A systematic approach to OSL application programming

by A. S. Minkoff

The Optimization Subroutine Library (OSL) provides powerful tools for solving mathematical programming problems, and permits the integration of these tools into larger applications. In order to access the computational power, an application must translate data between forms used in the rest of the application and the form in which the data can be manipulated by OSL. OSL does not currently offer tools to aid in this translation. The purpose of this paper is to provide a systematic approach for translating symbolic representations of mathematical programming problems into computer code that performs all necessary interactions with both OSL and the rest of the application.

The Optimization Subroutine Library (OSL) provides powerful tools for solving mathematical programming problems, as attested to by other OSL papers in this issue. But in order to access that power, an OSL-based application must pose problems in a form that OSL recognizes and interpret the solution that OSL provides. Although there are common modes for symbolically expressing mathematical programming problems, OSL does not offer a facility that translates symbolic representations of problem elements into application source code; the burden rests on the application developer. This paper seeks to assist the OSL application developer by formulating a systematic approach for translating symbolic representations of mathematical programming problems into computer code that properly sets up the required inputs to OSL, runs the solver, and extracts the solution.

The heart of an OSL application is the mathematical programming statement of the phenomenon being modeled. The model can usually be expressed concisely in a symbolic format, with the advantage that the same problem structure can apply to a variety of problem instances in which numerical values and even problem dimensions may change.

The vehicle through which a mathematical model is expressed in symbolic form is usually called a *modeling language*. A number of modeling languages have been developed for use in mathematical programming.[1-3] These language implementations either have been or potentially can be set up to use OSL to perform the problem solution task. But one of OSL's strengths is that it can be integrated within larger applications. To make this integration seamless, the application developer must transform the pure model statement into computer code. The aim of this paper is to educate the developer as to how this can be done.

This paper makes several fundamental assumptions. First, the problem to be modeled is represented in a symbolic form. Although we do not give a full syntax for this representation format, we provide examples that allude to it. We focus on linear programming problems; our final dis-

cussion looks at modifications for other problem types handled by OSL. From the implementation perspective, we code the application in the C language. The problem is stored internally and loaded into OSL via the `ekklmdl` subroutine. We do not address the use of files formatted by MPS (the Mathematical Programming System). (See Chapter 10 of Reference 4.)

We begin with a discussion of how mathematical programming problems may be represented symbolically, and we present an example. Next, we contrast problem representation in OSL with the symbolic format. The organization of the OSL-related portion of an application is studied, with special attention paid to how this organization manifests itself in computer code. The code must be explicitly linked with the mathematical problem statement. An example helps reinforce these concepts. In the conclusion, this approach is explored further and extensions are discussed.

Mathematical programming model representation

Rather than use the syntax of a specific modeling language, we use an algebraic symbol representation to describe models. The representation must identify the following problem components:

- Various entities that constitute the phenomenon
- Attributes of these entities (both given and variable)
- Relationships among entities

Alternate terms for these components are employed, matching terminology used in many modeling languages. Entities are often referred to as "sets" or "indices"; their attributes are "data" that are input to the problem, and "variables" that are to be solved for; the "objective function" and "constraints" are the relationships of interest.

The problem description is usually organized using the devices of *groups* and *indices* within groups. The group concept lends modeling languages their power in describing very large models, in terms of overall numbers of variables and constraints, in relatively compact terms. An entity group is a set of similar entities keyed by some index. Indices will run over consecutive integers, from 0 to $N - 1$, where there are N entities in a particular set. (This numbering convention was adopted to correspond with C language array indexing schemes.) A compound entity may be defined as a combination of primitive entities; each compound entity may be pointed to with multiple indices. The index of a compound entity is an ordered set of integers. An attribute group is a measure of a primitive or compound entity that takes different values for each entity in the group. This gives rise to data groups (input attributes) and variable groups (output attributes). Although models considered here have single objective functions, they may have several constraint groups, each one running over one or more indices. A constraint group is a logical function of arithmetic expressions involving indices, data, and variables.

An example of a mathematical programming problem, expressed in the format just indicated, is given next. We study the meaning of the problem and the symbolic representation, then use this problem later to exemplify the techniques described in this paper. The problem arises in the context of fixed-income portfolio revision. Suppose we have a portfolio of fixed-income instruments and want to revise its composition so as to maximize expected return, but under a number of constraints. A formulation of this problem under stochastic conditions follows in traditional symbolic terminology. Afterwards, the various pieces of the formulation are classified according to the scheme for representing the model.

