LINDO API 6.0

User Manual

LINDO Systems, Inc.
1415 North Dayton Street, Chicago, Illinois 60642
Phone: (312)988-7422 Fax: (312)988-9065
E-mail: info@lindo.com

Published by

LINDO SYSTEMS INC.

1415 North Dayton Street
Chicago, Illinois 60642
Technical Support: (312) 988-9421
E-mail: tech@lindo.com
http://www.lindo.com

TABLE OF CONTENTS

Preface

LINDO Systems is proud to introduce LINDO API 6.0. The general features include a) stochastic optimization b) global and multistart solvers for global optimization, c) nonlinear solvers for general nonlinear optimization, d) simplex solvers for linear optimization e) barrier solvers for linear, quadratic and second-order-cone optimization f) mixed-integer solvers for linear-integer and nonlinear-integer optimization, g) tools for analysis of infeasible linear, integer and nonlinear models, h) interfaces to other systems such as MATLAB, Ox, Java and .NET and i) support of more platforms (see below). The primary solvers in LINDO API 6.0 are:

- **Stochastic Solver:**
  The stochastic programming solver provides the opportunity of decision making under uncertainty through multistage stochastic models with recourse. The user is required to express the uncertainity by providing distribution functions, either built-in or user-defined, and the stochastic solver will optimize the model to minimize the cost of the initial stage plus the expected value of recourse over the planning horizon. Advanced sampling modes are also available to approximate stochastic parameters from parametric distributions.

- **General Nonlinear Solver:**
  LINDO API is the first full-featured solver callable library to offer general nonlinear and nonlinear/integer capabilities. This unique feature allows developers to incorporate a single general purpose solver into their custom applications. As with its linear and integer capabilities, LINDO API provides the user with a comprehensive set of routines for formulating, solving, and modifying nonlinear models. The Nonlinear license option is required in order to use the nonlinear capabilities with LINDO API.

- **Global Solver:**
  The global solver combines a series of range bounding (e.g., interval analysis and convex analysis) and range reduction techniques (e.g., linear programming and constraint propagation) within a branch-and-bound framework to find proven global solutions to non-convex NLPs. Traditional nonlinear solvers can get stuck at suboptimal, local solutions. This is no longer the case when using the global solver.

- **Multistart Solver:**
  The multistart solver intelligently generates a sequence of candidate starting points in the solution space of NLP and mixed integer NLPs. A traditional NLP solver is called with each starting point to find a local optimum. For non-convex NLP models, the quality of the best solution found by the multistart solver tends to be superior to that of a single solution from a traditional nonlinear solver. A user adjustable parameter controls the maximum number of multistarts to be performed. See Chapter 7, *Solving Nonlinear Models*, for more information.

- **Barrier (Interior-Point) Solver:**
  Barrier solver is an alternative way for solving linear and quadratic programming problems. LINDO API's state-of-the-art implementation of the barrier method offers great speed advantages for large scale sparse models. LINDO API 6.0 also includes a special variant of the barrier solver specifically designed to solve *Second-Order-Cone* problems. See Chapter 6, *Solving Second-Order-Cone Models*, for more information.

- **Simplex Solvers:**

LINDO API 6.0 offers two advanced implementations of the primal and dual simplex methods as the primary means for solving linear programming problems. Its flexible design allows the users to fine tune each method by altering several of the algorithmic parameters.

- **Mixed Integer Solver:**
  The mixed integer solver's capabilities of LINDO API 6.0 extend to linear, quadratic, and general nonlinear integer models. It contains several advanced solution techniques such as a) cut generation b) tree reordering to reduce tree growth dynamically, and c) advanced heuristic and presolve strategies.

- **Model and Solution Analysis Tools:**
  LINDO API 6.0 includes a comprehensive set of analysis tools for a) debugging of infeasible linear, integer and nonlinear programs using series of advanced techniques to isolate the source of infeasibilities to smaller subset of the original constraints, b) performing sensitivity analysis to determine the sensitivity of the optimal basis to changes in certain data components (e.g. objective vector, right-hand-size values etc..).

