.
EKK7404	*Error.*	Sort of C01 failed in subroutine M01.
EKK7405	*Error.*	Error detected while trying to open SPL file C01 in subroutine M01.
EKK7406	*Error.*	Error detected while trying to close SPL file C01 in subroutine M01.
EKK7407	*Error.*	Attempt to reposition within SPL file C01 failed in subroutine M01.
EKK7408	*Error.*	Error detected in subroutine M01 while writing to SPL file C01.
EKK7409	*Error.*	Error detected in subroutine M01 while reading from SPL file C01.
EKK7410	*Error.*	Invalid mode specified in subroutine M01 for opening SPL file C01.
EKK7411	*Error.*	Invalid mode specified in subroutine M01 for closing SPL file C01.
EKK7412	*Error.*	Unable to allocate file pointer for SPL file C01 in subroutine M01.
EKK7413	*Error.*	Unable to allocate filename storage for SPL file C01 in subroutine M01.
EKK7414	*Error.*	Invalid SPL file pointer specified in subroutine M01.
EKK7415	*Error.*	Error detected in subroutine M01 while attaching SPL file C01.
EKK7416	*Error.*	Error detected in subroutine M01 while detaching SPL file C01.
EKK7417	*Error.*	Invalid virtual record specified in subroutine M01 for SPL file C01.
EKK7418	*Error.*	Negative virtual record size specified in subroutine M01 for SPL file C01.
EKK7419	*Error.*	Empty string specified for SPL filename in subroutine M01.
EKK7420	*Error.*	Existing SPL file C01 opened in subroutine M01 is improperly initialized.
EKK7421	*Error.*	Error detected in subroutine M01 while deleting SPL file C01.
EKK7422	*Error.*	File-size discrepancy for SPL file C01 detected in subroutine M01.
EKK7423	*Error.*	Unknown error code for SPL file access used in subroutine M01.
EKK7424	*Error.*	Unable to initialize C01 for SPL file C02 in subroutine M01.
EKK7425	*Error.*	Error detected while attempting to save SPL file with name C01.
EKK7426	*Error.*	Number of C01 requested, I01 exceeds declared maximum I02.
EKK7427	*Error.*	Branching stage out of bounds.
EKK7428	*Error.*	Scenario has negative probability.
EKK7429	*Error.*	Ancestor scenario I01 does not exist.
EKK7430	*Error.*	Number of C01 in CORE model I01 does not match number I02 in scenario.
EKK7431	*Error.*	Data for current model-block is not block lower-triangular.
EKK7432	*Error.*	Total probability in model is zero.
EKK7433	*Error.*	Node list is empty. Cannot construct model.
EKK7434	*Error.*	Error creating hash tables for C01 names.
EKK7435	*Error.*	Not enough space in dspace for C01 file processing.

EKK7436	*Error.*	Index error in C01 file.
EKK7437	*Error.*	Error reading C01 file.
EKK7438	*Error.*	Distribution type not implemented.
EKK7439	*Error.*	Premature EOF reached in C01 file.
EKK7440	*Error.*	Expecting C01 card in C02 file.
EKK7441	*Error.*	Too many elements in scenario I01. Consider revising CORE file, so that more of these elements appear there.
EKK7442	*Error.*	Ranges not implemented in SMPS format.
EKK7443	*Error.*	OSL not initialized or corrupted. This was detected in subroutine M01.
EKK7444	*Error.*	Not enough space in OSL for matrix. Need at least I01 more doublewords.
EKK7445	*Error.*	Duplicate root node discovered in model node tree.
EKK7446	*Error.*	Master status is confused. Exiting.
EKK7447	*Error.*	Bad return code from OSL solving subproblem I01 in cycle I02.

History and Acknowledgements

The precursor of the IBM Stochastic Extensions is the *Stochastic Programming Interface to the Optimization Subroutine Library*, also known as SP/OSL, which was an experimental software library developed over the period 1990-1995 by Alan King and colleagues at the IBM 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 during a summer visit in 1994. The elegant SP-file object and the flexible and robust implementation of the parallel decomposition were entirely designed and written by him.

Robert Entriken (now a consultant with PG&E) wrote the first version of SP/OSL, 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 tree-generation software during his stint as a summer student in 1992. Shabbir Ahmed (now at the University of Illinois, Urbana Champaign) prepared the section on examples in this documentation. Special acknowledgement also is due to Roger Wets, for inspiration, guidance and support during his frequent visits throughout the development period.

Two significant corporate consulting engagements played a major role in the development of SP/OSL.

In 1990, the Frank Russell Corporation asked IBM Research to assess the solvability of certain stochastic programs generated by an asset/liability management system. In response, Robert Entriken quickly built a matrix generator capable of translating SMPS format to OSL problem structures, and Alan King and Robert Entriken working together solved the problem during several overnight runs on an IBM mainframe computer.

In 1992, the Allstate Corporation's Palo Alto Research Center asked IBM to assist them in the development of a multiperiod asset allocation pilot for their property and casualty portfolio. This relationship spurred major development of SP/OSL both as a model generator and as a decomposition solver.

In the period 1993 to 1997, SP/OSL was distributed under special agreement with principal investigators at selected universities and corporate research centers for field testing.

[**Top of Page | Previous (Users Guide) | Next (Examples) | Roadmap**]

Example Problems and Code

- Sample problems
  KandW3 PROD_MIX LandS HYDRO CAP
- Guide to the Driver Source Codes - Illustrating:
  Tree Construction Subroutines
  SMPS Input
  Direct Solution by exsolv
  Post Solution Processing
- Sample Driver Coding
  exsmps.c exsgbds.c exsolv.c exnest.c exsdst.c exsgbdt.c
- Bottom of Page

A collection of sample problems

We describe a collection of small multistage stochastic programming test problems that are available with the IBM Stochastic Extensions. For each problem, both the formulation and test data are described. The SMPS input files for the problems are available in the Stochastic Extensions product directory .\oslstoch\samples\sampc\

- Problem KandW3 Statement Formulation Data Solution
- Problem PROD_MIX Statement Formulation Data Solution
- Problem LandS Statement Formulation Data Solution
- Problem HYDRO Statement Formulation Data Solution
- Problem CAP Statement Formulation Data Solution

Problem KandW3

Statement Formulation Data Solution

This problem has been modified from the production planning example in the text book by Kall and Wallace.

Problem statement

A refinery can blend *N* raw materials into *M* different products. At present, the management is trying to decide how much of each of the raw materials to purchase and stock, so that they can be blended to satisfy the demand for the products in future *T* time periods. The demand has to be completely satisfied, and in case of raw material shortage the products can be outsourced at a higher cost. There is an inventory constraint on how much raw material can be stocked in total.

Formulation

Indices:

i	for raw materials (i = 1,...,N).
j	for products (j = 1,...,M).
t	for time periods (t = 1,...,T).

Variables:

x_{it}	Quantity of raw material i purchased for use in period t .
y_{jt}	Quantity of product j outsourced in period t .

