IMPLEMENTATION AIDS FOR OPTIMIZATION ALGORITHMS THAT SOLVE SEQUENCES OF LINEAR PROGRAMS BY THE REVISED SIMPLEX METHOD

Larry Nazareth

November 1982
WP-82-107

INTERNATIONAL INSTITUTE FOR APPLIED SYSTEMS ANALYSIS
2361 Laxenburg, Austria

ABSTRACT

We describe a collection of subroutines designed a) to facilitate the implementation of algorithms that are based upon linear programming, b) to serve as a tutorial on the development of such implementations. We make this collection the basis for a discussion of some of the broader issues of software development.

IMPLEMENTATION AIDS FOR OPTIMIZATION ALGORITHMS THAT SOLVE SEQUENCES OF LINEAR PROGRAMS BY THE REVISED SIMPLEX METHOD

Larry Nazareth

1. Introduction

In this paper we describe a collection of subroutines designed with two purposes in mind. Firstly, it is designed to facilitate the implementation of algorithms which solve one or more linear programs in sequence, by the revised simplex method. For convenience, throughout this paper, we shall refer to such algorithms as LP algorithms. Examples are algorithms based upon the Decomposition Principle of Dantzig and Wolfe, 1961 or certain algorithms for solving stochastic programs, see Nazareth and Wets, 1982. Secondly, the collection is designed to serve as a tutorial on the development of such implementations.

We make this collection the basis for a discussion of some of the broader issues of LP software development. In particular, we discuss the idea of hierarchical implementation of LP algorithms, and this enables us

to be more specific about the purposes and limitations of our routines.

2. Hierarchical Implementation of LP Algorithms

In the early stages of the development of an LP algorithms, a useful computational aid is a suitable high level language, preferably one available in an interactive computing environment. This enables new ideas to be quickly and easily implemented and tested out. The computational experience thus obtained often results in new insights and developments, and helps in laying out the basic features of an algorithm. Such a language should permit programs to be written with relative ease, in the vernacular of applied mathematics. It serves as a medium for communicating algorithmic ideas precisely. The MPL language, see Dantzig et al., 1970, was specifically designed with this in mind. Other examples of suitable languages are Speakeasy, see Cohen and Pieper, 1976, and APL, see Gilman and Rose, 1976. When a collection of subroutines which carry out some of the basic operations of linear programming, for example, the main steps in the cycle of the revised simplex method, are also implemented, the usefulness of the language is further enhanced. We shall call such subroutines *modules*, and they can be thought of as a suitable extension of the language. From now on we shall refer to experimental implementations of LP algorithms developed in such an extended high level language as *level- 1 implementations*. They comprise the first level in the hierarchy of implementation and they can clearly suffer from some serious limitations. For example, the coding is often "quick and dirty", the routines are often only effective on toy problems, and they will not infrequently encounter numerical difficulties. Being able to work in a high

level language, no matter how convenient, does not circumvent a basic stumbling block, namely, routines which are numerically sound and efficient in running time and use of storage, are difficult to write.

When more emphasis is to be placed upon a numerically sound implementation which can be run on more realistic problems, we then come to implementations in the second level of the hierarchy (called *level-2 implementations*). Problems that arise from real world applications are usually sparse. For example, even relatively small models, say having 300 to 800 rows and 500 to 1200 columns, tend to have a density of about 0.2 to 0.4, see Greenberg, 1978. Thus efficient representations of date are needed which take sparsity into account, and the implementations must whenever possible be robust, flexible and transportable. They should be able to work with LP problems which are specified in standard MPS input format. There would again be the need to identify the components that are used to build LP routines at this level, to specify them clearly and carefully, to implement them as modules in a manner that makes them flexible and easy to use, and to have some standardization of the communicating data structure. Because of the above goals, it would be natural to implement these modules in Fortran, since it is now the accepted language of scientific computing for any sort of software intended for wide distribution. One can then draw upon the quite extensive experience in developing mathematical software described, for example, in Smith et al., 1974, Ford and Hague, 1974. Such a collection of modules would be useful both for research purposes and as a teaching aid for more advanced computational aspects of LP algorithms than those at the first level of the hierarchy, as described above. Our paper is

concerned with the development of a collection of modules to aid in producing level-2 implementations.

