


# MAPPING EFFICIENT NUMERICAL METHODS TO THE SOLUTION OF MULTIPLE OBJECTIVE LINEAR PROGRAMS

THESIS

Michael A. Shields
Captain, USAF

AFIT/GOR/ENC/91M-1



DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY
Wright-Patterson Air Force Base, Ohio

AFIT/GOR/ENC/91M-1


# MAPPING EFFICIENT NUMERICAL METHODS TO THE SOLUTION OF MULTIPLE OBJECTIVE LINEAR PROGRAMS

THESIS

Michael A. Shields
Captain, USAF

AFIT/GOR/ENC/91M-1

Approved for public release; distribution unlimited

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



| 1. AGENCY USE ONLY *(Leave blank)* | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | 1990 March | MS Thesis |

**4. TITLE AND SUBTITLE**

Mapping Efficient Numerical Methods To The Solution Of Multiple Objective linear Programs

**5. FUNDING NUMBERS**

**6. AUTHOR(S)**

Michael A. Shields, B.S. , Capt, USAF

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

School Of Engineering
Air Force Institute Of Technology
Wright-Patterson AFB, Oh. 45433-6583

**8. PERFORMING ORGANIZATION REPORT NUMBER**

AFIT/GOR/ENC/91M-1

**9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

**11. SUPPLEMENTARY NOTES**

**12a. DISTRIBUTION / AVAILABILITY STATEMENT**

Approved for public release; distribution unlimited

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*

This investigation was initiated to increase the speed, accuracy and capacity of m-simplex algorithms for solving multiple objective linear programming problems. Specifically, improvements were sought through the application of general numerical techniques.

It soon became apparent that the m-simplex algorithm, like the simplex algorithm, is heavily dependent upon the technology of solving related systems of linear equations. The numerical arguments for the application of LU triangular matrix factorization techniques to simplex computations are well know. Of special significance to m-simplex performance is the case of rank- k updates to basis factorizations. A stable and efficient LU approach to the rank-k update problem is discussed. Accompanying software supports the solution of linear and transposed linear systems formed from rank-k updated LU factorizations. The software is constructed using BLAS and Linpack libraries.

| 14. SUBJECT TERMS | 15. NUMBER OF PAGES |
|---|---|
| | 67 |
| | 16. PRICE CODE |

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | UL |

AFIT/GOR/ENC/91M-1

THESIS APPROVAL

STUDENT: Michael A. Shields CLASS: GOR-91M

THESIS TITLE: MAPPING EFFICIENT NUMERICAL METHODS TO THE SOLUTION OF MULTIPLE OBJECTIVE LINEAR PROGRAMS

DEFENSE DATE: 1 March 1991

| COMMITTEE: | NAME/DEPARTMENT | SIGNATURE |
|---|---|---|
| (Advisor)/co-Advisor *(circle appropriate role)* | Dr. Bruce Suter/ENG | Bruce W. Suter |
| co-Advisor/(ENS Representative) *(circle appropriate role)* | Dr. Yupo Chan/ENS | Yupo Chan |
| Reader | Dr. Mark E. Oxley/ENC | Mark E. Oxley |

AFIT/GOR/ENC/91M-1

# MAPPING EFFICIENT NUMERICAL METHODS TO THE SOLUTION OF MULTIPLE OBJECTIVE LINEAR PROGRAMS

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science Operations Research and Applied Mathematics

Michael A. Shields, B.S.
Captain, USAF

March, 1991

## *Preface*

In performing the investigation and writing this thesis I have benefitted greatly from the help of others. I am indebted to my Thesis Advisor, Dr. Bruce Suter, for his patience and substantial assistance throughout the entire evolution of this project. I am also indebted to my other committee members, Dr. Chan and Dr. Oxley for their supportive attention and insightful comments. I must mention the significant influence on this project of Gill, Murray and Wright's textbook,*Numerical Linear Algebra and Optimization.* Dr. Suter should not complain that I have depleted his library of yet another text ... I now have my own copy. I also wish to show appreciation to Dr. Ralph Steuer for his kind assistance in regards to his extensive software package, ADBASE. Finally, I wish to thank my wife Tina and my parents, without whose understanding and support this project would not yet be completed.