Parameters:

c_i	Present cost of raw material i .
a_{ij}	Raw material i required per unit of product j .
f_{jt}	Cost of outsourcing product j in time period t .
b	Inventory capacity.
$\tilde{d}^k_{jt}$	Random demand of product j in time period t .

KandW3 :

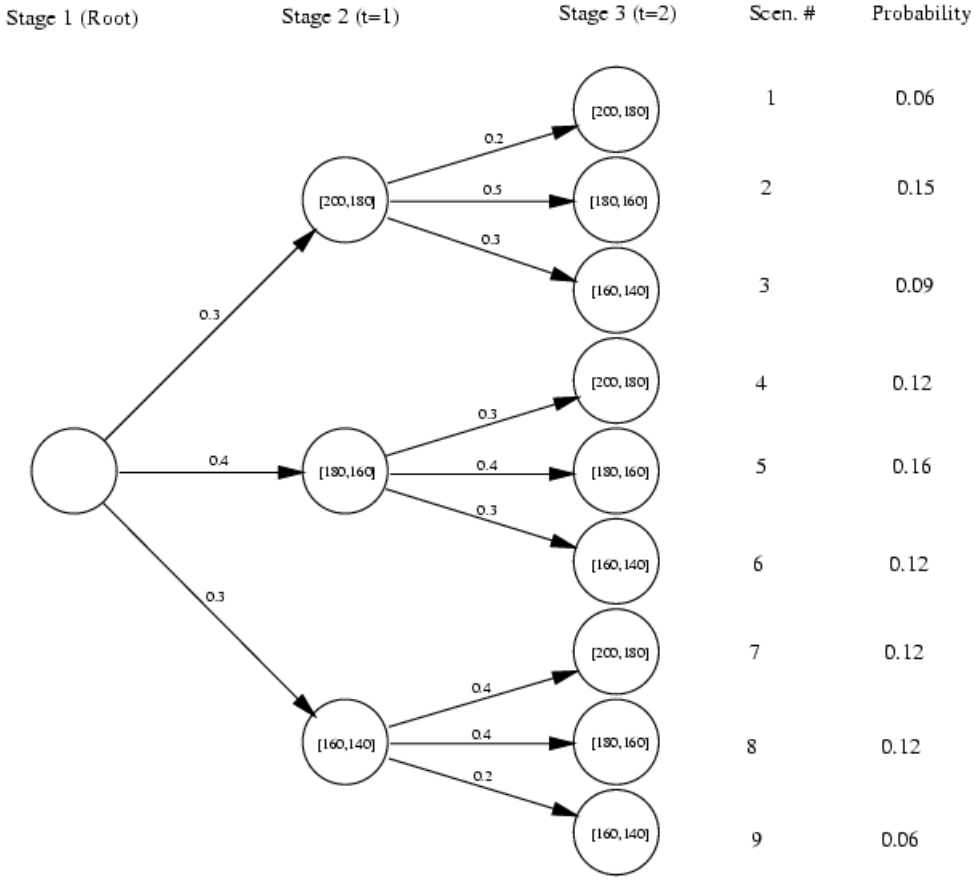
$$
\begin{array}{ll}
\min & \sum_{i,t} c_i x_{it} + \mathrm{E}[\sum_{j,t} f_{jt} y_{jt}] \\
\text{s.t.} & \sum_{i,t} x_{it} \leq b \\
& \sum_i a_{ij} x_{it} + y_{jt} \geq \tilde{d}_{jt} \qquad \forall j,t \\
& x_{it}, y_{jt} \geq 0 \qquad \forall i,j,t
\end{array}
$$

Test problem data

$$
\begin{array}{lcl}
N & = & 2 \\
M & = & 2 \\
T & = & 2 \\
c & = & [2.0, 3.0] \\
a & = & \begin{bmatrix} 2.0 & 6.0 \\ 3.0 & 3.4 \end{bmatrix} \\
f & = & \begin{bmatrix} 7.0 & 10.0 \\ 12.0 & 15.0 \end{bmatrix} \\
b & = & 50
\end{array}
$$

The scenario tree for the demand realizations is shown in Scenario tree for problem KandW3.

Figure 3.1: Scenario tree for problem KandW3



Solution

$$
\begin{array}{rcl}
\text{Objective value} & = & 2613.0 \\
x_{11}^* & = & 0.0 \\
x_{21}^* & = & 20.0 \\
x_{12}^* & = & 0.0 \\
x_{22}^* & = & 30.0
\end{array}
$$

Problem PROD_MIX

Statement Formulation Data Solution

This problem is taken from Examples from the Literature.

Problem statement

The problem consists of determining the product mix for a furniture shop with two workstations: carpentry and finishing. The availability of labor in man-hours at the two stations is limited. There are four product classes, each consuming a certain number of man-hours at the two stations. Each product earns a certain profit and the shop has the option to purchase labor from outside. The objective is to maximize the profit.

Formulation

Indices:

i	for product class (i = 1,...4).
j	for workstation (j=1,2).

Variables:

x_i	Amount of product class j produced.
v_j	Man-hours purchased from outside for workstation j .

Parameters:

c_i	Profit per unit of product class i .
q_j	Cost per man-hour for labor for workstation j .
$\tilde{t}_{ji}$	Random man-hours at work station j required per unit of product class i .
$\tilde{h}_j$	Random man-hours available at workstation j .

PROD_MIX:

$$\begin{array}{lll} \max & \sum_i c_i x_i - \mathrm{E}[\sum_j q_j v_j] & \\ \text{s.t.} & \sum_i \tilde{t}_{ji} x_i \le \tilde{h}_j + v_j & \forall j \\ & x_i, v_j \ge 0 & \forall i, j \end{array}$$

Note that the problem is in the form of a minimization problem in the data files.

Test problem data

$$
\begin{array}{rcl}
c & = & [12.0, 20.0, 18.0, 40.0] \\
q & = & [5.0, 10.0] \\
\tilde{h} & = & [\mathcal{N}(6000, 100), \mathcal{N}(4000, 50)] \\
\tilde{t} & = & \begin{bmatrix} \mathcal{U}(3.5, 4.5) & \mathcal{U}(8.0, 10.0) & \mathcal{U}(6.0, 8.0) & \mathcal{U}(9.0, 11.0) \\ \mathcal{U}(0.8, 1.2) & \mathcal{U}(0.8, 1.2) & \mathcal{U}(2.5, 3.5) & \mathcal{U}(36.0, 44.0) \end{bmatrix}
\end{array}
$$

Above $\mathcal{N}(a,b)$ denotes a normal distribution with mean *a* and standard deviation *b* , and denotes a uniform distribution in the range [*a*,*b*] .

Solution

The solution of the above problem for a sample of 300 scenarios is as follows:

$$
\begin{array}{rcr}
\text{Objective value} & = & -17730.3 \\
x_1^* & = & 1381.86 \\
x_2^* & = & 0.0 \\
x_3^* & = & 0.0 \\
x_4^* & = & 55.92
\end{array}
$$

Note that the objective value is negative since the problem was supplied as a minimization problem.