Finally, we come to *level- 3 implementations* which are designed primarily to solve user problems. The MINOS code of Murtagh and Saunders, 1978, written in Fortran, is an example, it being a library quality, user oriented, transportable code. Other widely used codes are the commercially available Mathematical Programming Systems like MPSX/370. These large scale MP Systems have extensive control and data management facilities and since they are usually tailored to the characteristics of a specific machine for maximum run time efficiency, many of the subroutines that carry out frequently repeated operations may be implemented in machine language. Such systems are expensive to use, and there is, of course, a premium to be paid in terms of flexibility and transportability, since they are designed for specific machines. Sometimes some of the high level routines are made available to the algorithm developer. (Figure 1 lists some of the algorithm oriented modules that are available in MPSX/370.)

An eventual goal of research into optimization algorithms is to develop good level-3 implementations. Developing level-1 and level-2 implementations represents the achievement of important intermediate goals. Distinctions between the three different sorts of implementations are, of course, not clear cut and are primarily a question of which goals are emphasized. Level-2 implementations can and should be used to solve practical problems, and level-3 implementations can and should be used to study the encoded algorithm, and by replacement of parts of the code, to develop and test out related algorithms. For example, MINOS is

primarily a level-3 implementation, but it could well be used for algorithm experimentation. XMP, see Marsden, 1980, is specifically addressed to both levels 2 and 3. It is important to note however, that the distinctions between implementations at the three levels we have discused above, are not primarily governed by the size of problems addressed. Thus a quality code for solving small nonsparse LP problems could be in the third level of the hierarchy rather than the first.

Modules can be developed at all three levels of the hierarchy, but especially at the first two levels, they are much more than subroutines in a well structured program. In addition to having a well-defined function and interface, they should be flexible and, whenever possible, context independent. We like to think of modules at the first two levels of the hierarchy as the primitives or basic operators of a language for implementing LP algorithms. At the third level modules tend more towards being well specified and designed subroutines in a structural programming sense, but here again the distinctions are not precise. For example, the modules listed in Figure 1 are flexible and useful for developing codes for algorithmic experimentation. Other useful collections of modules are, for example, given by Reid, 1976, Cline, 1977 and Land and Powell, 1973. As we have noted above, there is also a need at each level for a standardized communicating data structure, and this gets increasingly complex as we move down in the hierarchy. We have also mentioned the standard MPS input formats which level-2 and -3 implementations should be able to handle.

Figure 1. Some MPSX/370 modules.

SETLIST	(internal translation of variable)	PRICEP	(Pricing)
INVALUE	(match a list of names)	CHUZR1	(choose row)
GETVEC1	(moves column)	FTRANL1	
POSTMUL	(matrix-vector operations)	FTRANU1	(forward and
PREMUL		BTRANL1	(backward transforms)
		BTRANL1	
FIXVEC	(computes basics)	INVCTL1	(inversion)

Given the above context, we can now be more specific about the goals of this research effort, and about its limitations. As we have already stated, we have developed a small collection of modules designed to aid the development of level-2 implementations of LP algorithms, and to serve as a classroom tutorial on such implementations. We have drawn upon the work of many different workers in the field, for example, Saunders, 1977, Reid, 1976, Tomlin, 1975 and Greenberg 1978. Nothing that is particularly new in the way of techniques is suggested and ours is primarily a systematization and organization effort. Many of our routines are derived from MINOS, see Murtagh and Saunders, 1978. However, since we have made a great many modifications to suit our particular needs, responsibility for errors rests with us, and shortcomings of our routines should in no way reflect upon the source of the code.

We expect our modules to be of help to someone who is developing a level-2 implementation of an LP algorithm, particularly if it is based upon the Decomposition Principle. We do not however expect them to be used in a 'plug-in' fashion. Rather they provide a starting point for development. For tutorial purposes, the code is sufficiently readable to provide a

detailed illustration of implementation techniques.

3. Description of Modules

We now give an overview of our modules, and in particular, the considerations that guided our design. We do not however limit our discussion solely to the modules we have implemented since an aim of this paper is to give the reader a feel for some of the broader issues involved in a effort such as this one. We attempt, in our discussion, to strike a balance between describing what we have implemented and speculation about a more comprehensive collection. A much fuller description of our implementation can be found in the documentation (see Section 4).