- **Quadratic Recognition Tools:**
  The QP recognition tool is a useful algebraic pre-processor that automatically determines if an arbitrary NLP is actually a quadratic model. QP models may then be passed to the faster quadratic solver, which is available as part of the barrier solver option.

- **Linearization Tools:**
  Linearization is a comprehensive reformulation tool that automatically converts many non-smooth functions and operators (e.g., max and absolute value) to a series of linear, mathematically equivalent expressions. Many non-smooth models may be entirely linearized. This allows the linear solver to quickly find a global solution to what would have otherwise been an intractable nonlinear problem.

- **Decomposition Tools:**
  Many large scale linear and mixed integer problems have constraint matrices that are totally decomposable into a series of independent block structures. A user adjustable parameter can be set, so the solver checks if a model can be broken into smaller independent models. If total decomposition is possible, it will solve the independent problems sequentially to reach a solution for the original model. This may result in dramatic speed improvements. Refer to the *Block Structured Models* section in Chapter 10, *Analyzing Models and Solutions*, for more information.

- **Java Native Interface:**
  LINDO API includes Java Native Interface (JNI) support for Windows, Solaris, and Linux platforms. This new feature allows users to call LINDO API from Java applications, such as applets running from a browser.

- **MATLAB Interface:**
  The Matlab interface allows using LINDO API functions from within MATLAB. Using MATLAB's modeling and programming environment, you can build and solve linear, nonlinear, quadratic, and integer models and create custom algorithms based upon LINDO API's routines and solvers.

- **.NET Interface:**
  LINDO API includes C# and VB.NET interfaces that allow it to be used from within .NET's distributed computing environment (including Windows Forms, ADO.NET, and ASP.NET).

The interfaces are in the form of classes that allow managed .NET code to interact with unmanaged LINDO API code via the "System.Runtime.InteropServices" namespace.

- **Ox Interface:**
  This interface provides users of the Ox statistical package, the ability to call LINDO API's functions the same way they call native Ox functions. This offers greater flexibility in developing higher-level Ox routines that can set up and solve different kinds of large-scale optimization problems, testing new algorithmic ideas or expressing new solution techniques.

- **Platforms:**
  LINDO API 6.0 is currently available on Sparc Solaris 32/64 bit, Windows 32/64 bit, Linux 32/64-bit, Mac Intel 32-bit and Mac PowerPC 32-bit. For availability of LINDO API 6.0 on all other platforms, you may wish to contact LINDO Systems, Inc.

LINDO Systems, Inc
1415 N. Dayton
Chicago, Illinois
(312) 988 9421

info@lindo.com
http://www.lindo.com

January 2009

Chapter 1: Introduction

What Is LINDO API?

The LINDO Application Programming Interface (API) provides a means for software developers to incorporate optimization into their own application programs. LINDO API is designed to solve a wide range of optimization problems, including linear programs, mixed integer programs, quadratic programs, and general nonlinear non-convex programs. These problems arise in areas of business, industry, research, and government. Specific application areas where LINDO API has proven to be of great use include product distribution, ingredient blending, production and personnel scheduling, inventory management… The list could easily occupy the rest of this chapter.

Optimization helps you find the answer that yields the best result; attains the highest profits, output, or happiness; or achieves the lowest cost, waste, or discomfort. Often these problems involve making the most efficient use of your resources—including money, time, machinery, staff, inventory, and more. Optimization problems are often classified as linear or nonlinear, depending on whether the relationships in the problem are linear with respect to the variables.

The most fundamental type of optimization problems is the *linear program* (LP) of the form:

Minimize (or maximize) $c_1x_1 + c_2x_2 + \ldots + c_nx_n$

Such that

$$A_{11}x_1 + A_{12}x_2 + \ldots + A_{1n}x_n \;?\; b_1$$
$$A_{21}x_1 + A_{22}x_2 + \ldots + A_{2n}x_n \;?\; b_2$$
$$\vdots \qquad \ldots \qquad \vdots$$
$$A_{m1}x_1 + A_{m2}x_2 + \ldots + A_{mn}x_n \;?\; b_m$$
$$L_1 \le x_1 \le U_1$$
$$L_2 \le x_2 \le U_2$$
$$\vdots$$
$$L_n \le x_n \le U_n$$

where A_{ij}, c_j, b_i, L_j, U_j are known real numbers; *?* is one of the relational operators '≤', '=', or '≥'; and $x_1, x_2, \ldots, x_n$ are the decision variables (unknowns) for which optimal values are sought.