Define the following set indices:

- i Fixed-income instruments that may be bought or sold
- t Time period (when either a liability must be paid or a coupon payment is received); let the portfolio be revised at time 0, the first payment or liability occur at time 1, and the final period be T
- s Scenario for final instrument values and interest rates
- j Asset class to which instruments may belong

Input data to the problem are:

$PROB_s$	Probability of scenario s occurring
FLO_{its}	Cash inflow from security i in period t under scenario s (also incorporates projected value of security at $t = T$)

I_RATE_{ts}	Interest rate on short-term investments forecasted for period t under scenario s
B_RATE_{ts}	Interest rate on short-term borrowing forecasted for period t under scenario s
$LIAB_t$	Liability, or amount to be paid from cash, in period t
S_PRC_i	Price obtained from sale of one unit of instrument i
B_PRC_i	Price required to buy one unit of instrument i
$CASH0$	Initial cash on hand
$HOLD_i$	Initial holding of instrument i
GRP_{ij}	Asset class indicator (equals 1 iff instrument i belongs to class j)
P_GRP_j	Maximum dollar-weighted proportion of portfolio that may be invested in asset class j
MIN_INV_s	Minimum wealth acceptable at period T under scenario s

The decision variables for this problem are denoted as:

$port_i$	New portfolio holding in instrument i
$sell_i$	Amount of instrument i sold to revise portfolio
buy_i	Amount of instrument i bought to revise portfolio
inv_{ts}	Amount of cash invested short-term in period t under scenario s
bor_{ts}	Amount of cash borrowed short-term in period t under scenario s

With these definitions, we seek to maximize

$$\sum_s PROB_s inv_{Ts} \tag{1}$$

subject to

$$\sum_i FLO_{its} port_i + I\_RATE_{t-1,s} inv_{t-1,s} - inv_{ts} + bor_{ts} - B\_RATE_{ts} bor_{t-1,s} = LIAB_t \quad \forall s, t \geq 1 \tag{2}$$

$$inv_{0s} - bor_{0s} - \sum_i S\_PRC_i sell_i + \sum_i B\_PRC_i buy_i = CASH0 \quad \forall s \tag{3}$$

$$port_i + sell_i - buy_i = HOLD_i \quad \forall i \tag{4}$$

$$\sum_i (GRP_{ij} - P\_GRP_j) S\_PRC_i port_i \leq 0 \quad \forall j \tag{5}$$

$$inv_{Ts} \geq MIN\_INV_s \quad \forall s \tag{6}$$

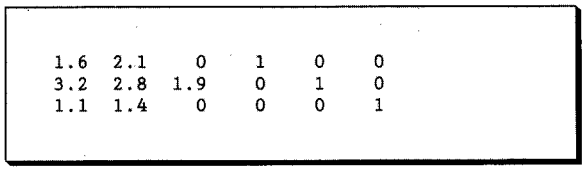
The literal interpretation of this problem is one of maximizing expected wealth at the end of the time horizon (period T). This is stated in Equation 1. By including instrument value at time T in FLO_{iTs}, inv_{Ts} represents not only cash on hand at the end of the horizon, but also total value of the portfolio. Equation 2 tracks the flow of money at each time period and scenario, whether the money comes from coupons, final instrument value, investments, or borrowing. Initial cash on hand (period 0) is determined through Equation 3. It takes into account any cash on hand prior to revising the portfolio, plus new purchases, sales, and borrowing. Equation 4 determines the amount of each instrument bought and sold by comparing the new holding of the instrument to the original holding. Equation 5 limits the amount of investment in each asset class. Finally, Equation 6 seeks to limit risk by putting a lower bound on terminal wealth under each scenario. Note that each variable must take on only nonnegative values.

The primitive entities in this problem are the fixed-income instruments (for example, corporate bonds from a certain company, of a certain coupon rate and maturity), time periods (like June 1992), scenarios, and asset classes (e.g., junk bonds). Compound entities are combinations of two or more primitive entities, such as period-scenario. A relevant attribute of this compound entity is the interest rate data group I_RATE, whereas the liability LIAB is an attribute solely of time period. Equations 1–6 are all relationships among entities, reflected in their attributes. Equation 1 is an arithmetic relationship whose value we want to maximize, while the others are logical relationships that must be satisfied in any acceptable solution to the problem.