Figure 2. Overview

1. PROBLEM ORIENTED MODULES
   PREADR, PREADC, PRDRHS, PREADB, PCHKST
2. ALGORITHM ORIENTED MODULES
   - 2.1 Data Structure Manipulation
     ADCONC, ADRNDX, ADINTF, ADUPKC, ADDELC
   - 2.2 Basic Simplex Modules
     MODRHS, FORMC, PRICE, CHUZR, UPBETA
   - 2.3 Sparse Linear Algebra Modules
     Interface to routines of Reid, 1976.

We have grouped our modules according to their function, and Figure 2 gives a summary of them. We have a naming convention that the first character indicates the main category to which the module belongs - problem oriented or algorithm oriented, the second character may indicate a subcategory, and the remaining characters indicate the module's function. In some cases however, the module names are so standard, that we have dispensed with the naming convention. (A third category - code oriented modules - could usefully be added though we have not done so

here. These provide aids to coding, e.g., routines to efficiently do inner products, and so on.)

3.1. Problem Oriented Modules

In order to solve an LP problem both conveniently and efficiently, a user requires more than just a well implemented LP algorithm. Problem oriented modules are designed to help provide the interface between the user and his LP matrix on the one hand, and the LP optimization routine on the other.

Interface features are, for example:

a) To read in the LP matrix specified in some standard input format and develop a suitable data structure representing it.

b) Verify information about the matrix and/or gather statistics about it.

c) Output solution found in some standard format.

d) Having set up the input matrix, by permutation of rows and columns, try to reorder it into a specific structure, e.g., block angular.

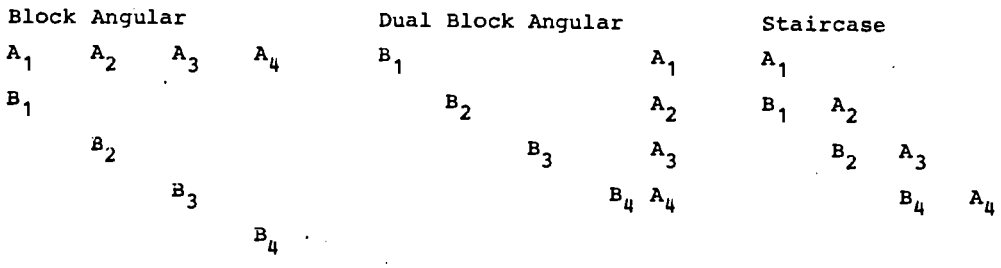
e) Modify portions of the initial problem, e.g., delete a set of rows.

We have concentrated upon a) and b) and the following factors have influenced our design:

i. We want to be able to handle practical problems of a reasonable size i.e., problems that are representative of real life applications, and these are often specified in standard MPS input format (see Appendix I).

ii. Such LP problems are usually sparse, and therefore they should be stored in some packed representation (see Appendix I).

iii. Furthermore, LP problems are often structured and we expect our modules to be used for implementing algorithms that take advantage of this structure. Typical examples of structured LP's are:



A routine designed to take advantage of special structure may have to keep different parts of the LP matrix, e.g., $\begin{bmatrix} A_i \\ B_i \end{bmatrix}$ in different packed data structures, perhaps with rows consecutively numbered. It would therefore not be appropriate to provide a general input routine which reads and packs a single matrix specified in MPS input format. Instead, using to a large extent the input routines of MINOS, we have developed a set of components from which a suitable input routine can be built.

Our modules, which we have designed to be very flexible, are as follows:

a) PREADR (Problem oriented READ Rows)

This module reads in the list of row names and row types from the ROWS Section of the matrix and optionally builds a hash table, see Brent, 1973, to speed up input of matrix elements. Extensive error checks are

provided.

b) PREADC (Problem oriented READ Columns)

This module reads in specified subset of columns from the COLUMNS Section of the *LP* matrix and builds a new packed data structure or extends a previously built one. Hashing can optionally be used to speed input. Again extensive error checks are provided, for example, upper and lower bounds on row indices can be set, to verify that the matrix is structured as expected.

c) PRDRHS (Problem oriented Read RHS)

This module reads in a specified right hand side vector from the RHS Section of the *LP* matrix into a packed data structure.

d) PREADB (Problem oriented READ Bounds)

Reads in a specified bounds vector from the BOUNDS Section of the *LP* input matrix. Lower bounds are set up in an array BL and upper bounds in an array BU. All variables are initially set to default lower and upper bounds and then reset as follows, if they are included in the bounds vector:

Field specifying type of bound	**Setting for BL**	**Setting for BU**
LO	bound value	unchanged
UP	unchanged	bound value
FX	bound value	bound value
FR	- PLINFY	+ PLINFY
PL	0	+ PLINFY
MI	- PLINFY	0

where PLINFY is a machine representation of infinity.

e) PCHKST (Problem oriented CHecK STatistics)

Checks bounds and reports statistics on the input matrix.