Problem LandS

Statement Formulation Data Solution

The following problem is taken from Louveaux and Smeers.

Problem statement

The following two-stage problem consists of determining the optimal capacity investment in various types of power plants so as to meet next period demands for electricity. Four power plants are considered and they can operate in three different modes. The next period demand for each of the three modes are to be met. There is a budget constraint and also a constraint on the minimum total capacity.

Formulation

Indices:

i	for power plant type (i = 1,...,4).
j	for operating mode (j = 1,...,3).

Variables:

x_i	Installed capacity for power plant of type i .
y_{ij}	Operating level of power plant i in mode j .

Parameters:

c_i	Cost per unit of capacity installed for plant type i .
f_{ij}	Cost per unit operating level of power plant i in mode j .
m	The minimum total capacity to be installed.
b	Budget available for capacity installment.
$\tilde{d}_j$	Random demand for power in mode j .

LandS:

$$
\begin{aligned}
\min \quad & \sum_i c_i x_i + E \sum_{i,j} f_{ij} y_{ij} \\
\text{s.t.} \quad & \sum_i x_i \geq m \\
& \sum_i c_i x_i \leq b \\
& \sum_j y_{ij} \leq x_i \quad \forall i \\
& \sum_i y_{ij} \geq \tilde{d}_j \quad \forall j \\
& x_i, y_{ij} \geq 0 \quad \forall i, j
\end{aligned}
$$

Test problem data

$$
\begin{aligned}
c &= [10.0, 7.0, 16.0, 6.0] \\
f &= \begin{bmatrix} 40.0 & 24.0 & 4.0 \\ 45.0 & 27.0 & 4.5 \\ 32.0 & 19.2 & 3.2 \\ 55.0 & 33.0 & 5.5 \end{bmatrix} \\
m &= 12.0 \\
b &= 120.0 \\
\tilde{d} &= [\xi, 3.0, 2.0]
\end{aligned}
$$

where ξ is a random variable taking the values 3.0,5.0 or 7.0 with probability 0.3, 0.4 and 0.3 respectively.

Solution

$$
\begin{aligned}
\text{Objective value} &= 381.85 \\
x_1^* &= 2.67 \\
x_2^* &= 4.00 \\
x_3^* &= 3.33 \\
x_4^* &= 2.00
\end{aligned}
$$

Problem HYDRO

Statement Formulation Data Solution

The following problem is taken from Pereira and Pinto.

Problem statement

The objective of the problem is to determine a generation schedule for a hydrothermal generating system such that expected operating costs are minimized. These costs are composed of fuel costs for the thermal units and penalties for failure to meet power demand. The hydrothermal system consists of four hydroelectric reservoirs and a thermal generation unit. Constraints to the problem are the mass balance equations for the water supply across the reservoirs and across the stages; the demand constraints and the capacity of the operating units. The schedule has to be developed for three time periods.

Formulation

Indices:

i	for the reservoirs in the system (i = 1,...4).
i'	an alias for i .
t	for the time periods (t = 1,...3).

Variable:

GT_t	Thermal generation in time period t .
D_t	Unmet power demand in time period t .
V_{it}	Water volume at reservoir i at the end of period t .
Q_{it}	Water volume passed through reservoir i turbines in period t .
S_{it}	Water volume spilled from reservoir i in period t .

Parameters:

M_i	The set of reservoirs immediately upstream of reservoir i .
c_t	Cost of thermal generation in period t .
f_t	Penalty for unmet demand in period t .
L_t	Power demand in period t .
ρ_i	Power generated per unit volume turbined at reservoir i .
V_{i0}	Initial volume in reservoir i .
$\overline{V}_i$	Capacity of reservoir i .
$\overline{Q}_i$	Turbine capacity at reservoir i .
$\tilde{A}_{it}$	Random water inflow to reservoir i in period t .

HYDRO:

Test problem data

The reservoir configuration is shown in Reservoir configuration for problem HYDRO.

The scenario tree for the water inflow vectors is given in Scenario tree for problem HYDRO.

Figure 3.2: Reservoir configuration for problem HYDRO

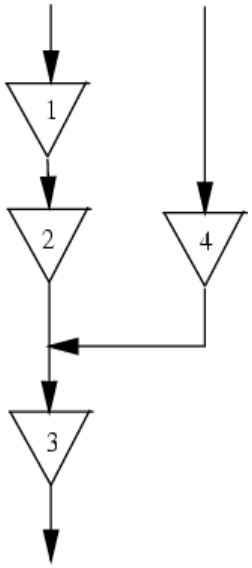
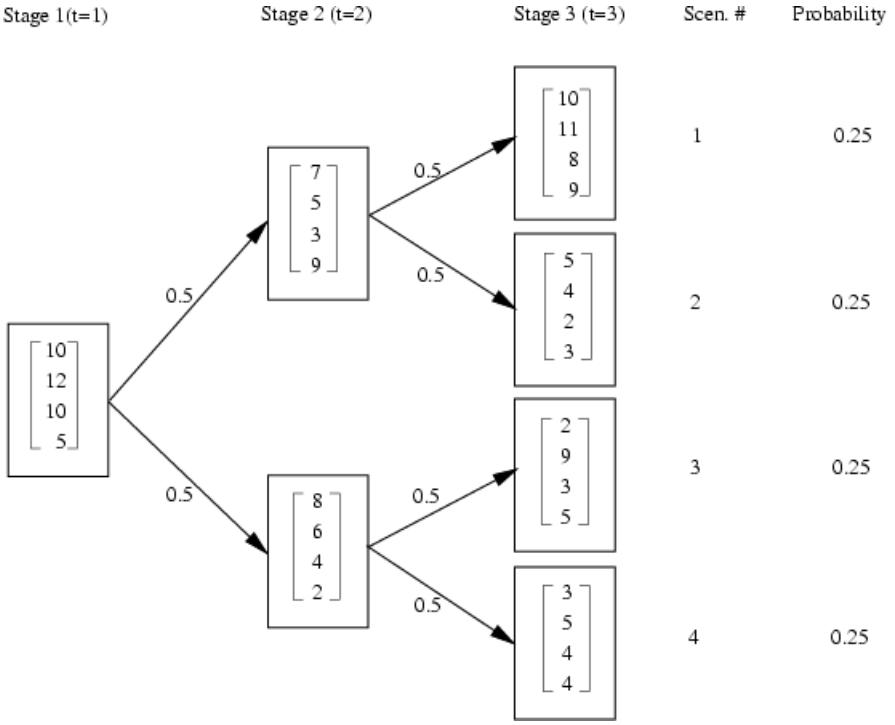


Figure 3.3: Scenario tree for problem HYDRO



Solution

The first stage solutions are reported below. Note that these solutions do not conform to those reported in the original paper Pereira and Pinto.

$$\begin{aligned}\text{Objective value} &= 265.3\\ GT_1^* &= 28.0\\ D_1^* &= 0.0\\ V_1^* &= [0.0, 112.0, 15.0, 0.0]\\ Q_1^* &= [60.0, 0.0, 100.0, 55.0]\\ S_1^* &= [0.0, 0.0, 0.0, 0.0]\end{aligned}$$

Problem CAP

Statement Formulation Data Solution

Problem statement

The following problem consists of designing the capacities of the plants in a chemical processing network in order to maximize expected future profits. The network contains three processes and two chemical products, and a planning horizon of two periods is considered. The profit is the difference between the sales revenue of the chemicals and the capacity installation and operating costs for the plants.

formulation

Indices:

i	for processes (i = 1,...,3).
j	for chemicals (j=1,2).
t	for time periods (t=1,2).
τ	an alias for t .