Problem representation in OSL

OSL offers several means to represent instances of mathematical programming problems. It takes a substantial amount of work to transform an alge-

Figure 1 Matrix to illustrate storage-by-indices format



```
1.6  2.1    0    1    0    0
3.2  2.8  1.9    0    1    0
1.1  1.4    0    0    0    1
```

braic statement in the form described in the last section to one of these formats. We can, however, build a systematic and relatively straightforward approach on the problem representation format known in OSL as *storage by indices*. This is an internal format, meaning the problem is stored in arrays inside the program, as opposed to externally in a file (OSL permits this option, too, as it accepts MPS-format files[4]). A series of arrays and integer variables is used to contain problem information. These are essentially the arguments to OSL subroutine `ekklmdl`, and are reviewed as follows:

`nrow`	Number of constraint rows in problem
`ncol`	Number of variable columns in problem
`nel`	Number of nonzero coefficients in the constraint matrix
`dobj`	Objective function coefficients
`drlo`	Lower bounds on row activities
`drup`	Upper bounds on row activities
`dclo`	Lower bounds on column activities
`dcup`	Upper bounds on column activities
`mrow`	Row indices for nonzero coefficients in constraint matrix
`mcol`	Column indices for nonzero coefficients in constraint matrix
`dels`	Values of nonzero coefficients in constraint matrix

The storage-by-indices format applies to the last three arrays. To further illustrate how this format works, consider the simple matrix shown in Figure 1. This matrix would be stored as:

```
mrow = ( 1   1   2   2   2   3   3  1  2  3)
mcol = ( 1   2   1   2   3   1   2  4  5  6)
dels = (1.6  2.1  3.2  2.8  1.9  1.1  1.4  1  1  1)
```

The nonzero elements of the matrix are contained in `dels`. At the i-th position of `dels`, the value there is located in row `mrow`[i] and column `mcol`[i] of the matrix.

We observe here a fundamental conflict between the storage-by-indices format and the group representation of the mathematical model. In OSL, indices are *global*, whereas in the group representation, they are *local*. For instance, in the portfolio revision problem, suppose there are 10 instruments labeled 1 through 10 (temporarily disregarding the convention of starting with 0), and the columns are sorted in the order that the variable groups are listed. Then the column representing $\texttt{sell}_3$ is actually the 13th column, globally speaking, in the model. This is because ten `port` columns precede the first `sell` column. The local index of $\texttt{sell}_3$ is 3, and the global index is 13. This conflict could be alleviated to some extent through the use of OSL's block description facility, but this introduces other bookkeeping measures and will not be adopted here.

The indexing conflict is relieved with a pair of index adjustment schemes:

1. A local index within a group that converts single or multiple indices to an integer
2. A local-to-global index conversion scheme

The local scheme is trivial when a group runs over a single index. Where there are multiple indices, the last index is varied first in making the correspondence between ordered sets of indices and the local index integer. The local-to-global conversion is accomplished by determining offsets for each group relative to global column 0 (for variables) or global row 0 (for constraints). The first variable group has a column offset of 0, and the second has a column offset equal to the number of columns in the first variable group. Each subsequent variable group's offset is equal to the number of columns processed prior to reaching that variable group. The same scheme is applied to global row index determination. Note that lower and upper bounds and objective function coefficients must be loaded into the respective arrays under the global indexing scheme.

The OSL application

OSL, as a subroutine library, is designed to be integrated within systems that may perform other functions as well. OSL may be imbedded in an application that also contains database manager,

spreadsheet, graphics, user interface, and other facilities. We isolate the optimization function in this paper by assuming that the relevant operations are performed in a function (in the sense used in the C language) called from another program. We also discuss the nature of communication between the function and the calling program.

The optimization-specific function, which we label `oslproc`, can be divided into six tasks:

1. Function initialization
2. Receive input data
3. Prepare OSL inputs
4. Solve problem
5. Retrieve OSL outputs
6. Transmit solution

Function initialization covers declarations of all variables and constants and definitions of all macros that are used in the processing portions of the program. Values of some variables are initialized in this task as well.

The task of acquiring input data, whether from flat files, databases, and/or keyboard entry, is assumed to occur in another module. Receiving input data corresponds here to having data passed from the calling program as arguments to `oslproc`, or to assigning data variables to the `extern` storage class (and using the same names for the variables in the calling program).

The input arguments to OSL subroutines are assembled in the next task. Preparing OSL inputs involves taking the raw input data and organizing them, as described in the previous section.