Michael A. Shields

# *Table of Contents*

## *List of Figures*

## *Abstract*

This investigation was initiated to increase the speed, accuracy and capacity of m-simplex algorithms for solving multiple objective linear programming problems. Specifically, improvements were sought through the application of general numerical techniques. Objectives included:

1. Decomposing the m-simplex algorithm into functional elements which may be independently improved.
2. Focusing upon the implementation of computationally intensive portions of the algorithm.
3. Utilizing well-known modular library routines.
4. Using ADBASE as a vehicle for integrating and demonstrating improved algorithms.

It soon became apparent that the m-simplex algorithm, like the simplex algorithm, is heavily dependent upon the technology of solving related systems of linear equations. Gill, Murray and Wright have emphasized the importance of rank-1 updates to simplex performance (6). This research extends their emphasis to the case of rank-$k$ updates and m-simplex performance.

The numerical arguments for application of $LU$ triangular matrix factorization techniques to simplex computations are well known. A stable and efficient $LU$ approach to the rank-$k$ update problem is discussed. Accompanying software supports the solution of linear and transposed linear systems formed from rank-$k$ updated $LU$ factorizations. The design is constructed using BLAS and Linpack libraries. At this time, the software has not been integrated into ADBASE.

# MAPPING EFFICIENT NUMERICAL METHODS TO THE SOLUTION OF MULTIPLE OBJECTIVE LINEAR PROGRAMS

## *I. Introduction*

### *1.1 Multiple Criteria Optimization*

Multiple Criteria Optimization (MCO) techniques help decision makers to identify efficient uses for their resources. When multiple criteria are applied, the idea of 'best' suffers from potential ambiguity. Despite the presence of ambiguous cases, many choices may exist where one decision alternative is clearly better than another. We say a solution 'dominates' another when it is preferred according to *all* criteria. Dominating solutions which cannot, themselves, be dominated are said to be Pareto optimal, or *efficient*. 'Best' solutions are efficient solutions, but the converse is not always the case. There are implied trade-offs in selecting between *efficient* solutions. The region formed by the set of efficient points is known as the efficient frontier.

For example, suppose a software engineer wishes to minimize his algorithm storage requirements. Simultaneously, he wishes to arrange computations in a manner that minimizes floating point operations (FLOPS). These are two competing criteria. MCO techniques may be applied to identify *efficient* alternatives. Suppose out of four hundred proposals, three *efficient* designs are identified. The first proposal minimizes FLOPS by storing intermediate values in memory for convenient reference. The second reduces storage requirements by recomputing intermediate values as they are needed. The third stores some intermediate values and recomputes others. Depending upon the marginal costs of storage requirements and FLOP

counts, one of the *efficient* alternatives will emerge as 'best'. The software engineer, applying his expert knowledge of the computing environment may then make an informed decision by considering only three alternatives.

### *1.2 Multiple Objective Linear Programs (MOLP)s*

MOLPs solve a very restricted set of MCO problems. A MOLP identifies *efficient* solutions for a set of linear criteria evaluated over a linearly constrained feasible region. All such problems may be easily transformed into a standard form. For a standard Form MOLP, we seek a vector $x \in \Re^n$ to

minimize:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_r \end{bmatrix} = \begin{bmatrix} c_{11}x_1 + c_{12}x_2 + \cdots + c_{1n}x_n \\ c_{21}x_1 + c_{22}x_2 + \cdots + c_{2n}x_n \\ \vdots \quad \vdots \quad \ddots \quad \vdots \\ c_{r1}x_1 + c_{r2}x_2 + \cdots + c_{rn}x_n \end{bmatrix}$$

subject to:

$$\begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \quad \vdots \quad \ddots \quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

$$x_i \geq 0 \qquad \text{for each} \quad i = 1, 2, \ldots, n.$$

According to standard form:

1. Each maximization criteria is written as minimization criteria;

$$\max y_k = f_k(x) \quad \text{becomes} \quad \min y_k = -f_k(x).$$

2. Each negative resource level, $b_i$ is written as a positive level by applying a negative scalar to the entire $i^{th}$ linear constraint;

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n = b_i$$

becomes

$$-a_{i1}x_1 - a_{i2}x_2 - \cdots - a_{in}x_n = -b_i.$$

3. Each inequality constraint is written as an equality constraint by generating an independent variable, $S_i$, to account for slack or surplus relations;