The expression being optimized is called the objective function and $c_1, c_2, \ldots, c_n$ are the objective coefficients. The relationships whose senses are expressed with *?* are the constraints; $A_{i1}, A_{i2}, \ldots, A_{in}$ are the coefficients; and b_i is the right-hand side value for the i^{th} constraint. L_j and U_j represent lower and upper bounds for the j^{th} decision variable and can be finite or infinite.

Sample Applications

The distribution package contains several sample application programs that illustrate the use of LINDO API using a high level programming language. The majority of the examples provided are in C/C++. Sample applications in other languages, such as Visual Basic, C#, Delphi, Fortran 90, and Java/J++ are also given.

> ***Note:*** The header files required by each programming language are located in *LINDOAPI\INCLUDE* directory. These headers contain macro definitions and function prototypes (calling sequences) for each programming language. For a detailed description of available LINDO API functions, please refer to Chapter 2, *Function Definitions*.

Each sample is located in a separate directory along with a MAKEFILE and/or an IDE Project (for Windows only) to build the application. Depending on your platform, use MAKEFILE.UNX (for Solaris and Linux) or MAKEFILE.WIN (for Windows).

Now, let's illustrate how to get started using LINDO API by setting up and solving a small LP using a programming language.

Array Representation of Models

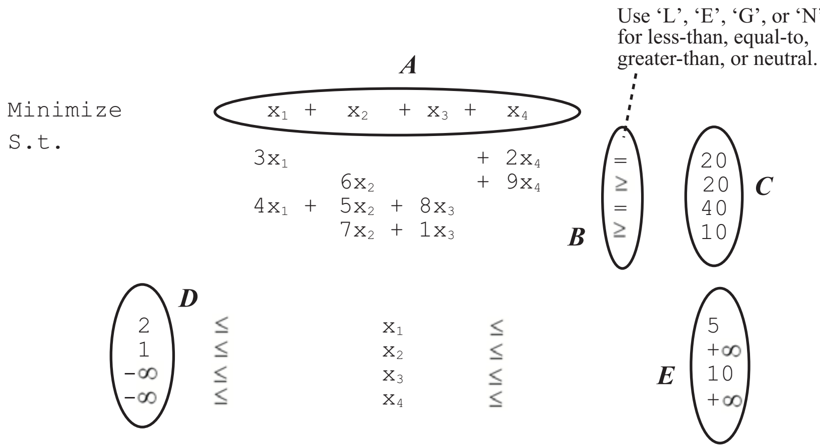
From within a programming environment, models can be entered into LINDO API in either of two ways: 1) characterize the model using data structures (array representation) and pass the associated data objects to LINDO API via model loading routines in LINDO API, or 2) read the model from a file directly into LINDO API via input/output routines available. Supported file formats are MPS, LINDO, and MPI formats, which are described in Appendices *B*, *C*, and *D*, respectively. Here, we focus on the first alternative, which we have referred to as array representation, and describe how to characterize an LP model within a programming environment. In our discussion, the terms 'array' and 'vector' are used interchangeably.