OSL subroutines permit varying levels of control in the solution of mathematical programming problems. Although this is where the real power of OSL lies, we do not dwell at length on solving problems here. OSL may be accessed with a few subroutine calls, and efficient use of OSL is covered elsewhere.[4]

OSL provides a routine for printing solutions, but the destination of the printout is generally a file or the terminal. We need to store the solution in array variables, so that it can be manipulated by other modules in the system. Processing OSL outputs is concerned with extracting the solution from the main work array.

Once the solution is extracted, we need to transmit the solution back to the calling program. We can use the same methods here that we use for receiving input data.

The next sections detail one way to implement each of these tasks. We describe and illustrate the coding of these tasks in the C language.

Coding the OSL application

This section describes how one proceeds from a symbolic statement of a mathematical programming problem to the source code for the optimization component of a system that uses OSL. The objective of this section is to relate how to produce the function `oslproc`. The function is written in C and is designed to be called from another module that acquires the data and stores them in the arrays we name in the problem statement. The task of receiving the input data is achieved by constructing a header file that declares the pertinent variables with the appropriate storage class. The header file will be included in both the calling program and `oslproc`, to assure consistency in the use of the variables. The bulk of `oslproc` is devoted to the OSL input preparation task. Once the OSL input is ready, we invoke the appropriate OSL subroutines. Afterwards, `oslproc` places the solution in arrays named after the problem statement's variable groups. These arrays have storage class `extern`, so that other modules may access the solution for report writing and other functions. In the next section, we show how these techniques are applied to the problem statement described in the section on mathematical programming model representation. Below, we study the implementation of each task enumerated in the previous section.

Function initialization. Before the code for `oslproc` is listed, a series of header files must be included. These include standard header files such as `stdlib.h`, the OSL header file `ekkc.h`, and an application-specific header file. Call this last file `oslproc.h`.

The file `oslproc.h` performs several duties. It declares `oslproc` and some special-purpose procedures used in preparing OSL inputs. It declares a number of OSL subroutine arguments, both scalar and array. This also includes the use of macros to define the sizes of the arrays and to set up synonyms for key elements of some of these arrays.

Figure 2 addcol

```
void addcol(double aobj, double alo, double aup)
{
  dobj[ncol]= aobj;
  dclo[ncol]= alo;
  dcup[ncol]= aup;
  ncol++;
  return;
}
```

The file oslproc.h also declares the problem-specific arrays that match data and variable groups from the problem statement. For each set, we define a variable that holds the number of elements in the set and a macro that supplies the maximum number of elements permissible in the set. The latter is used in the declarations of the model's arrays (hence storage capacity may be easily modified; alternatively, one may allocate space for these arrays dynamically). Finally, we define macros for each variable and constraint group that obtains the local index from the indices that are used to refer to the individual variable or constraint.

Additional local variables are declared within oslproc. These are the temporary floating-point and integer variables used as arguments to the special-purpose functions, arrays for column and row offsets, a return code variable, and set index variables. The other initialization tasks are to allocate space for the OSL work array (if this is being done dynamically), and to set row, column, and constraint matrix element counters to zero.

Receive input data. This task has already been accomplished by virtue of having the data stored in arrays with the extern storage class.

Prepare OSL inputs. This task is accomplished sequentially. First, columns are defined to the problem representation, one by one. Then each row is established by first defining the row, then each of the nonzero elements in the row. We use the word "add" to refer to this process of defining a column, row, or element. The special-purpose functions support this process, so we study them first. (Note that these functions are not supplied with OSL.)

Figure 2 lists the function addcol. It is used to add a new column to the internal problem representation. The function addcol takes these arguments:

aobj	Column's objective function coefficient
alo	Lower bound for column
aup	Upper bound for column

These values are added to the objective value array dobj, and the lower and upper bound arrays dclo and dcup, all at position ncol. Arrays ncol, dobj, dclo, and dcup will be declared to be global in scope, so they need not be included as arguments to addcol. The column counter variable ncol is incremented only after the arguments are loaded in, because the ncol-th element is stored at position ncol-1 in the C arrays.

Function addrow is shown in Figure 3. It is similar to addcol except that rows have no objective function coefficients, only bounds drlo and drup. Function addel in Figure 4 is used to add a nonzero element from the constraint matrix to the OSL arrays mrow, mcol, and dels. Argument icol is the global index of the added element's column, and ael is the value of the element. It is assumed that the element belongs in the row most recently added to the problem representation. The element is only added if it is nonzero. Again, since the OSL arrays have global scope, they can be omitted from the function call. It is important to note that in OSL, row and column numbering start at 1, even when the driver program is written in C. The function addrow increments nrow prior to the respective addel calls, so nrow can be used as is. But one must be added to icol, because we have chosen to number these rows internally starting at zero.