More extensive descriptions of the above modules are given in the documentation (see Section 4) and the testing programs of Chapter III of this documentation give an example of how the modules can be used.

3.2. Algorithm Oriented Modules

These provide some of the basic building blocks of LP algorithms, and we have gathered them into three groups as follows:

3.2.1. Data Structure Manipulation Modules

An LP algorithm will usually carry out numerous operations which modify and update its representation of data. For example, a decomposition algorithm will continuously add and delete columns from the packed data structure holding its master problem. Another example was mentioned earlier in Section 3.1, where we talked about the need to reindex rows in a packed data structure, and there are numerous other examples of this type. LP algorithms that exploit the special structure of the matrix often require complex strategies, for example, how many columns to add or purge from a data structure, how often to do this, and so on. By isolating basic operations on packed data structures, we can make a distinction between the task of devising a good strategy upon which the success of a particular algorithm often depends, and the task of implementing this strategy, which data structure manipulation modules can facilitate.

We have provided just a few basic operations of this type, and more can be added as the need arises:

a) ADCONC (Algorithm oriented Data str. manip. CONCatenate data structures)

Concatenates two packed data structures, and returns result in the first one.

b) ADRNDX (Algorithm oriented Data st. manip. ReiNDeX data structure)

Reindexes the rows in a packed data structure.

c) ADINTF (Algorithm oriented Data str. manip. INTerFace)

Converts a packed data structure into an element/row index/column index data structure ad thus provides an interface to routines that use the latter.

d) ADUPKC (Algorithm oriented Data str. maip. UnPacK Column)

Unpacks a specified column of a packed data structure.

e) ADDLC (Algorithm oriented Data sr. manip. DELete Column)

Deletes a column of a packed data structure and closes it up.

3.2.2. Basic Simplex Modules

Different algorithms for structured LP usually require a somewhat different version of the simplex algorithm. For example, in the Dantzig-Wolfe decomposition algorithm, a subproblem may be solved by the revised simplex method, but several intermediate solutions will usually be saved and passed back to the master problem. If the subproblem is

unbounded, the extreme ray solution that is found must again be passed back to the master. This requires a tailored version of the revised simplex algorithm. Implementing such an algorithm and algorithms of this type, is made a whole lot easier, by having at ones disposal the modules of this section.

In devising modules that help in implementing different versions of the revised simplex method, some conventions must be established about:

1. The canonical form in which the LP problem is set up.

2. The data structure that provides the communication between modules.

We have been motivated in our design by techniques used by Tomlin 1975, Saunders 1977, and others, and we have adopted the following conventions:

1. Computational Canonical Form

Suppose that the initial LP problem is

$$\textit{minimize} \qquad c^T x$$

$$\textit{subject} \quad Ax \begin{Bmatrix} \leq \\ = \\ \geq \end{Bmatrix} b$$

$$l \leq x \leq u$$

If the problem was specified in MPS input format, the type of constraint would be given by the ROWS Section and the bounds constraints can be identified as described in Section 3.1 d).

Transform the problem as follows:

$$\textit{minimize} \qquad c^T x$$

$$\textit{subject} \quad Iz + Ax = b$$

$$l \le x \le u$$

and

$0 \le z_i \le \infty$ if row i is a $\le$ row (nonnegative slack)

$-\infty \le z_i \le 0$ if row i is a $\ge$ row (nonpositive slack)

$0 \le z_i \le 0$ if row i is an = row (artificial)

Finally we have the computational canonical form:

$$\textit{minimize} \; -z_0$$

$$\textit{subject} \quad z_0 + c^T x = 0$$

$$Iz + Ax = b$$

$$l \le x \le u$$

$$-\infty \le z_0 \le +\infty$$

and z bounded as above.