Variables:

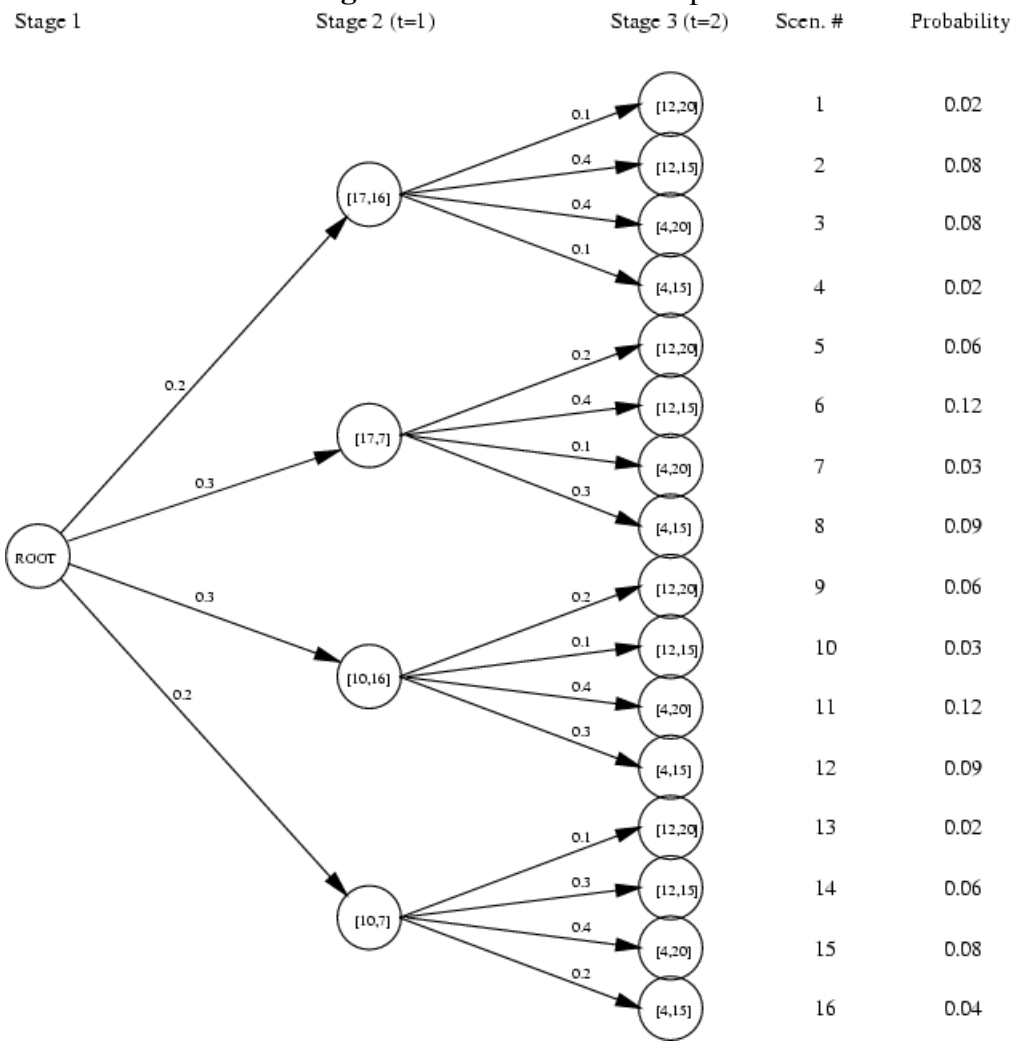
x_{i0}	Design capacity of process i .
x_{it}	Capacity installed for process i in time period t .
w_{it}	Operating level of process i in period t .

Parameters:

c_{i0}	Cost per unit of design capacity for process i .
c_{it}	Cost per unit of capacity installed in process i in period t .
q_{it}	Operating cost of process i in period t .
r_{jt}	Sales price for chemical j in period t .
a_{ij}	Production of chemical j per unit operating level of process i .
$\tilde{d}^L_{jt}, \tilde{d}^U_{jt}$	Demand upper and lower bounds for chemical j in period t .

CAP:

Figure 3.4: Scenario tree for problem CAP



Test problem data

The demand lower bounds are assumed to be zero. The scenario tree for the demand upper bounds is given in Scenario tree for problem CAP.

Solution

Objective value $= -272.83$

$x_0 = [7.083, 0.0, 0.0]$

Note that the objective value is negative since the problem was recast into the minimization form.

Shabbir Ahmed:

Operations Research Laboratory, Department of Mechanical and Industrial Engineering, University of Illinois, Urbana-Champaign.

Guide to the Driver Source Codes

The driver programs in the following sections illustrate the use of some of the options for input, solution and data access available in the Stochastic Extensions modules. The C-language coding for the examples may be found in the ./examples directory. This code is reproduced in the Sample Driver Coding section.

All the input examples create an SPL-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:

```
> exsmps data/app0110
> exsolv data/app0110
```

Driver illustrating Tree Construction Subroutines

Driver exsgbdt is an example based on the aircraft allocation example of exsgbds. In this example, the distribution is sampled and the tree built up using ekks_AddToTree and ekks_GetScenarioFromTree subroutines. This is only a one-period example, nevertheless, multiperiod examples can easily be constructed on the basis of this code. The exsgbdt call produces the SP file sgbd_smpl.spl.

Driver Illustrating SMPS Input

The driver exsmps reads the SMPS input format. To use the driver, invoke it as follows:

```
> exsmps 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.

Direct Solution by exsolv

Driver exsolv restores the stochastic program from the SPL-file <file>.spl and solves it using decomposition. For instance,

```
> exsolv data/app0110
```

solves the problem generated by the invocation of exsmps in the example given above.

Post Solution Processing

Driver exsdst restores the stochastic program from the SP-file <file>.spl and solves the problem using decomposition. The optimal value along each scenario is then computed by using the user-access routines ekks_GetCoreData, ekks_GetScenarioTree and ekks_GetNodeData, and the results are displayed as a distribution. For instance,

```
> exsdst data/app0110
```

solves the problem generated by the invocation of smps in the example given above, and produces the distribution of objective values.

Sample Driver Coding

exsmps.c

Reads SMPS formatted files and saves SP file.