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \leq b_i$$

becomes

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n + S_i = b_i.$$

Any MOLP in standard form may be compactly written using matrix notation as:

$$\begin{aligned} \text{minimize:} \quad & y = Cx \\ \text{subject to:} \quad & Ax = b \\ & x_i \geq 0 \qquad \text{for } each i = 1, 2, \ldots n. \end{aligned}$$

In the MOLP literature, vector optima are defined as Pareto optimal, non-inferior, admissible and non-dominated solution sets (17:22). A clear and formal development of Pareto preference structures and optimization theory is available from Yu in *Multiple-Criteria Decision Making: Concepts, Techniques, and Extensions* (17:10,21-53).

The form of a MOLP is reminiscent of that for a linear program (LP), only $C$ is a matrix instead of a vector. According to MCO theory, minimization will generate a region of feasible points which is denoted as the *efficient* frontier. For a MOLP,

this frontier is contiguously located on the surface of the feasible region and is simply connected. It is completely specified by a finite subset of *efficient* vertex solutions, together with any associated emanating unbounded *efficient* edges.

### *1.3 Solution with Simplex and M-Simplex Methods*

The simplex method is a practical technique for solving LPs. The simplex algorithm searches for optimal solutions by *moving* between vertex solutions. Suppose the standard form is expressed in expanded matrix notation:

minimize:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_r \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{r1} & c_{r2} & \cdots & c_{rn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_r \end{bmatrix}$$

subject to:

$$\begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} x_1 + \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} x_2 + \cdots + \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix} x_n = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

$$x_i \geq 0 \qquad \text{for each } i = 1, 2, \ldots, n.$$

Vertices and unbounded edges are defined by forming a basic vector set, $\mathcal{B}$, from independent columns of the constraint matrix, $A$. Every formation of $\mathcal{B}$ which produces non-negative coordinates (a basic feasible solution, $x$) for $b$ is associated with a vertex solution. Movement between vertex solutions clearly involves updating a basis to reflect the formation of a new vertex solution. If movement can no longer improve the solution, it is determined to be optimal. The *efficient* vertex solutions of a MOLP

may be determined according to an approach which is a natural generalization of the simplex method.

First, consider an LP experiencing degeneracy at optimality; moving to a degenerate vertex does not improve the current solution. However, for the simplex method to confirm 'optimality', all vertices must be explored until it is determined that only degenerate paths remain. This determination is a sufficient condition of optimality. The exploration also identifies *every* optimal vertex solution. Consider the following example taken from problem ATEST 1010 in Part II of the *ADBASE Operating Manual* (15):

$$\text{maximize: } y = \quad x_2$$

$$\begin{array}{lrrrl} \text{subject to:} & x_1 & & \leq & 4 \\ & & x_2 & \leq & 3 \\ & -x_1 & +x_3 & \leq & 0 \\ & x_1 & +x_3 & \leq & 6 \\ & x_1 & & \geq & 2 \end{array}$$

$$x_1, x_2, x_3 \geq 0$$

Optimality is *not* necessarily the property of a unique vertex solution (see Figure 1.1). For this problem, four additional slack variables and one surplus variable must be included to write the equation in standard form. There are five optimal vertex solutions given by the superscripted solution vectors, $x^1, x^2, x^3, x^4$ and $x^5$. Notice that when evaluated by the criterion function,

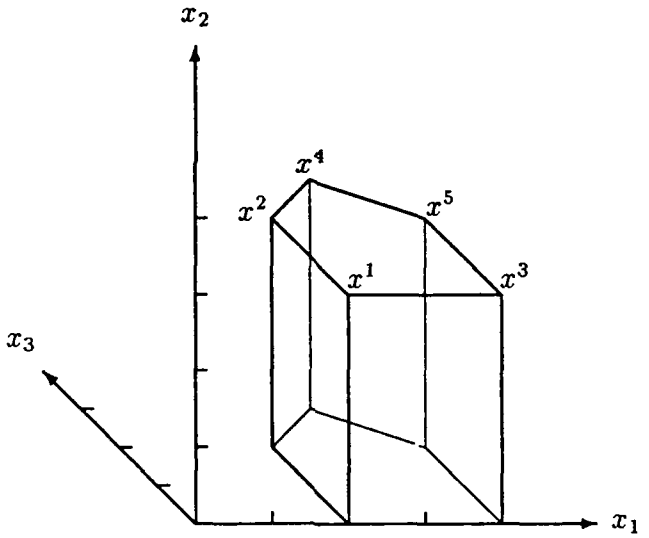
$$y = f(x_1, x_2, x_3, \ldots, x_8) = x_2$$

that

$$y^1 = y^2 = y^3 = y^4 = y^5 = 3.$$

A 'cycling prevention rule' is essential to determine that the exploration of degenerate paths is complete and to ensure the algorithm will terminate.