We define

$$\bar{A} = \begin{bmatrix} I & c \\ & A \end{bmatrix}$$

We call x the *structural* variables and (z_0, z) the *logical* variables. Thus in the computational canonical form which we work with, a full identity matrix for the logical variables is assumed to be written at the start of A. The bounds on these logical variables are determined by the type of row, and no distinction is made between nonpositive, nonnegative slacks or artificial. They simply have different bounds that they must satisfy.

2. Communicating Data Structure

The data structure hat we use for communication between different modules is summarized in Figure 3. We have followed Tomlin 1975, Saunders 1977, and Ho 1974 in our naming conventions. The matrix is in computational canonical form and is packed as explained in Appendix I in arrays A, HA and HE. The integer variables LDA, LDHE, N and NHE give information about the data structure. KINBAS and PEG identify the state of each variable of the problem and a small extension of the simplex method is permitted in that variables can be temporarily pegged between their bounds. This idea is related to the superbasic variables of Murtagh and Saunders, 1978, but the latter used in a more powerful way, since an optimization is carried out in the subspace that they define. The use of pegged non-basic variables involves some straightforward extensions to the modules PRICE, CHUZR and UPBETA described below. PEG contains the current value of every variable in the problem, both logicals and structurals. Thus there is some redundancy of information stored; but this is not too great a penalty to pay at this level, given the added flexibility that PEG makes possible, for example, being able to start with a non-basic feasible solution which the user may have available to him, as often

happens in decomposition algorithms. Finally, the array JH keeps track of the basis, and the variables JXOUT = JH(JP) and JXIN keep track of the existing and incoming variables, respectively. IOBJ points to the objective row.

The modules we have implemented communicate through the above data structure. They carry out the main steps in the cycle of the simplex method, apart from the operations involving the basis matrix, which are discussed in the next section. There is, of course, a substantial overlap between our modules and those listed in Figure 1.

a) MODRHS (Algorithm oriented, basic simplex, MODify Right Hand Side)

Given the values of the nonbasic variables in PEG, this module forms the starting basic solution. It also returns a vector whose elements are useful for determining whether the level of rounding error is significant.

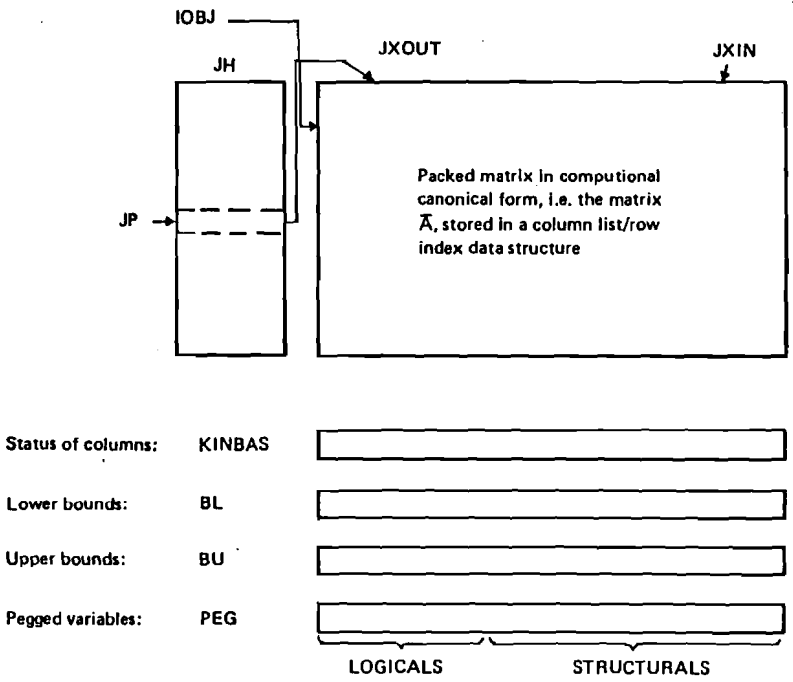
b) FORMC (Algorithm oriented, basic simplex, FORM Cost row)