```
/*  exsmps.c:  Main driver for reading SMPS files.
 *
 *          Usage:   exsmps <file>
 *
 *          Reads CORE, TIME, and STOCH files from
 *          <file>.core, <file>.time, and <file>.stoch.
 *          Writes the SP-file as <file>.spl.
 *
 *          Implements INDEPENDENT DISCRETE, as well as SCENARIOS
 *          versions of the standard input format.
 *
 *          Demonstrates coding for taking SAMPLES from the independent
 *          distribution.
 *
 *         ( compile with /DOSLMSDLL on INTEL platforms. )
 */

#include <stdlib.h>
#include <string.h>
#include "ekkstoch.h"

#define SAMPLE 0                        /* (0,1) means (don't, do) sample */

#define MXSCN 400

void main( long argc, char *argv[] ) {

    Stoch_Data *s_data=NULL;                    /* Stoch_Data pointer */
    Tree *tree=NULL;
    long type;                                  /* distribution type */
    long replace;                 ;              /* set by ekks_ReadMPS() */
    long rc=0;                                  /* return code */


/* 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\n");
      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\n");
      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" );

/* Initialize OSL environments */
    rc = ekks_InitializeOSL();           if(rc>0) return;

/* initialize Stoch_Data with maximum MXSCN scenarios */
    ekks_InitializeApplication( &rc,&s_data,MXSCN );
        if ( rc>1 ) return;

/* Process MPS file + Process STOCH file + Process TIME file  */
    ekks_ReadMPS( &rc,s_data,&type,&replace, corefile, timefile, stocfile);
    if ( rc>1 ) return;

/* Process discrete distribution (if that's what it is) */
    if ( type==INDEP_DISCRETE || type==BLOCK_DISCRETE ){

        Independent *s_indp = NULL;
        long *indx;

/* ...get pointer to independent distribution structure */
        ekks_GetPointerToDistribution( &rc,s_data,&s_indp );
                if ( rc>1 ) return;

/* ...malloc indexing arrays */
        nsize = (s_indp->nrvar) * sizeof(long);
        indx = (long *) malloc( nsize );        if (!indx) return;

/* ...loop for sampling independent distributions: */
        if (1==SAMPLE) {
                long jj,irand,ev_indx;
                long ns;
     ;           double *dprobs;
                double dp,ds;
                Discrete *pdist;
```

```
                long nindp = s_indp->nrvar;

                ds = 1.0e0;
                ds /= MXSCN;
                for (ns=0;ns < MXSCN;ns++) {

/*   ...for each indep rv, process random number to get index */
                   for (jj=0;jj<nindp;jj++) {
                           ev_indx = 0;
                           pdist = s_indp->rvar[jj]->dist;
                           dprobs = pdist->dprb;
                           dp = dprobs[ev_indx];
                           irand = rand();
                       ;    while ( irand > dp*RAND_MAX )
                                   dp += dprobs[++ev_indx];
                           indx[jj] = ev_indx;
                   }

/*   ...add event */
                   ekks_AddEvent( &rc,s_data,&tree,indx,ds,replace );
                        if ( rc>1 ) return;

                }

        } /* end if(SAMPLE) */

/* ...if NOT SAMPLE then we add all events: */
        if (0==SAMPLE) {
           long jj,ii,iii;
           long ns;
           double dp;
           long *incr;
           Discrete *pdist;

/* ... malloc increments array */
           incr = (long *) malloc( nsize );     if (!incr) return;

/* ...initialize the event increment to 1 and event index to 0 */
           for (jj=0;jj<s_indp->nrvar;jj++) incr[jj] = 1;
           memset(indx,0,nsize);

/* ...compute probability of first event */
           dp = 1.0;
           for (jj=0;jj<s_indp->nrvar;jj++) {
                pdist = s_indp->rvar[jj]->dist;
                dp *= pdist->dprb[ indx[jj] ];
           }

/* ...add first event */
                   ekks_AddEvent( &rc,s_data,&tree,indx,dp,replace );
                if ( rc>1 ) return;

/* ...main loop
        The logic of this loop is as follows:

        - If nevent[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 ekks_AddScenario with the new event.

        This works because the total number of events is
        the product of the number for each independent RV.
*/

           ns = 1;
           for (ii=0;ii<s_indp->nrvar;ii++) {
                pdist = s_indp->rvar[ii]->dist;
                ns *= pdist->nevt;
           }

           for (ii=1;ii<ns;ii++) {
                iii=ii; jj=0;
                pdist = s_indp->rvar[jj]->dist;
                while ( !(iii%pdist->nevt) ) {
                        iii /= pdist->nevt;
                        incr[jj] = -incr[jj];
                        jj++;
                        pdist = s_indp->rvar[jj]->dist;
                }

/* ...  adjust probability */
                dp /= pdist->dprb[ indx[jj] ];
                indx[jj] += incr[jj];
                dp *= pdist->dprb[ indx[jj] ];

/* ...  add event */
                    ekks_AddEvent( &rc,s_data,&tree,indx,dp,replace );
           ;                                        if ( rc>1 ) return;

           } /* end for */
        } /* end if(!SAMPLE) */

    } /* end if ( type==INDEP_DISCRETE || type==BLOCK_DISCRETE ) */

/* Save SP-file */
    ekks_PutStochasticData( &rc,s_data,splfile );

/* Free Stoch_Data structures */
    ekks_FreeStructure(s_data);
}
```

exsgbds.c

Generate aircraft allocation problem using ekks_AddScenario.

```
/*  exsgbds.c:  Generic driver using ekks_AddScenario.
 *
 *         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.
 *         ( compile with /DOSLMSDLL on INTEL platforms. )
 */

#include  <stdlib.h>
#include "ekkstoch.h"

#define INF 1.0e31

void 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;

/* Initialize OSL environments */
   rc = ekks_InitializeOSL();
     if (rc > 0) return;

/* compute number of scenarios */
    for (ii=0;ii<nindp;ii++) ns *= nsamp[ii];
/* initialize Stoch_Data with ns scenarios */
    ekks_InitializeApplication( &rc,&s_data,ns );
    if ( rc>1 ) return;

/* Create Core */
    ekks_CreateCore( &rc,s_data,nstg,rstg,cstg,nrow,ncol,nels,
          dobj,drlo,drup,dclo,dcup,mrow,mcol,dels,0 );

/* 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) );
    memset( indx,0,(1+nindp)*sizeof(long));
    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 */
    ekks_AddScenario( &rc,s_data,1,0,2,dp,1,
          nrow,ncol,nels,dobj,drlo,drup,dclo,dcup,
          mrow,mcol,dels,ADD_TO_CORE_VALUES );

/* ...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 ekks_AddScenario 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] ];
     ekks_AddScenario( &rc,s_data,ii+1,0,2,dp,1,
               nrow,ncol,nels,dobj,drlo,drup,dclo,dcup,
               mrow,mcol,dels,ADD_TO_CORE_VALUES );
    }

/* Save SP-file as "sgbd.spl" */
    ekks_PutStochasticData( &rc,s_data,"sgbd.spl" );

/* Free Stoch_Data structures */
    ekks_FreeStructure(s_data);
}
```

exsolv.c

Read an SP file and solve using decomposition.