Figure 1.1. An LP with 5 Optimal (*Efficient*) Vertex Solutions




*ATEST1 Example 1010 in Part II of ADBASE Operating Manual (15).*

To generalize the simplex to an m-simplex method, stretch the concepts of 'optimality' and 'degeneracy' to additionally encompass the idea of *efficiency*. With the m-simplex method, movement is taken to dominating vertex solutions. Remember that when multiple criteria are involved, only 'dominating' alternatives are guarenteed to improve the current solution. If movement cannot be made to a 'dominating' alternative, the current vertex solution is determined to be *efficient*. From an *efficient* vertex, a situation develops which is similar to that of a LP experiencing degeneracy at optimality. All alternate *efficient* vertex solutions now need to be explored. For each *efficient* vertex solution, all unexplored, adjacent *efficient* feasible vertices are detected and their associated bases encoded. The coded bases form a spanning tree whose traversal specifies the order of future movement between vertex solutions. Tree traversal ensures each and every *efficient* vertex solution is visited once, and that any *efficient* emanating edges will be detected. In the presence of degeneracy, as in the LP case, many distinct vertex solutions may be evaluated as

equivalent according to criteria measures. Look again at the five *efficient* points illustrated in Figure 1.1. The necessary record keeping and encoding of the tree, by definition, prevent cycling. When all detected efficient vertex solutions have been visited, the algorithm terminates. Termination signals the identification of the *entire* set of *efficient* vertex solutions and unbounded edges.

### *1.4 Simplex Method and Changing Bases*

When applying the simplex method to an LP, *movement* between vertex solutions is associated with an exchange of basic and non-basic column vectors. Since all such movement is between immediately adjacent vertex solutions, each simplex iteration is equivalent to a rank-1 linear update of the basis vectors. When applying the m-simplex method to a MOLP, movement between *efficient* vertex solutions is associated with an exchange between $n$ basic and $n$ non-basic column vectors. This is equivalent to performing a rank-$n$ linear update of the current basis vectors. For the m-simplex method, movement is not necessarily between immediately adjacent vertex solutions, but such that a path may be constructed by tracing a succession of immediately adjacent, *efficient* vertex solutions. Such a path may be constructed from a breadth-first traversal of the spanning tree formed according to the m-simplex algorithm. The $n$ basic vectors which are updated in an m-simplex movement reflects the number of levels which must be traversed in order to find a common ancestor between subsequent *efficient* bases (4:138).

### *1.5 ADBASE*

Steuer has implemented an m-simplex algorithm within ADBASE (16). He provides a terse description of the algorithm (denoted as Phase III) in section 7.1 of the *ADBASE Operating Manual*:

> Phase III operates by pursuing each efficient edge emanating from each efficient extreme point. This involves a bookkeeping system to keep track

> of where ADBASE has been and where ADBASE has yet to go in its search for all efficient extreme points and unbounded edges. (15)

The list structures he implements for bookkeeping support what amounts to an implicit breadth-first tree search. (2:182–187).

### *1.6 Solution with Non-Simplex Methods*

Identifying *efficient* adjacent vertex solutions and *efficient* unbounded emanating edges appears to be a problem which prescribes the simplex approach. However, there are other practical ways of exploring the *efficient* frontier. One such approach is somewhat empirical and relies upon the systematic application of explicit weighting vectors. Whenever weights are established which relate criterion measures, a MOLP will collapse to a regular LP formulation where the multiple criteria may be replaced by a single vector expression. To exploit this property, hypothetical sets of standardized weights may be systematically applied and the resulting LPs solved for optimal points. Each weighted problem identifies a mapping to a point on the *efficient* frontier. To explore remaining gaps or extreme areas, additional weights may be selected and the associated LPs solved to provide additional mappings. This 'shot-gun' approach does not guarantee finding all of the extreme boundary of an *efficient* frontier, but it can be done without application of the simplex and m-simplex methods. This research effort will focus upon the implementation of efficient numerical methods for the solution of MOLPs according to the m-simplex algorithm.