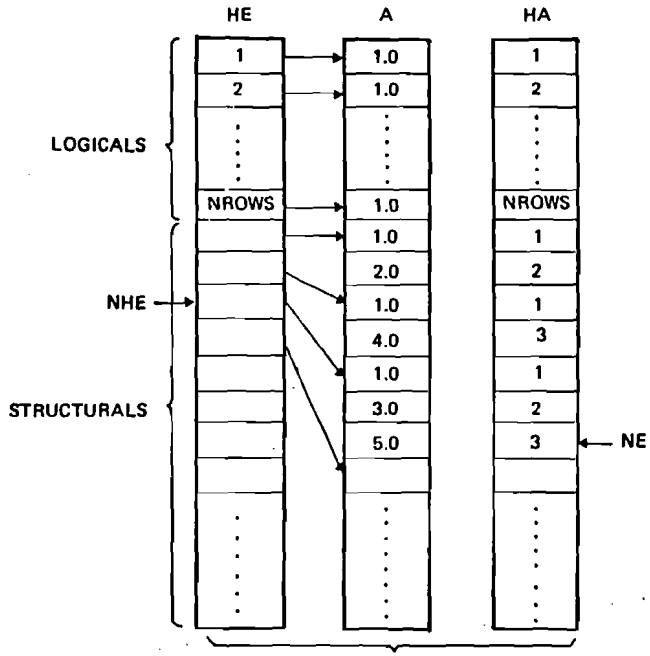
This module sets up the objective row vector c suitably, depending on whether the current solution is feasible or not. If feasible, then $c_1 = -1$ and $c_j = 0$ for $j \geq 2$, (see the computational canonical form of Section 3.2.2). If infeasible, then $c_i = 0$ if x_i is feasible, $c_i = -1$ if x_i violates its lower bound, and $c_i = +1$ if x_i violates its upper bound. The documentation (see also Section 4) justifies this in detail.

c) PRICE (Algorithm oriented, basic simplex, PRICE out columns)

Determine one or more variables as suitable candidates to enter the basis, i.e., use the vector of prices π to calculate the reduced cost of the nonbasic columns. Various options are provided including partial and

Figure 3. Data Structure for Simplex Modules





The LP matrix shown here is the one given in the Appendix transformed into 'computational cannonical form' and then packed

JH (I) Points to the I'th variable of the basis

KINBAS (J)
- = 0 if the J'th variable is at lower bound
- = 1 if the J'th variable is at upper bound
- = 2 if J'th variable is pegged between bounds
- = 3 if J'th variable is basic

JXIN points to column to enter basic(determined by PRICE)

JP points into JH and identifies which column JXOUT will exit from basic (determined by CHUZR)

A, HA, HE packed data structure, A and HA are of dimension LDA, and HE is of dimension LDHE

NE number of elements in Ā

NHE number of columns of Ā

multiple pricing.

d) CHUZR (Algorithm oriented, basic simplex, choose (CHUZ) Row)

Given the index of the incoming variable, this module determines which variable it replaces. There are two cases:

(i) basic variables are feasible. In this case the basic procedure is straightforward, but there are a number of special cases which make the implementation a little messy. a) The entering variable is the first to hit its bound. In this case the basis is unchanged. b) The entering variable can be increased indefinitely leading to an unbounded optimal solution. c) Ties in the choice of the exiting basic variable are found. In this case we use the two pass perturbation technique of Harris as implemented by Tomlin 1975.

(ii) Some basic variables are infeasible. In this case we use the method of Rarick, again as implemented by Tomlin 1975. For algorithmic details see also Greenberg 1978, and the documentation of Section 4.

e) UPBETA (Algorithm oriented, basic simplex, UPdate solution (BETA))

This module updates the basic solution and the driving arrays JH, KINBAS and PEG.

There are again many other modules that could be added to the collection. For example, if we wished to implement methods based upon the dual simplex method, we would need a version of CHUZR that worked with rows rather than columns. But the ones given above arise most commonly, and others can be added as the need arises.

3.2.3. Sparse Linear Algebra Modules

In the revised simplex method, the basis matrix is maintained and updated in some factored form, and used to transform columns of the $\mathcal{LP}$ matrix by the FTRAN operation as it is commonly called, (see Figure 1), and to compute the price vector by the BTRAN operation. Factorization of the basis that was based upon Gauss-Jordan elimination was the earliest method used, but now much more sophisticated techniques are available, see, e.g., Saunders 1976, Forrest and Tomlin 1972, Hellerman and Rarick, 1971, Reid 1976, and Cline 1977. At level-2 for which our modules are intended, the routines of Reid 1976 which employ LU factorization and Bartels-Golub updating are almost ideal, and we have done little more than provide an interface to them. The subroutines of Cline 1977 are also numerically stable, but they do not take sparsity into account and were therefore not suitable for our needs, and the bump and spike method of Hellerman and Rarick 1971, or the method of Saunders 1976, are more suited to level-3 implementations.