```
/*
 * exsolv.c:  Main driver for reading SP-files and solving.
 *
 *         Reads problem from file .spl and solves using
 *         exsolv().
 *         (compile with  /DOSLMSDLL on INTEL platforms.)
 */

#include
#include
#include
#include "ekkstoch.h"

#define MXSCN 750              /* maximum scenarios */

#define TARG_OPT_GAP  1.0e-8   /* default target optimality gap */

#define abs(a) (a>0?a:-(a))

void 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 *solution;                   /* space for solution */
    long *osl_indx;                     /* space for osl indices */
    long rc, nf, ii, nblks, nmod=1;     /* local variables */
    long Stage, Scenario;               /* Stage#, Scenario# */
    long colbeg, colindex;              /* Column start, column index */

    /* filenames */
    char *splfile,*outfile;
    char myfile[80];
    long nchar,nsize;

    /* get application name from command line */
    if ( argc != 2 ) {
         printf(" Usage:  solv   , where "
             "{file.spl} is a valid SPL file.\n");
         return;
    }
    nchar = (long) strlen( argv[1] );
    nsize = (nchar+1)+4;
    splfile = (char *) malloc(sizeof(char)*nsize);
    if ( splfile==NULL ) {
         printf(" ERROR: solv is unable to store input file-name\n");
         return;
    }
    outfile = (char *) malloc(sizeof(char)*nsize);
    if ( outfile==NULL ) {
         printf(" ERROR: solv is unable to store output file-name\n");
         return;
    }
    strcpy( splfile,argv[1] );
```

```
    strcat( splfile,".spl" );
    sprintf(myfile,"%s",splfile);

    strcpy( outfile,argv[1] );
    strcat( outfile,".out" );

/* initialize OSL environment */
    rc = ekks_InitializeOSL();  if (rc>0) return;

/* initialize Stoch_Data with maximum MXSCN scenarios */
    ekks_InitializeApplication( &rc,&s_data,MXSCN );
     if ( rc>1 ) return;

/* Read SP-file */
    sprintf(splfile,"%s",myfile);
    ekks_GetStochasticData( &rc,s_data,splfile );
     if ( rc>1 ) return;

/* Declare stochastic model (full model) */
    ekks_DescribeModel( &rc,s_data,0,0,&s_model,&nblks,1 );
     if ( rc>1 ) return;

/* Solve -- using ekks_BendersLSolve */
    {
        long subproblems=10,iterations=100;
        long ipcut=2, inwmt=0, icrsh=3, ilpdc=0, iscal=0, iprsl=0,
                iprsltype=0, iprslunit=0, isalg=1;
        long large_bounds=0;
        long status;
        double optmaxmin=1.0, optgap=TARG_OPT_GAP;  /*  optmaxmin= 1.0:
minimize
                                                        optmaxmin=-1.0:
maximize */

        /* Solve */

        long bndrrtcod;
        ekks_BendersLSolve( &bndrrtcod, s_model, ipcut,
            subproblems, iterations, optmaxmin,
            inwmt, icrsh, ilpdc,
            iscal,
            iprsl, iprsltype, iprslunit,
            isalg, large_bounds,
            &status,&optgap );
        if ((rc = bndrrtcod) > 1) return;

    }

/* Get core data */
    ekks_GetCoreData(&rc, s_data, &u_core );
    if ( rc>1 ) return;

/* Get stage/scenario solution */
    Scenario = 3;
    Stage = 2;
/* Malloc space for solution and index arrays */
    solution = (double *) malloc( u_core->ncol*sizeof(double) );
    osl_indx = (long *) malloc( u_core->ncol*sizeof(long) );
```

```
    colbeg = u_core->colstart[Stage-1]-1;
    nf = u_core->colstart[Stage] - u_core->colstart[Stage-1];

/* Get the solution */
    ekks_GetSolution( &rc,s_model,Scenario,0,solution,osl_indx);
    if ( rc>1 ) return;

/* Print scenario/stage solution */
    printf("\n\nScenario %d has %d columns in stage %d\n\n",Scenario, nf,
Stage);
    for (ii=0;iisortin2exc[ii+colbeg]-1;
     if ( abs(solution[colindex]) < 1.0e-10 ) solution[colindex] = 0.0;
     printf(" Column[%d] = %g\n",osl_indx[colindex], solution[colindex]);
    }

/* Free data structures */
    ekks_FreeStructure( s_model ); ekks_FreeStructure( s_data );
        return;
}
```

exnest.c

Read an SP file and solve using NestedLSolve.

```
/*  exnest.c:  Another driver for reading SP-files and solving.
 *
 *         Reads problem from file .spl and solves using
 *         ekks_NestedLSolve.
 *         ( compile with  /DOSLMSDLL on INTEL platforms. )
 */

#include
#include
#include
#include "ekkstoch.h"

#define MXSCN 1000              /* maximum scenarios */
#define TARG_OPT_GAP  1.0e-8    /* default target optimality gap */
#define abs(a) ( a>0 ? a : -(a) )

void 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;                 /* OSL workspace         */
    double *solution;                   /* space for solution    */
    long *osl_indx;                     /* space for OSL indices */
    long rc, nf, ii, nblks, nmod=1;     /* local variables       */

   /* filenames */
    char *splfile,*outfile;
    char myfile[80];
    long nchar, nsize;
```

```
   /* get application name from command line */
    if ( argc != 2 ) {
         printf(" Usage:  solv   , where "
             "{file.spl} is a valid SPL file.\n");
         return;
    }
    nchar = (long) strlen( argv[1] );
    nsize = (nchar+1)+4;
    splfile = (char *) malloc(sizeof(char)*nsize);
    if ( splfile == NULL ) {
         printf(" ERROR: solv is unable to store input file name\n");
         return;
    }
    outfile = (char *) malloc(sizeof(char)*nsize);
    if ( outfile == NULL ) {
         printf(" ERROR: solv is unable to store output file name\n");
         return;
    }
    strcpy ( splfile, argv[1] );
    strcat ( splfile, ".spl" );
    sprintf( myfile, "%s", splfile );
    strcpy ( outfile, argv[1] );
    strcat ( outfile, ".out" );
    free   ( splfile );
    free   ( outfile );

/* initialize OSL environment */
    rc = ekks_Initialize();  if ( rc > 0 ) return;

/* initialize Stoch_Data with at most MXSCN scenarios */
    ekks_InitializeApplication( &rc, &s_data, MXSCN );
    if ( rc > 1 ) return;