Given the above functions, the preparation of OSL input follows somewhat straightforwardly. (The reader may find it helpful to refer to the code listings in the next section through the remainder of this section.) First the variable groups are processed. The processing for each variable group works as follows. The column offset is determined as described in the section on problem representation in OSL. Then a series of for-loops is initiated, one for each of the group's indices. We begin with the leftmost index, meaning that we increment the rightmost index earliest, as we have indicated previously. Following the for-loop initializations, we may want to test membership of the index combination in some subset, if not all combinations of indices need be included in the model. The objective function coefficient and lower and upper bounds for the column to be added are obtained from the model statement, and `addcol` is called with these arguments. Then the for-loops are closed and we move on to the next variable group.

Next we must treat the rows of the problem, by processing each constraint group. We obtain the row offset for the constraint group as we did for variable groups. Likewise, we begin for-loops for each of the indices that the group runs over. Inside the loops, we check for subset membership and specify lower and upper row bounds. A call to `addrow` adds the new row to the problem. Now, within that loop, the nonzero elements for the row must be added to the problem representation through calls to `addel`. Three different types of element processing may be carried out, depending on how variables appear in a constraint group in the model statement:

1. A variable appears in the constraint group in a summation term.
2. A variable appears in the constraint group singly, without a summation term.
3. A variable does not appear in the constraint group.

Obviously, in the last case, it is not necessary to add elements to the constraint arrays. The difference between the first and second cases is that the first case requires for-loops over the summation indices to add multiple columns from the variable group to the element set, whereas the second calls for only a single element. In each case, we must determine the global index of the column to be added. This number is found by adding the local column index, as supplied by the local index macro, to the column offset for that group. This column number, and the value of the coefficient, are the arguments to `addel`.

Figure 3 addrow

```
void addrow(double alo, double aup)
{
  drlo[nrow]= alo;
  drup[nrow]= aup;
  nrow++;
  return;
}
```

Figure 4 addel

```
void addel(long icol, double ael)
{
  if (ael != 0) {
    mrow[nel]= nrow;
    mcol[nel]= icol + 1;
    dels[nel]= ael;
    nel++;
  }
  return;
}
```

It should be noted that if a constraint group contains only one variable group that is not in a summation, then that constraint may be in fact a lower or upper bound on the variable. If so, it is more efficient to use this information in determining one or both of the variable group's bounds, and not include it as a constraint group.

Solve problem. Once all the arrays have been set up, the OSL subroutines are invoked. The subroutine sequence can range from a bare minimum to more sophisticated measures that can speed the solution process, including pre- and post-solve, scaling, parameter setting, decomposition, and making use of previous solutions or bases. The content of the code that performs this task is independent to some extent of the problem instance, although there will be cases in which one might obtain solutions more efficiently by applying some model-specific manipulations. Efficient use of OSL subroutines is covered elsewhere, and

Figure 5 Header file

```
#define maxspc 100000
#define MXEL 10000
#define MXROW 500
#define MXCOL 500
#define  osliln   47
#define  oslrln   34
#define  oslnln   30

void addcol(double, double, double);
void addrow(double, double);
void addel(int, double);
int oslproc();

LOC int nrow,ncol,nel;
LOC int mrow[MXEL],mcol[MXEL];
LOC double dobj[MXCOL],dclo[MXCOL],dcup[MXCOL];
LOC double drlo[MXROW],drup[MXROW],dels[MXEL];
LOC double *dspace, oslr[oslrln];
LOC int rtcode, osli[osliln], osln[oslnln];

#define MX_i 20
LOC int n_i;
#define MX_t 20
LOC int n_t;
#define MX_s 20
LOC int n_s;
#define MX_j 20
LOC int n_j;

LOC double PROB[MX_s];
LOC double FLO[MX_i][MX_t][MX_s];
LOC double I_RATE[MX_t][MX_s];
LOC double B_RATE[MX_t][MX_s];
LOC double LIAB[MX_t];
LOC double S_PRC[MX_i];
LOC double B_PRC[MX_i];
LOC double CASH0;
LOC double HOLD[MX_i];
LOC double GRP[MX_i][MX_j];
LOC double P_GRP[MX_j];
LOC double MIN_INV;
LOC double port[MX_i];
#define port_idx(i) (i)
LOC double sell[MX_i];
#define sell_idx(i) (i)
LOC double buy[MX_i];
#define buy_idx(i) (i)
LOC double inv[MX_t][MX_s];
#define inv_idx(t,s) (t)*n_s + (s)
LOC double bor[MX_t][MX_s];
#define bor_idx(t,s) (t)*n_s + (s)
```

will not be discussed further here. (See Chapter 8 of Reference 4.)