We will use a small LP with four decision variables x_1, x_2, x_3, x_4 (n=4) and four constraints (m=4) for our example. The lower and upper bounds are specified for each variable explicitly. If neither bounds are given, it would be assumed the variable is continuous, bounded below by zero and bounded from above by infinity. The model appears as follows:

$$
\begin{array}{lrcl}
\text{Minimize} & x_1 + x_2 + x_3 + x_4 & & \\
\text{S.t.} & & & \\
 & 3x_1 \qquad\qquad\quad + 2x_4 & = & 20 \\
 & 6x_2 \qquad + 9x_4 & \geq & 20 \\
 & 4x_1 + 5x_2 + 8x_3 \qquad & = & 40 \\
 & 7x_2 + 1x_3 \qquad & \geq & 10
\end{array}
$$

$$
\begin{array}{ccccc}
2 & \leq & x_1 & \leq & 5 \\
1 & \leq & x_2 & \leq & +\infty \\
-\infty & \leq & x_3 & \leq & 10 \\
-\infty & \leq & x_4 & \leq & +\infty
\end{array}
$$

The diagram below shows how each component of LP data, except the coefficients of the constraint matrix, can be trivially represented by vectors (arrays). The circled elements labeled *A,B,C,D*, and *E* in the following figure symbolize these components and refer to *objective coefficients*, *constraint senses*, *right-hand sides*, *lower-bounds*, and *upper-bounds*, respectively.



In this small example, these vectors translate to the following:

```
A = [ 1  1  1  1 ].
B = [ E  G  E  G ].
C = [ 20  20  40  10 ].
D = [ 2  1  -LS_INFINITY  -LS_INFINITY ].
E = [ 5  LS_INFINITY  10  LS_INFINITY ].
```

Each of these vectors can be represented with an array of appropriate type and passed to LINDO API via model loading routines. Although it is also possible to represent the coefficients of the constraint matrix with a single vector, a different representation, called the *sparse matrix representation*, has been adopted. This is discussed in more detail below.

Sparse Matrix Representation

LINDO API uses a sparse matrix representation to store the coefficient matrix of your model. It represents the matrix using three (or optionally four) vectors. This scheme is utilized, so it is unnecessary to store zero coefficients. Given that most matrix coefficients in real world math programming models are zero, this storage scheme proves to be very efficient and can drastically reduce storage requirements. Below is a brief explanation of the representation scheme.

We will use the coefficients of the constraint matrix in our sample LP from above. These are as follows:

$$\begin{array}{cccc} x_1 & x_2 & x_3 & x_4 \end{array}$$

$$\begin{bmatrix} 3 & 0 & 0 & 2 \\ 0 & 6 & 0 & 9 \\ 4 & 5 & 8 & 0 \\ 0 & 7 & 1 & 0 \end{bmatrix}$$

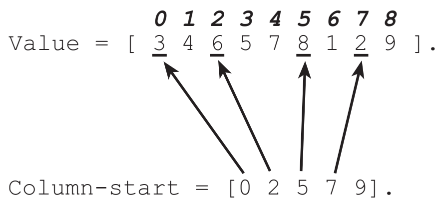
Three Vector Representation

Three vectors can represent a sparse matrix in the following way. One vector will contain all of the nonzero entries from the matrix, ordered by column. This is referred to as the *Value* vector. In our example, this vector has 9 entries and looks like:

```
Value = [ 3  4  6  5  7  8  1  2  9 ].
```

Note that all of the entries from the first column appear first, then the entries from the second column, and so on. All of the zeros have been stripped out.

In the second vector, which we call the *Column-start* vector, we record which points in the *Value* vector represent the start of a new column from the original matrix. The n^{th} entry in the *Column-start* vector tells us where in the *Value* vector to find the beginning of the n^{th} column. For instance, the column starts for the *Value* vector of our small example are underlined in the following diagram. Note that LINDO API uses zero-based counting, so the *Column-start* vector is as follows:



Note that the *Column-start* vector has one more entry than there are columns in our matrix. The extra entry tells LINDO where the last column ends. It will always be equal to the length of the *Value* vector.