4. DOCUMENTATION

The documentation is organized into four chapters as follows:

Chapter 1: A discussion of each module under the headings

1. PURPOSE
2. USAGE
3. ALGORITHMIC & PROGRAMMING DETAILS

Each of the main categories of modules described in Section 3 above is in addition preceded by an introductory section, which provides back-

ground information. For example, the introductory section for Problem oriented modules of Section 3.1 discusses MPS input format and hashing.

Chapter 2: In order to make it possible to add to the collection and maintain some uniformity in the coding, we describe here some coding and documentation conventions that were used.

Chapter 3: For each major group of modules we provide a testing program and give its input and its output. The testing program on Problem Oriented modules gives a detailed illustration of how to construct a routine to read an MPS tape using the modules provided, and of the error checking that is made possible. The testing program on Data Structure Manipulation modules simply calls each one in turn. Finally the testing program for Basic Simplex and Sparse Linear Algebra modules shows in detail how to implement the cycle of the revised simplex method. These testing programs could also provide a useful starting point when coding an *LP* algorithm.

Chapter 4: A Fortran listing of each module in the collection.

The above four chapters of documentation and listings are written in machine readable form. They are available as a single file on a magnetic tape, which can be read and partitioned according to one's own needs. For further details write to the author at the following address:

IIASA
System and Decision Sciences Area
A-2361 Laxenburg, Austria

We should conclude on a note of caution. The effort described above is limited in scope, and we do not claim that our routines meet the stan-

dards of quality software and transportability as set out for example in Smith et al., 1974. Testing is still continuing, and the test programs of Chapter 3 give the current extent of testing to which the modules have been subjected. We believe however that we have met our goals as laid out in Sections 1 and 2, namely:

a) to provide some aids which serve as a starting point for developing level-2 implementations of *LP* algorithms. Indeed, we are currently using them in the implementation of an algorithm for two-stage stochastic programming with fixed recourse;

b) to provide a tutorial on implementation of *LP* algorithms.

5. Acknowledgement

The author is most grateful to Drs. M. Saunders, J. Tomlin, J. Reid and H. Greenberg who provided the foundation upon which rests much of this work.

REFERENCES

Brent, R.P. (1973), Reducing the retrieval time of Scatter Storage Techniques, Comm. A.C.M., 16, pp. 105-109.

Cline, A.K. (1977), Two Subroutine Packages for the Efficient Updating of Matrix Factorizations, University of Texas at Austin, Department of Computer Science Report TR-68, Austin, Texas.

Cohen, S. and S.C. Pieper (1976), The Speakeasy-3 Reference Manual, Level Lamda, Argonne National Laboratory, Report ANL-8000, Argonne, Illinois.

Dantzig et al. (1970), MPL-Mathematical Programming Language - Specification Manual, Report STAN-CS-70-187, Computer Science Dept., Stanford University.

Dantzig, G.B. and P. Wolfe (1961), The Decomposition Algorithm for Linear Programming, Econometica, 29, pp. 767-778.

Ford, B. and Hague, S.T. (1974), The Organization of Numerical Algorithms Libraries, In Proceedings of IMA Conference on Software for Numerical Mathematics, J. Evans, (Ed.), Academic Press, pp. 357-372.

Forrest, J.J.H. and Tomlin, J.A. (1972), Updating Triangular Factors of the Basis to Maintain Sparsity in the Product Form Simplex Method, Mathematical Programming, 2, pp. 263-278.

Gilman, L. and A.J. Rose (1976), APL An Interactive Approach, (Second edition, revised) Wiley.

Greenberg, H.. (1978), Pivot Selection Techniques, In Design and Implementation of Optimization Software, H. Greenberg (Ed.), NATO Advanced Studies Institute Series E. Applied Science, No. 28, Sijthoff and Noordhoff, pp. 1-26.

Hellerman, E. and Rarick, D. (1971), Reinversion with the Preassigned Pivot Procedure, Mathematical Programming, 1, p. 195-216.