Retrieve OSL outputs. After the solution is determined, it must be retrieved. It is stored in the OSL work area, but somewhere in the middle and according to global column indices. To extract it, we first use a call to `ekknget` to obtain the index where the solution starts in the work area. This will be in the 7th position in the `osln` array argument to `ekknget`, but in C, this would be denoted `osln`[6]. Also, because OSL is FORTRAN-based, we must subtract one from `osln`[6] to get the C array position of the first column of the solution. Then the

Figure 6 Initialization

```
int oslproc()
{
  double dtemp,dtemp_o,dtemp_l,dtemp_u;
  int icol,coloff[30],rowoff[30],rc;
  int i,t,s,j;

  dspace = (double *) malloc(maxspc*sizeof(double));
  if (!dspace) return (1000);

  nrow = ncol = nel = 0;
```

global column number must be added to this position, to get the storage location of the solution value for the corresponding variable. We may use the same global column number calculation as was done in constraint processing when adding elements. Dual variable values are extracted in a similar fashion (although this is not done in the example that follows).

Transmit solution. The arrays storing the solution elements were previously defined as external arrays, hence the values will be available to the calling program. Employing this approach, it is not necessary to take any action to transmit the solution.

Example

The approach outlined in the preceding section is illustrated next, using the model statement given in the section on mathematical programming model representation.

The header file is listed in Figure 5. Correspondences may be detected between elements of the set, data group, and variable group descriptions, and declarations in the header. More than one index may range over a given set, although this is not the case here. A set is identified by the first listed index for it. Then the actual size of the set will be stored in the variable named by the set identifier prefixed by `n_`. The macro for the maximum size of the set is denoted by `MX_` prepended to the set identifier.

This header is included in the calling program, so that all variables are accessible to both calling program and `oslproc`. We use the `LOC` macro to have

Figure 7 Preparing columns

```
/* Variable port */
coloff[0] = ncol;
for (i = 0; i < n_i; i++) {
  dtemp_o = 0;
  dtemp_l = 0;
  dtemp_u = 1e31;
  addcol(dtemp_o,dtemp_l,dtemp_u);
}
/* Variable sell */
coloff[1] = ncol;
for (i = 0; i < n_i; i++) {
  dtemp_o = 0;
  dtemp_l = 0;
  dtemp_u = 1e31;
  addcol(dtemp_o,dtemp_l,dtemp_u);
}
/* Variable buy */
coloff[2] = ncol;
for (i = 0; i < n_i; i++) {
  dtemp_o = 0;
  dtemp_l = 0;
  dtemp_u = 1e31;
  addcol(dtemp_o,dtemp_l,dtemp_u);
}
/* Variable inv */
coloff[3] = ncol;
for (t = 0; t < n_t; t++) {
for (s = 0; s < n_s; s++) {
  if (t == n_t - 1) {
    dtemp_o = PROB[s];
    dtemp_l = MIN_INV[s];
  } else {
    dtemp_o = 0;
    dtemp_l = 0;
  }
  dtemp_u = 1e31;
  addcol(dtemp_o,dtemp_l,dtemp_u);
}
}
/* Variable bor */
coloff[4] = ncol;
for (t = 0; t < n_t; t++) {
for (s = 0; s < n_s; s++) {
  dtemp_o = 0;
  dtemp_l = 0;
  dtemp_u = 1e31;
  addcol(dtemp_o,dtemp_l,dtemp_u);
}
}
```

Figure 8A Preparing rows

```
/* Constraint group 1: Track cash */
rowoff[0] = nrow;
for (s = 0; s < n_s; s++) {
for (t = 1; t < n_t; t++) {
  dtemp_l = LIAB[t];
  dtemp_u = LIAB[t];
  addrow(dtemp_l,dtemp_u);
  for (i = 0; i < n_i; i++) {
    /* Variable port */
    icol = coloff[0] + port_idx(i);
    dtemp = FLO[i][t][s];
    addel(icol,dtemp);
  }
    /* Variable inv */
    icol = coloff[3] + inv_idx(t-1,s);
    dtemp = I_RATE[t-1][s];
    addel(icol,dtemp);
    icol = coloff[3] + inv_idx(t,s);
    dtemp = -1;
    addel(icol,dtemp);
    /* Variable bor */
    icol = coloff[4] + bor_idx(t,s);
    dtemp = 1;
    addel(icol,dtemp);
    icol = coloff[4] + bor_idx(t-1,s);
    dtemp = -B_RATE[t][s];
    addel(icol,dtemp);
}
}
```