# *II. Numerical Approach*

The main limiting assumption treated by this investigation concerns the numerical characteristics of m-simplex implementations. Achievement of algorithmic stability is central to the discussion. Over the years, several numerical developments have supported extensive and powerful revisions to simplex implementations. It seems natural these revisions should benefit m-simplex implementations, as well.

## *2.1 Simplex Revisions*

Katta Murty, in his 1983 text *Linear Programming*, reported that;

> At present, the revised simplex method is still the only approach that has proved to be computationally efficient and robust in practice (12:231).

For a numerically stable form that can preserve sparsity, he recommended the version which maintains a matrix $B$, formed from the basic vectors $\mathcal{B}$, in a lower - upper ($LU$) triangular matrix factorization, $B = LU$. He reasoned that it provides much better accuracy than either the explicit inverse or product form of the inverse. Bazaraa, Jarvis and Sherali, in their 1990 text declared the explicit and product forms of the revised simplex method to be

> ...computationally somewhat obsolete because it is based on a complete Gauss-Jordan elimination technique for solving systems of equations. A more popular technique used by most modern computer packages is the *LU factorization method*, which is based on the more efficient Gaussian triangularization strategy. ...it is accurate and numerically stable (round - off errors are controlled and do not tend to accumulate). (1:199–200)

## *2.2 Basis Formations*

As the previous paragraph indicates, revised simplex implementations are frequently classified according to the form in which the basic set is held and how solu-

tions are updated. When an explicit inverse is maintained for the current basis, linear systems involved in the revised simplex method may be solved by straight forward matrix multiplication. Updating the basis (inverse) may be directly accomplished by applying a special form of the Sherwin-Morrison formula adapted for column replacement to construct 'simplex-pivot' matrix operators. Applying a simplex-pivot amounts to the direct application of Gauss-Jordan elimination *without* stabilization from partial pivoting. Gill, Murray and Wright point out, in their 1990 textbook:

> Broadly speaking, numerical difficulties arise with the Sherman-Morrison formula because the quality of each updated inverse is affected by the condition of all previous matrices. ... We repeat that, with the Sherman-Morrison formula, the lingering effects of former ill-conditioning may damage all subsequent updated inverses. ... In contrast, standard techniques for updating $LU$ and $QR$ factorizations can (and do) "recover" from an initial ill-conditioned [matrix $B$.] ... [C]losely related systems of equations should be solved numerically by updating matrix factorizations. (6:149–150)

### *2.3 Avoiding Explicit Inverses*

Numerical analysts appear to agree that the computation and storage of inverse matrices is unnecessary and may be avoided. According to Golub:

> ... [W]hen a matrix inverse is encountered in a formula, we must think in terms of solving equations rather than in terms of explicit inverse formation (7:121).

Rice notes:

> In particular, any matrix expression applied to a vector can be evaluated efficiently without inverting any matrices, no matter how many inverse matrices there are in the expression. ... If you are not actually examining the elements of an inverse matrix, then it is very unlikely that you should be computing the inverse. (13:23)

### *2.4 Solving $Ax = b$*

Gill, Murray and Wright dispell the notion that a solution to a rank-$m$ linear equation " ...obtained with the inverse is somehow 'better','more accurate', or 'more elegant' (6:100)." Specifically, the number of operations to obtain a solution, given the inverse of a rank $m$ system, is of $O(m^2)$, the same as for the solution, given an $LU$ or $QR$ factorization. They analyzed ill-conditioned matrices using the singular value decomposition, $A = U\Sigma V^T$ (7:70-74). Given $A$ has distinct singular values, they find that accurate solutions are obtained with the inverse *only* when the right-hand side of the linear equation is close to a multiple of the column of $U$ associated with the largest singular value of $A$. In this case, $b$ is said to reflect the condition of $A$ and the inverse computes a 'large' solution. 'Large' solutions have the best chance of being accurate and producing a small residual. Solutions are poor when aligned with the column associated with the smallest singular value. Out of numerical consideration, they find that even when an inverse is formed as accurately as possible, its explicit use should be avoided. (6:101–104)

### *2.5 Adaptations*

The original version of ADBASE supports an m-simplex algorithm with a generalized revised simplex implementation that carries an explicit form of the inverse. To limit catastrophic errors resulting from well-known instabilities, a filter sets very small basis entries to zero. The threshold for small pivot elements is (TPIV = 1.0d-6) and the threshold for large problem coefficients is (COEFMX = 5.0d5). To retard the accumulation of roundoff errors, all computations are performed in double precision.