/* Read SP-file */
    ekks_GetStochasticData( &rc, s_data, myfile );
    if ( rc > 1 ) return;

/* Declare stochastic model ( full model ) */
    ekks_DescribeModel( &rc, s_data, 0, 0, &s_model, &nblks, 1 );
    if ( rc > 1 ) return;

/* Solve using BendersLSolve or NestedLSolve*/
    {
       long subproblems=10, iterations=1, nestedits=30;
       long ipcut=2, inwmt=1, icrsh=3, ilpdc=0, iscal=0,
            iprsl=0, iprsltype=0, iprslunit=0, isalg=1;
       long large_bounds=0;
       long status;
       double optmaxmin=1.0, optgap=TARG_OPT_GAP;
      /* optmaxmin= 1.0 => minimize
       * optmaxmin=-1.0 => maximize */
       long bndrrtcod=0, nestedrtcod=0, minNumRows=0, isslvinit=1;

      /* ekks_BendersLSolve( &bndrrtcod, s_model, ipcut,
       *    subproblems, iterations, optmaxmin,
       *    inwmt, icrsh, ilpdc, iscal,
       *    iprsl, iprsltype, iprslunit,
       *    isalg, large_bounds,
```

```
       *    &status,&optgap );
       * if ( (rc = bndrrtcod) > 1) return; */

       ekkse_NestedLSolve( &nestedrtcod, s_model, minNumRows,
                    ipcut, subproblems, nestedits, optmaxmin,
                    inwmt, icrsh, iscal, isalg, isslvinit,
                    large_bounds, &status, &optgap );
       if ( (rc = nestedrtcod) > 1) return;
    }

/* Get core data */
    ekks_GetCoreData(&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) );
    ekks_GetSolution( &rc, s_model, 1, 0, solution, osl_indx);
    if ( rc > 1 ) return;

/* Print first stage solution */
    nf = u_core->colstart[1] - u_core->colstart[0];
    printf("\n\nNumber of first stage columns %d\n", nf);
    for ( ii = 0; ii < nf; ii++ ) {
      if ( abs(solution[ii]) < 1.0e-10 ) solution[ii] = 0.0;
      printf(" Solution for index %d is %g\n", ii+1, solution[ii] );
    }

/* Free data structures */
    ekks_FreeStructure( s_model ); ekks_FreeStructure( s_data );
    free( solution ); free( osl_indx );

    return;
}
```

exsdst.c

Solve and compute distribution of optimal values.

```
/*  exsdst.c:  Present distribution of optimal values.
 *
 *         Usage:  exsdst <file>
 *
 *         Reads problem from file <file>.spl and solves using
 *         ekks_BenderLsolve.  Then, as an example of user coding, computes
 *         the distribution of the optimal value.
 *         ( compile with /DOSLMSDLL on INTEL platforms. )
 */

#include <stdio.h>
#include <math.h>
#include <string.h>
#include "ekkstoch.h"
```

```
#define MXSCN 750               /* maximum scenarios */

#define TARG_OPT_GAP  1.0e-8   /* default target optimality gap */

void 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 *solution=NULL;               /* space for solution */
    long rc, rtcod=0, nf, ii, nblks, nmod=1;        /* local variables */

     /* filenames */
     char *splfile;
     char myfile[80];
     long nsize,nchar;
     long len;

     /* get application name from command line */
     if ( argc != 2 ) {
          printf(" Usage:  exsdst <file>  , where "
              "{file.spl} is a valid SPL file.\n");
          return;
     }
     nchar = (long) strlen( argv[1] );
     nsize = (nchar+1)+3;
     splfile = (char *) malloc(sizeof(char)*nsize);
     if ( splfile==NULL ) {
          printf(" ERROR: exsdst is unable to store input file-name\n");
          return;
     }
         strcpy( splfile,argv[1] );
     strcat( splfile,".spl" );
     sprintf(myfile,"%s",splfile);


/*  Initialize OSL environments */
    rc = ekks_InitializeOSL();  if (rc > 0) return;

/* initialize Stoch_Data with maximum MXSCN scenarios */
    ekks_InitializeApplication( &rc,&s_data,MXSCN );
     if ( rc>1 ) return;

/* Read SP-file */
    sprintf(splfile,"%s",myfile);
    ekks_GetStochasticData( &rc,s_data,splfile );
     if ( rc>1 ) return;

/* Declare stochastic model (full model) */
    ekks_DescribeModel( &rc,s_data,0,0,&s_model,&nblks,1 );
     if ( rc>1 ) return;

/* Solve -- using ekks_BenderLSolve */
    {
        long subproblems=10,iterations=1;
        long ipcut=2, inwmt=0, icrsh=3, ilpdc=0, iscal=0, iprsl=0,
                iprsltype=0, iprslunit=0, isalg=1;
        long large_bounds=0;
```

```
        long status;
        double optmaxmin=1.0, optgap=TARG_OPT_GAP;
                /* optmaxmin=  1.0: minimize
                            = -1.0: maximize */


        /* Solve */

        long bndrrtcod;
        ekks_BenderLSolve( &bndrrtcod, s_model, ipcut,
            subproblems, iterations, optmaxmin,
            inwmt, icrsh, ilpdc,
            iscal,
            iprsl, iprsltype, iprslunit,
            isalg, large_bounds,
            &status,&optgap );
        if ((rc = bndrrtcod) > 1) return;

    }

/*
 * Example of User computation using scenario tree and OSL
 *
 */

     {
          usrStoch *stree = NULL;
          usrNodeData *ndata = NULL;
          usrCoreData *cdata = NULL;
          usrNode *pnode = NULL;
          double *node_obj = NULL;

          /* 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 */
          ekks_GetCoreData( &rc,s_model,&cdata );   if ( rc>1 ) return;

          /* get stochdata information */
          ekks_GetScenarioTree( &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 */
          dsoln = (double *) malloc ( stree->node->ncol  * sizeof(double));
          if (dsoln == NULL) {
                      printf("\n Not enough memory available");
                          return;
                  }

          ekks_GetPointerToSolution(&rc, s_model, &nmod, &dsoln);
          if (rc > 1) return;
```

```
/*
 * 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 ( ii > 0 ) {
                  /* get node objective */
                  ekks_GetNodeData( &rc, s_model, ii, &ndata );
                  dnobj = ndata->dobj;

                  if(dnobj!=NULL)
             ;           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(" \n \t Distribution of Optimal Values \n \n");

          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(" \t Scenario %d value %f probability %f \n",
                   is,scenvalue,dprob);
```

```
                mean += dprob*scenvalue;
                variance += dprob*scenvalue*scenvalue;

            }

            variance -= mean*mean;
            if (variance < 0.0) variance=0.0;

            printf(" \t \t Mean Value: \t \t %f \n",mean);
            printf(" \t \t Standard Deviation: \t %f \n",sqrt(variance));

    }

/* Free data structures */
    ekks_FreeStructure( s_model ); ekks_FreeStructure( s_data );

}
```

exsgbdt.c

Assemble tree using ekks_*Tree() subroutines.