Figure 8B Preparing rows

```
/* Constraint group 2: Track initial cash */
rowoff[1] = nrow;
t = 0;
for (s = 0; s < n_s; s++) {
  dtemp_l = CASH0;
  dtemp_u = CASH0;
  addrow(dtemp_l,dtemp_u);
    /* Variable inv */
    icol = coloff[3] + inv_idx(t,s);
    dtemp = 1;
    addel(icol,dtemp);
    /* Variable bor */
    icol = coloff[4] + bor_idx(t,s);
    dtemp = -1;
    addel(icol,dtemp);
  for (i = 0; i < n_i; i++) {
    /* Variable sell */
    icol = coloff[1] + sell_idx(i);
    dtemp = -S_PRC[i];
    addel(icol,dtemp);
  }
  for (i = 0; i < n_i; i++) {
    /* Variable buy */
    icol = coloff[2] + buy_idx(i);
    dtemp = B_PRC[i];
    addel(icol,dtemp);
  }
}
```

Figure 8C Preparing rows

```
/* Constraint group 3: Determine sell and buy */
rowoff[2] = nrow;
for (i = 0; i < n_i; i++) {
  dtemp_l = HOLD[i];
  dtemp_u = HOLD[i];
  addrow(dtemp_l,dtemp_u);
    /* Variable port */
    icol = coloff[0] + port_idx(i);
    dtemp = 1;
    addel(icol,dtemp);
    /* Variable sell */
    icol = coloff[1] + sell_idx(i);
    dtemp = 1;
    addel(icol,dtemp);
    /* Variable buy */
    icol = coloff[2] + buy_idx(i);
    dtemp = -1;
    addel(icol,dtemp);
}
```

these arrays and variables declared with the `extern` storage class in `oslproc`, but without it in the calling program, where the arrays are originally set up. One may verify that the macros for determining

Figure 8D Preparing rows

```
/* Constraint group 4: Limit class investment */
rowoff[3] = nrow;
for (j = 0; j < n_j; j++) {
  dtemp_l = -1e31;
  dtemp_u = 0;
  addrow(dtemp_l,dtemp_u);
  for (i = 0; i < n_i; i++) {
    /* Variable port */
    icol = coloff[0] + port_idx(i);
    dtemp = (GRP[i][j]-P_GRP[j])*SPRC[i];
    addel(icol,dtemp);
  }
}
```

Figure 9 Solving the problem

```
ekkdsca(&rtcode,dspace,maxspc,1);
ekkdscm(&rtcode,dspace,1,5);
ekkrget(&rtcode,dspace,oslr,oslrln);
oslr[2] = -1.0;
ekkrset(&rtcode,dspace,oslr,oslrln);
ekklmdl(&rtcode,dspace,1,nrow,ncol,nel,dobj,drlo,drup,dclo,dcup,mrow,mcol,dels);
ekkslv(&rtcode,dspace,1,2);
```

local column indices will give the appropriate results. For example, `inv_idx(0,0)` will return 0, and the rightmost index is incremented soonest.

We now examine the function `oslproc`. The rest of the initialization task (beyond what is done in the header file) is listed in Figure 6. Integer variables corresponding to each set index are declared here.

Figure 7 displays the code that adds all the columns to the problem representation. Observe that for variable group `inv`, a lower bound may be obtained from Equation 6 of the problem formulation. But this bound, and the objective coefficient, only apply to the last period.

Constraint processing code is shown in Figure 8, A–D. The fifth constraint group, corresponding to Equation 6, was previously diagnosed as a column bound, so no processing is done here. Also observe processing for constraint group 1; variables `inv` and `bor` appear twice in Equation 2, but with different subscripts, and this must be incorporated in the formation of the OSL arrays.

A minimum set of OSL calls to solve the optimization problem is provided in Figure 9. Necessary OSL initialization is performed by `ekkdsca` and `ekkdscm`. Because the objective is to maximize, we use the calls to `ekkrget` and `ekkrset` to change OSL's default of minimization. The internally stored problem is converted by `ekklmdl` to a format that OSL can work with directly. The single call to `ekksslv` solves the linear programming problem with the simplex algorithm. Additional code should be placed here to determine the final problem status (optimal, infeasible, unbounded, etc.), and to check the return codes from each OSL call. The final problem status should be passed back to the calling program as the return code for `oslproc`.