Effectively, the ADBASE algorithm operates at single precision. Many of the extra digits are used to contain the accumulation of remaining round-off errors. Suppose the pivoting was stabilized by maintaining the basis in the form of an LU factorization. Digits are then free space for greater operating precision or may be eliminated from storage. The implementation of a stable algorithm according

to efficient numerical methods is key to the practical extension of the m-simplex algorithm.

*2.5.1 Ease of Use* The difference which might become apparent to a user of a numerically improved m-simplex implementation would be that execution time might be shorter, or that the algorithm might become practical for use on a larger class of problems. The ability to compute an inverse might be restricted or possibly eliminated from the code. Inverses are not necessary to the performance of m-simplex algorithms.

Useful numerical information, such as the condition number, or some other measures of merit might provide indicators concerning the accuracy of feasible basic solutions.

## *2.6 Research Objective*

This investigation was initiated to increase the speed and capacity of m-simplex algorithms through implementation of efficient numerical methods. The research objectives are as follows:

1. Decompose the m-simplex algorithm into manageable elements which may be independently improved.
2. Identify computationally intensive areas.
3. Identify utilizations for general numerical methods, basic linear algebra subroutines (BLAS) and other higher level library modules.
4. Modify ADBASE accordingly.
5. Compare performance of resulting software with the performance of the original package. Potentials for comparison include issues of stability, speed, accuracy and maintainability.

# *III. Methodology*

## *3.1 Engineering Software*

A defining character of any implementation is the way a problem is broken down and solved. Structured, modular approaches to programming speed the development of reliable software, increase programmer productivity and improve the clarity and maintainability of the final product (10:92). Software which is easy to understand can be readily integrated into larger systems.

Modularity implies a minimum of complexity in connecting program units. Further, it suggests that all interaction between program units is controlled by the explicit use of well-defined, formal structures. In a modular environment, the details within a module may be hidden from all other modules and be independently and incrementally refined. Well-designed libraries are instrumental in establishing modular environments.

An excellent example of modular implementations is provided by the Linpack library (3). Linpack routines are constructed with level-1 BLAS routines. As BLAS routines are refined, client routines, such as Linpack, are automatically benefitted *without* modification. Mathematical programs which invoke routines such as Linpack likewise benefit by the association.

The m-simplex is a mathematical program whose function may be decomposed into simpler, more manageable elements. Some of these program elements are naturally expressible in terms of established library routines. Other elements demand the design and development of additional library capabilities. For example, the m-simplex algorithm requires an efficient capability for solving linear systems altered by rank-$k$ column replacements. A desirable addition to a library collection might then include general routines for solving related systems of equations. Any applications exploiting such routines would then benefit from library refinements.

Library routines built from low-level *standard* libraries enjoy additional benefits from portablity across machine platforms. For example, programs employing BLAS, when *ported* to more capable platforms, can fully benefit from the additional power of the new environment, without requiring any alteration of program source. The reason is that native BLAS implementations efficiently utilize available machine capabilites.

Top-down design and modular implementation achieves clarity, flexibility and efficiency. Esoteric numerical refinements may be accumulated in library modules whose implementations may be hidden from application programmers. Further, such library development allows the natural and terse statement of general, machine independent algorithms by researchers.