From the *Column-start* vector, we can deduce which column is associated with each entry in our *Value* vector. The only additional information that we need is the row numbers of the entries. We store this information in a third vector, the *Row-index* vector. This vector is the same length as the *Value* vector. Each entry in the *Row-index* vector tells which row the corresponding entry from the *Value* vector belongs to. In our example, the number 3 belongs to the first row, which we call row 0, so the first entry in the *Row-index* vector is 0. Similarly, the second entry in the *Value* vector (4), belongs to the third row (row 2 when starting from zero), so the second entry of the *Row-index* vector is 2. Continuing in this way through the rest of the entries of the *Value* vector, the resulting *Row-index* vector appears as follows:

```
                0   1   2   3   4   5   6   7   8
Row-index = [ 0   2   1   2   3   2   3   0   1 ].
```

In summary, our transformation from a matrix into 3 vectors is:

$$\begin{bmatrix} 3 & 0 & 0 & 2 \\ 0 & 6 & 0 & 9 \\ 4 & 5 & 8 & 0 \\ 0 & 7 & 1 & 0 \end{bmatrix} \Rightarrow \begin{array}{ll} \text{Column-starts:} & [\,0\ 2\ 5\ 7\ 9\,] \\ \text{Value:} & [\,3\ 4\ 6\ 5\ 7\ 8\ 1\ 2\ 9\,] \\ \text{Row-index:} & [\,0\ 2\ 1\ 2\ 3\ 2\ 3\ 0\ 1\,] \end{array}$$

Four Vector Representation

The four vector representation allows more flexibility than the three vector representation. Use it when you expect to add rows to your original matrix (i.e., if you will be adding additional constraints to your model).

The four vector representation uses the same three vectors as above. However, it allows you to have “blanks” in your *Value* vector. Because of this, you must also pass a vector of column lengths, since the solver doesn’t know how many blanks there will be.

For example, suppose we wish to leave room for one additional row. Then, our *Value* vector becomes:

```
Value = [ 3  4  X  6  5  7  X  8  1  X  2  9  X ]
```

where the *X*’s represent the blanks. The blanks may be nulls or any other value, since they will be ignored for the time being.

Our *Column-start* vector becomes:

```
          0  1  2  3  4  5  6  7  8  9  10 11 12
Value = [ 3  4  X  6  5  7  X  8  1  X  2  9  X ].

Column-start = [0 3 7 10 13].
```

Our new vector is the *Column-length* vector. It will contain the length of each column (i.e., the number of nonzeros in each column). This allows the solver to skip the blanks (*X*’s) in the *Value* vector. In our small example, since the first column contains two nonzero and nonblank entries, the first element of the *Column-length* vector will be 2. Continuing through the remaining columns, the *Column-length* vector and its corresponding entries from the *Value* vector are as follows:

```
Column-length = [ 2  3  2  2 ].

Value = [ 3  4  X  6  5  7  X  8  1  X  2  9  X ].
```

Our *Row-index* vector is as before, except we add a blank for each blank in the *Value* vector. As with the *Value* vector, these blanks will be ignored, so they can contain any value. Thus, the *Row-index* vector becomes:

```
              0  1  2  3  4  5  6  7  8  9  10 11 12
Row-index = [ 0  2  X  1  2  3  X  2  3  X  1  2  X ].
```

In summary, the four vector transformation is:

$$\begin{bmatrix} 3 & 0 & 0 & 2 \\ 0 & 6 & 0 & 9 \\ 4 & 5 & 8 & 0 \\ 0 & 7 & 1 & 0 \end{bmatrix} \Rightarrow$$

Column lengths:	[2 3 2 2]
Column starts:	[0 3 7 10 13]
Values:	[3 4 X 6 5 7 X 8 1 X 2 9 X]
Row indexes:	[0 2 X 1 2 3 X 2 3 X 0 1 X]

Simple Programming Example

Up to this point, we have seen that the objective function coefficients, right-hand side values, constraint senses, and variable bounds can be stored in vectors of appropriate dimensions and the constraint matrix can be stored in three or four vectors using the sparse matrix representation. In this section, we show how these objects should be declared, assigned values, and passed to LINDO API to complete the model setup phase and invoke optimization.