Figure 10 Extracting the solution

```
/* Variable port */
for (i = 0; i < n_i; i++) {
  icol = coloff[0] + port_idx(i);
  port[i] = dspace[osln[6]-1 + icol];
}
/* Variable sell */
for (i = 0; i < n_i; i++) {
  icol = coloff[1] + sell_idx(i);
  sell[i] = dspace[osln[6]-1 + icol];
}
/* Variable buy */
for (i = 0; i < n_i; i++) {
  icol = coloff[2] + buy_idx(i);
  buy[i] = dspace[osln[6]-1 + icol];
}
/* Variable inv */
for (t = 0; t < n_t; t++) {
for (s = 0; s < n_s; s++) {
  icol = coloff[3] + inv_idx(t,s);
  inv[t][s] = dspace[osln[6]-1 + icol];
}
}
/* Variable bor */
for (t = 0; t < n_t; t++) {
for (s = 0; s < n_s; s++) {
  icol = coloff[4] + bor_idx(t,s);
  bor[t][s] = dspace[osln[6]-1 + icol];
}
}
```

Figure 10 shows how the solution is retrieved from the work area and stored in the appropriate arrays. The arrays are stored externally, so this information will be passed back implicitly to the calling program.

Conclusion

We have described an approach to developing OSL applications that can be applied to most formulations of mathematical programming problems. From a symbolic representation of the problem, the application developer may generate computer code that will perform the formatting of data for OSL, solve the problem, and retrieve the solution. The emphasis is on seamlessly integrating OSL with the rest of the application by working with data directly as the data may be stored elsewhere in the application.

The approach described in this paper may be applied to any linear programming problem that can be represented in the symbolic format described. But OSL handles other mathematical programming variations as well. The same techniques may be modified to allow one to write applications for quadratic and linear mixed-integer programming. For quadratic programming, one must recognize which terms in the objective function contain the product of two variables (or one variable squared). Then we invoke a function similar to `addel` to add an element to the quadratic matrix. And the solve task must include calls to `ekkqmdl` (to load the quadratic piece of the objective function) and `ekkqslv` (to solve it).

Release 2 of OSL allows the specification of mixed-integer programming problems within an OSL application, instead of relying on an MPS-type deck residing in an external file. Then, once again, these procedures may be applied. The problem statement must contain indications of which variable groups are integer-valued. This information is used to create the arrays that will be used in the call to `ekkimdl`, where integer variable information is communicated to OSL. The solve task must likewise be modified.

Because the programming approach described here is fairly well-defined, it is possible to codify the rules into a program that does the code generation automatically. The author is currently engaged in developing such a tool.

Cited references

1. A. Brooke, D. Kendrick, and A. Meeraus, *GAMS: A User's Guide*, Scientific Press, Redwood City, CA (1988).
2. R. Fourer, D. M. Gay, and B. W. Kernighan, "A Modeling Language for Mathematical Programming," *Management Science* **36**, No. 5, 519–554 (1990).
3. A. M. Geoffrion, "SML: A Model Definition Language for Structured Modeling," Working Paper No. 360, Western Management Science Institute, University of California, Los Angeles (1988).
4. *Optimization Subroutine Library: Guide and Reference*, SC23-0519-2, IBM Corporation; available through IBM branch offices.

Accepted for publication September 19, 1991.

Alan S. Minkoff *IBM Corporation, 33 Maiden Lane, New York, New York 10038.* Dr. Minkoff is a financial quantitative analyst with IBM's Mathematical and Analytics Computation Center (MACC) in New York City. His main responsibility is to provide expertise to customers in the fields of optimization (including the use of IBM's Optimization Subroutine Library and MPSX) and neural networks. He also conducts research into applications of optimization in the financial and other industries. His main research interests are in applications of the new breed of optimization heuristics, including simulated annealing and tabu search, and in developing tools to enhance

the use of optimization. Prior to joining MACC, Dr. Minkoff worked for IBM's Data Systems Division, developing operations research-based decision support tools for various manufacturing applications. Dr. Minkoff got his undergraduate degree, a B.S. in statistics, at Princeton University, and received a Ph.D. in operations research from MIT.

Reprint Order No. G321-5460.