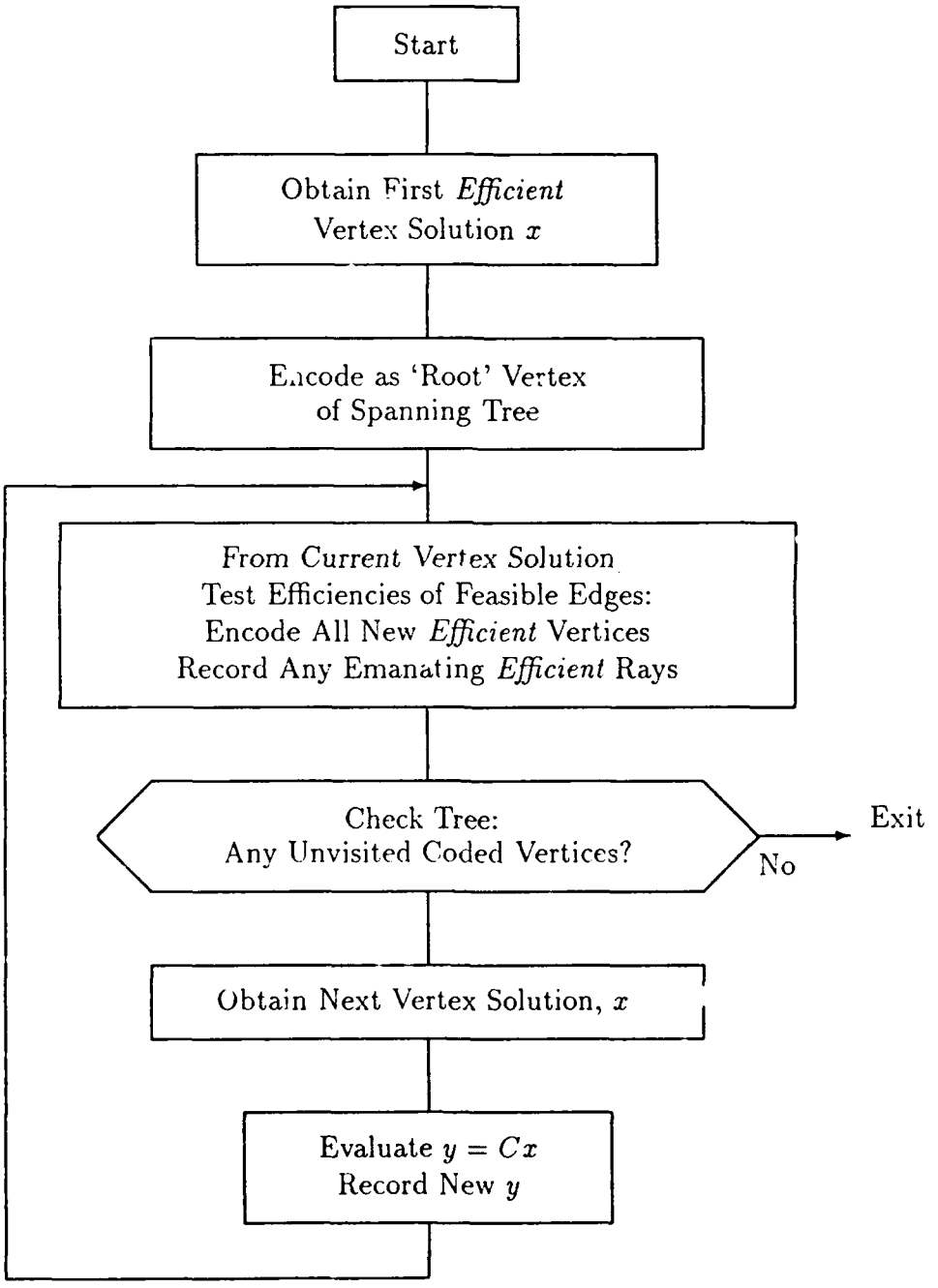
### *3.2 Defining the M-Simplex Algorithm*

The essential processing of an m-simplex algorithm is outlined in Figure 3.1. This breakdown reduces the algorithm into several components which may be encapsulated and developed in separate routines. The following description describes m-simplex activity according to the flow diagram.

Obtain a First *Efficient* Vertex Solution. There are several methods of securing an initial *efficient* vertex solution. An intuitive approach is to collapse the MOLP into an LP by selecting non-zero weights for each criteria function. As mentioned before, an optimal LP solution to such a problem is also an *efficient* vertex solution. Steuer implements additional techniques in ADBASE which incorporate a lexicographic approach to criteria optimization. This approach is more capable of finding Pareto optimal vertex solutions in the locality of unboundedness. Steuer also includes additional options which allow the detection of *efficiency*, if it occurs, before optimality is realized. (15)

Encode the first *efficient* vertex solution as the 'root' of a spanning tree. By the time of termination, this spanning tree will grow to accommodate the coded

Figure 3.1. Vector Optimization Algorithm



vertices of all *efficient* vertex solutions. Each vertex is uniquely identified by the formation of a basic set, $\mathcal{B}$, from the column vectors of $A$. The matrix $D$, formed from $\mathcal{B}$, is used to produce the vertex solution. To encode the vertex, the subscripts of $\mathcal{B}$ are stored.