Ho, J.K. (1974), Nested Decomposition of Large Scale Linear Programs with the Staircase Structure, Systems Optimization Laboratory Report SOL 74-4, Department of Operations Research, Stanford University.

Land, A.H. and S. Powell (1973), Fortran Codes for Mathematical Programming, Wiley.

Marsten, R.E. (1980), The Design of the XMP Linear Programming Library, Management Information Systems Report 80-2, University of Arizona, Tucson.

Murtagh, B.A. and M.A. Saunders (1978), Large Scale Linearly Constrained Optimization, Mathematical Programming, 14, pp. 41-72.

Nazareth, L. and R. J-B. Wets (1982), Algorithms for Stochastic Programs: the case of non-stochastic Tenders, IIASA Working Paper (Forthcoming).

Reid, J.K. (1976), Fortran Subroutines for handling Sparse Linear Programming Bases, A.E.R.E. Harwell Report R8269, Harwell, England.

Saunders, M.A. (1976), A Fast, Stable Implementation of the Simplex Method Using Bartels-Golub Updating, In Sparse Matrix Computations, Bunch and Rose (Eds.), Academic Press, pp. 213-226.

Saunders, M.A. (1977), MINOS-User's Manual, Systems Optimization Laboratory Report SOL 77-31, Department of Operations Research, Stanford University.

Smith, B.T., Boyle, J.M. and Cody W.J. (1974), The NATS Approach to Quality Software, in Proceedings of IMA Conference on Software for Numerical Mathematics, J. Evans (Ed.), Academic Press, pp. 393-405.

Tomlin, J.A. (1975), LPM1-User's Manual, Systems Optimization Laboratory, Department of Operations Research, Stanford University.

Appendix I

EXAMPLE OF MPS INPUT FORMAT AND PACKED MATRICES

LP

$$\min \quad x_1 + x_2 + x_3$$

$$\text{s.t.} \quad 2x_1 + 3x_3 \leqslant 10$$

$$4x_2 + 5x_3 \leqslant 20$$

$$100 \geqslant x_1 \geqslant 0, \quad x_2 \geqslant 0$$

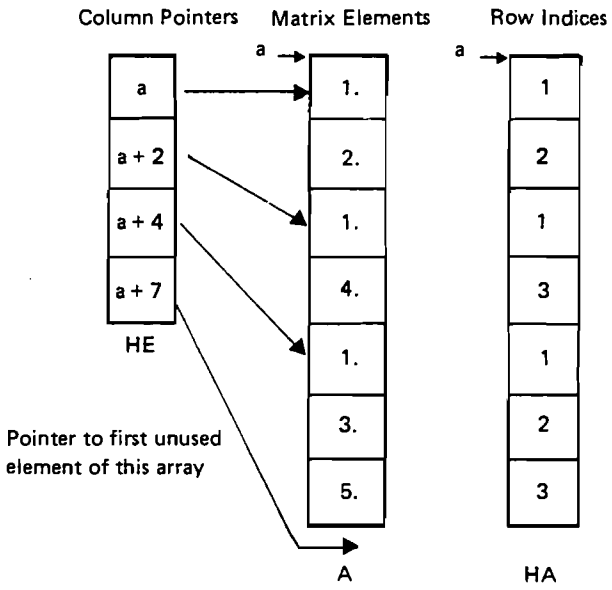
TABLEAU

Row names \ Column Names	CLM1	CLM2	CLM3		RTH
OBJ	1.	1.	1.		0
RWN1	2.	0	3.	$\leqslant$	10.
RWN2	0	4.	5.	$\leqslant$	20.

Sample MPS Input

```
NAME          LP
ROWS
 N            OBJ
 L            RWN1
 L            RWN2
COLUMNS
    CLM1      OBJ       1.0     RWN1      2.0
    CLM2      OBJ       1.0     RWN2      4.0
    CLM3      OBJ       1.0     RWN1      3.0
    CLM3                        RWN2      5.0
RHS
    RTH       RWN1      10.0
    RTH       RWN2      20.0
BOUNDS
 UP BVN       CLM1      100.
ENDATA
```

Packed Representation of Above Matrix, Excluding RHS (column list/row index data structure)



Thus the third column (called CLM3) starts at element a + 4 of array called 'Matrix Elements'. This column has three elements whose corresponding indices are given by the elements of the array called ' Row Indices'.