```
/*  exsgbdt.c:  Generic driver using ekks_*Tree() subroutines.
 *
 *         Input distribution is sampled and processed by
 *         ekks_*Tree() subroutines.
 *
 *         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.
 *         ( compile with /DOSLMSDLL on INTEL platforms. )
 */

#include <stdlib.h>
#include "ekkstoch.h"

#define INF 1.0e31

#define N_SAMPLES  300      /* number of samples to be generated */

void 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 };
        double dqdg[]={ 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,
              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 };

/* Tree structure */
    Tree *tree=NULL;

/* local variables */
    long rc,ns=1,ii,iii,jj,*indx,*incr;
    double dd,dp=1.0,dwgt;
    long irand,indx_beg,ev_indx;
    long nscen,ianc,istg;
    long *label;
    long evoffset;



/*  Initialize OSL environments */
    rc = ekks_InitializeOSL();
    if(rc > 0) return;

/* initialize Stoch_Data with ns scenarios */
    ekks_InitializeApplication( &rc,&s_data,N_SAMPLES );
        if ( rc>1 ) return;

    ekks_ProblemType( &rc,s_data,quadratic);
        if ( rc>1 ) return;

/* Create Core */
    ekks_CreateCore( &rc,s_data,nstg,rstg,cstg,nrow,ncol,nels,
          dobj,drlo,drup,dclo,dcup,mrow,mcol,dels,dqdg );


/* ...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;

/* ...malloc event indexing array */
    indx = (long *) malloc( (1+nindp)*sizeof(long) );
    memset( indx,0,(1+nindp)*sizeof(long) );

/* ...malloc label array */
    label = (long *) malloc( (nstg-1)*sizeof(long) );
```

```
/* Generate Scenarios from sampling */

    for (ns=0;ns < N_SAMPLES;ns++) {

/*   ...for each indep rv, process random number to get index */
       indx_beg = 0;
       for (jj=0;jj<nindp;jj++) {
               ev_indx = 0;
               irand = rand();
               dd = (double)irand/(double)RAND_MAX;
               dp = dprobs[indx_beg+ev_indx];
               while ( irand > dp*RAND_MAX )
                       dp += dprobs[++ev_indx + indx_beg];
               indx[jj] = ev_indx;
               indx_beg += nsamp[jj];
       }

/*   ...process indep events into a unique "event label"
       (there should be one such label for each stage in the simulation
       -- this example has only one stage ) */

       label[0]= indx[0];
       for (jj=1;jj<nindp;jj++)
               label[0] = label[0]*nsamp[jj] + indx[jj];

       dp = 1.0e0;
       dp /= N_SAMPLES;

/*   ...initialize tree if necessary, or add event to tree */
       if (!tree)
               ekks_InitializeTree( &rc,&tree,label,nstg-1,N_SAMPLES,dp );
       else
               ekks_AddToTree( &rc,tree,label,nstg-1,dp);
       if (rc>0) return;

/*   ...get scenario branching information from tree */
       ekks_GetScenarioFromTree( &rc,tree,ns,&ianc,&istg,&dwgt,

&nscen,&label );
       if (rc>0) return;

       if ( istg == 1 )        /* always true in this example */
               ianc = 0;

       istg++;                 /* always need to do this */

/*   ...assign values to right-hand sides based on index */
       evoffset=0;
       for (jj=0;jj<nindp;jj++) {
               drlo[ n_first_stg_rows + jj ] = demand[ indx[jj]+evoffset ];
               drup[ n_first_stg_rows + jj ] = demand[ indx[jj]+evoffset ];
               evoffset += nsamp[jj];
       }

/* ...add sample */
       ekks_AddScenario( &rc,s_data,nscen,ianc,istg,dwgt,1,
               nrow,ncol,nels,dobj,drlo,drup,dclo,dcup,
               mrow,mcol,dels,ADD_TO_CORE_VALUES );
```

```
        if (rc>1) return;
    }


/* Save SP-file as "sgbd_smpl.spl" */
    ekks_PutStochasticData( &rc,s_data,"sgbd_smpl.spl" );

/* Free Stoch_Data structures */
    ekks_FreeStructure(s_data);
}
```

[**Top of Page | Previous (Reference Section) | Next (Bibliography) | Roadmap**]

Bibliography

Bottom of Page

Select Author(s):

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.
Mathematical Programming Society Committee on Algorithms Newsletter, 17:1-20, 1987.
Available on the world wide web at http://ttg.sba.dal.ca/sba/profs/hgassmann/smps.html .

John R. Birge.
Decomposition and partitioning methods for multistage stochastic linear programs.
Operations Research, 33:989-1007, 1985.

J.R. Birge and F.V. Louveaux.
Introduction to Stochastic Programming
Springer-Verlag, New York, 1997.

IBM Corporation.
IBM Optimization Library, 1998.
Electronic publication http://www6.software.ibm.com/es/oslv2/features/lib.htm .

P. Kall and S.W. Wallace.
Stochastic Programming.
John Wiley and Sons, Chichester, England, 1994.

A.J. King.
Stochastic programming problems: Examples from the literature.
in Y. Ermoliev and R. J-B. Wets (eds.), Numerical techniques for stochastic optimization problems, pages 543-567, 1988.

Alan J. King.
SP/OSL version 1.0 stochastic programming interface user's guide.
Technical report, IBM Research Division, Thomas J. Watson Research Center, PO Box 218, Yorktown Heights, NY 10598, USA, 1994.

Alan J. King.
Implementation of dual decomposition in OSL.
Technical Report RC 20316, IBM Research Division, Thomas J. Watson Research Center, PO Box 218, Yorktown Heights, NY 10598, USA, 1995.
Available electronically in Postscript format.

M. Lane and P. Hutchinson.

A model for managing a certificate of deposit portfolio under uncertainty.
In M.A.H. Dempster, editor, *Stochastic Programming*. Academic Press, 1980.

F.V. Louveaux and Y. Smeers.
Optimal investments for electricity generation: A stochastic model and a test problem.
in Y. Ermoliev and R. J-B. Wets (eds.),Numerical techniques for stochastic optimization problems, pages 445-452, 1988.

M.V.F. Pereira and L.M.V.G. Pinto.
Stochastic optimization of a multireservoir hydroelectric system: A decomposition approach.
Water Resources Research, 21:779-792, 1985.

H. Raiffa.
Decision Analysis: Introductory Lectures on Choices Under Uncertainty.
Addison-Wesley, Reading, 1968.

Richard van Slyke and Roger J-B Wets.
L-shaped linear programs with applications to control and stochastic programming.
SIAM Journal on Applied Mathematics, 17:638-663, 1969.

[**Top of Page** | **Previous (Examples)** | **Next (Table of Contents)** | **Roadmap**]