Recall the small LP example model from the array representation section above:

$$
\begin{array}{llllll}
\text{Minimize} & x_1 + & x_2 + & x_3 + & x_4 & \\
\text{S.t.} & & & & & \\
& 3x_1 & & & + 2x_4 & = 20 \\
& & 6x_2 & & + 9x_4 & \geq 20 \\
& 4x_1 + & 5x_2 & + 8x_3 & & = 40 \\
& & 7x_2 & + 1x_3 & & \geq 10
\end{array}
$$

$$
\begin{array}{ccccc}
2 & \leq & x_1 & \leq & 5 \\
1 & \leq & x_2 & \leq & +\infty \\
-\infty & \leq & x_3 & \leq & 10 \\
-\infty & \leq & x_4 & \leq & +\infty
\end{array}
$$

It is easy to verify that the model has 4 variables, 4 constraints, and 7 nonzeros. As determined in the previous section, its constraint matrix has the following (three-vector) sparse representation:

```
Column-start = [ 0  2  5  7  9 ]
Values       = [ 3.0  4.0  6.0  5.0  7.0  8.0  1.0  2.0  9.0 ]
Row-index    = [ 0  2  1  2  3  2  3  0  1 ]
```

Other components of the LP data, as described above, are:

```
Right-hand side values = [ 20  20  40  10 ].
Objective coefficients = [ 1  1  1  1 ].
Constraint senses = [ E  G  E  G ].
Lower bounds = [ 2  1  -LS_INFINITY  -LS_INFINITY ].
Upper bounds = [ 5  LS_INFINITY  10  LS_INFINITY ].
```

Create an Environment and Model

Before any data can be input to LINDO API, it is necessary to request LINDO API to initialize the internal solvers by checking the license this user has and to get handles of the required resources (e.g., pointers to internal memory areas). This is achieved by creating a LINDO environment object and creating a model object within the environment. These reside at the highest level of LINDO API's internal object oriented data structure. In this structure, a model object belongs to exactly one environment object. An environment object may contain zero or more model objects.

The following code segment does this:

```
/* declare an environment variable */
pLSenv pEnv;

/* declare a model variable */
pLSmodel pModel;

/* Create the environment./
pEnv = LScreateEnv ( &nErrorCode, MY_LICENSE_KEY);

/* Create the model./
pModel = LScreateModel  ( pEnv, &nErrorCode);
```

The environment data type, *pLSenv*, and the model data type, *pLSmodel*, are both defined in the *lindo.h* header file. A call to *LScreateEnv()* creates the LINDO environment. Finally, the model object is created with a call to *LScreateModel()*. For languages other than C/C++ *pLSenv* and *pLSmodel* objects refer to integer types. The associated header files are located in the 'lindoapi/include' directory.

Load the Model

The next step is to set up the LP data and load it to LINDO API. This is generally the most involved of the steps.

Objective

The following code segment is used to enter the direction of the objective. The possible values for the direction of the objective are *LS_MAX* and *LS_MIN*, which are predefined macros that stand for maximize or minimize. For our sample problem, the objective direction is given as maximization with the following code:

```
int nDir = LS_MIN;
```

The constant terms in the objective function are stored in a double scalar with the following:

```
double dObjConst = 0.0;
```

Finally, the objective coefficients are placed into an array with the following:

```
double adC[4] = { 1., 1., 1., 1.};
```

Constraints

The following code segment is used to enter the number of constraints:

```
int nM = 4;
```

The constraint right-hand sides are place into an array with the following:

```
double adB[4] = { 20., 20., 40., 10. };
```

The constraint types are placed into an array with the following:

```
char acConTypes[4] = {'E', 'G', 'E' , 'G' };
```

The number of nonzero coefficients in the constraint matrix is stored:

```
int nNZ = 9;
```

Finally, the length of each column in the constraint matrix is defined. This is set to NULL in this example, since no blanks are being left in the matrix:

```
int *pnLenCol = NULL;
```

The nonzero coefficients, column-start indices, and the row indices of the nonzero coefficients are put into arrays with the following:

```
int anBegCol[5] = { 0   , 2   , 5   , 7   , 9 };
double   adA[9] = { 3.0, 4.0, 6.0, 5.0, 7.0, 8.0, 1.0, 2.0, 9.0 };
int    anRowX[9] = { 0   , 2   , 1   , 2   , 3   , 2   , 3   , 0   , 1   };
```

Note: Refer to the section *Sparse Matrix Representation* above for more information on representing a matrix with three or four vectors.

Variables

The following code segment is used to declare the number of variables:

```
int nN = 4;
```

The upper and lower bounds on the variables are defined with the following:

```
double pdLower[4] = {2, 1, -LS_INFINITY, -LS_INFINITY};

double pdUpper[4] = {5, LS_INFINITY, 10, LS_INFINITY};
```

Then, the variable types are placed into an array with the following:

```
char acVarTypes[4] = {'C','C','C','C'};
```

The variable types could actually be omitted and LINDO API would assume that the variables were continuous.

We have now assembled a full description of the model and pass this information to LINDO API with the following:

```
nErrorCode = LSloadLPData( pModel, nM, nN, nDir, dObjConst, adC, adB,
acConTypes, nNZ, anBegCol, pnLenCol, adA, anRowX, pdLower, pdUpper);
```

All LINDO API functions return an error code indicating whether the call was successful or not. If the call was successful, then the error code is zero. Otherwise, an error has occurred and its type could be looked up in Appendix A, *Error Codes*. It is imperative that the error code returned is always checked to verify that the call was successful.

Note: If there is a nonzero error code, the application program should stop, since the results would be unpredictable and it may cause the program to crash.

Solve

Since the model is an LP, a linear solver, such as the primal simplex method, can be used. The model is solved with the following call:

```
nErrorCode = LSoptimize( pModel, LS_METHOD_PSIMPLEX, &nSolStat);
```

Alternative solvers available for linear models include dual simplex and barrier (if licensed). When the second argument in the function call is set to LS_METHOD_FREE, LINDO API will decide the solver to use by examining its structure and mathematical content. See the *Common Macro Definitions* section of Chapter 2, *Function Definitions*, for more information on the predefined macros LS_METHOD_PSIMPLEX and LS_METHOD_FREE.

Retrieve the Solution

The next step is to retrieve the solution using solution query functions. Many of the LINDO API query functions need to have space allocated before calling the routine. You must be sure to allocate sufficient space for query routines that include a pointer to a string, an integer vector, a double precision vector, or character vector. If sufficient memory is not initially allocated, the application will crash once it is built and executed. See *Solution Query Routines* in Chapter 2, *Function Definitions*, for more information on which routines require space to be allocated for them. Refer to Chapter 3, *Solving Linear Programs*, for more details on building and solving the model and a programming example in Visual Basic.

Here, the objective value and optimal variable values will be displayed. The objective value is retrieved and printed with the following:

```
double adX[4];
nErrorCode = LSgetInfo( pModel, LS_DINFO_POBJ, &dObj);
printf( "Objective Value = %g\n", dObj);
```

See the context of the *LSgetInfo()* function in Chapter 2, *Function Definitions*, for more information on the predefined macro LS_DINFO_POBJ. It tells LINDO API to fetch the value of the primal objective value via the *LSgetInfo()* function. The optimal variable values are retrieved and printed with the following:

```
nErrorCode = LSgetPrimalSolution ( pModel, adX);
printf ("Primal values \n");
for (i = 0; i < nN; i++) printf( " x[%d] = %g\n", i, adX[i]);
printf ("\n");
```

The output of this program would appear as follows:

```
Objective Value = 10.44118
Primal values
 x[0] = 5
 x[1] = 1.176471
 x[2] = 1.764706
 x[3] = 2.5
```

Clear Memory

A last step is to release the LINDO API memory by deleting the LINDO environment with the following call:

```
nErrorCode = LSdeleteEnv( &pEnv);
```

This frees up all data structures LINDO API allocated to the environment and all of the environment's associated models.

References

Birge, J. and F. Louveaux(1997), *Introduction to Stochastic Programming*, Springer.

L'Ecuyer, P., R. Simard, E. Chen, and W. Kelton(2002), "An Object-Oriented Random-Number Package with Many Long Streams and Substreams", *Operations Research*, vol. 50, no. 6, pp. 1073-1075.

INDEX

1

A

B

C

D

E

J

K

L

M

N

O

P

Q

R

S

W