Given an initial *efficient* solution, the algorithm enters a four step search loop.

1. From the current vertex solution, examine all adjacent feasible edges for efficiency. A subproblem routine identifies all *efficient* adjacent vertices and updates the spanning tree to include any new additions. Any unbounded *efficient* edges emanating from the current vertex solution are recorded as output.

   Subproblem routines are usually formulated as LPs and feasibility tests. Both are dependent upon the solutions of linear and transposed linear systems of equations involving the current basis matrix. Because of this, the subproblem can be encapsulated in its own module and its operating details hidden.

2. Check the tree. If all coded vertices stored on the spanning tree have been visited, then exit the search loop: The m-simplex search is complete.

3. Use the spanning tree to identify an *efficient* vertex which has not yet been visited. Identify the vectors which will enter the basic set and the vectors which will leave.

4. Update $\mathcal{B}$ and solve for the vertex solution $x$. Evaluate $x$ in terms of the criteria and identify a vector in the criteria space: $y = Cx$. If $y$ is distinct from previous evaluations, then record $y$ as an efficient and extreme solution of the MOLP. Return to the first step in the search loop.

### *3.3 Updating Vertex Solutions*

The major computational effort for the simplex method involves the solution of related systems of linear equations (6:365). This is also the case for the m-simplex method as described above. It is, perhaps, more important since there are many

more related systems to solve. The simplex case need only be concerned with rank-1 changes.

### *3.4 Sherman-Morrison Formula.*

*3.4.1 Rank-1 Updating.* Suppose the $j^{th}$ column of a square, invertible, $m$ by $m$ matrix $B$ is replaced. The change may be expressed by the following procedure:

Let $v = e_j$, the $j^{th}$ column vector of the identity matrix, $I$, and let $d$ be the vector difference by which the $j^{th}$ column vector of $B$ is changed by the replacement, then:

$$B_{new} = B_{old} + dv^T$$

The solution of related systems involving $B_{new}$ when given $B_{old}^{-1}$ is practicable through application of the Sherman-Morrison Formula (11:70):

$$B_{new}^{-1} = B_{old}^{-1} + \alpha(B_{old}^{-1}d)(v^T B_{old}^{-1})$$

where

$$\alpha = \frac{1}{1 - v^T B_{old}^{-1} d}$$

This yields the form of a 'simplex-pivot' operator, $T$, which may appear familiar to analysts who have hand-solved revised simplex tableaus. Let $w = B_{old}^{-1}d$ and $T = (I + \alpha w v^T)$, then:

$$B_{new}^{-1} = T B_{old}^{-1}$$

*3.4.2 Rank-k Updating.* Generalizing to the rank-$k$ update of $k$ changed vectors implies:

$$B_{new}^{-1} = T_k T_{k-1} \cdots T_1 B_{old}^{-1}$$

where $T_1, T_2, \ldots, T_k$ are simplex-pivot operators. Note that the rank-$k$ update may be conveniently held in product form and traces a path of rank-1 updates. These

rank-1 updates are reminiscent of a string of Gaussian elimination operators (without the benefit of partial pivoting).

Generalizing for a direct update implies a more complicated relation according to the Sherman-Morrison-Woodbury formula which requires the inversion of a rank-$k$ matrix (7:51).

### *3.5 Stable Update Formula*

The Sherman-Morrison Formulas are not numerically stable algorithms. We know that using the inverse to numerically solve linear systems is generally not necessary or even advisable. Efficient solutions usually involves factorizations such as the $LU$ decomposition.

*3.5.1 Rank-1 Updating.* Stable and efficient rank-1 update techniques applied directly to $LU$ factorizations have become popular in simplex applications. Two techniques for the LU factorization are presented and illustrated with examples by Gill, Murray and Wright in Chapter 4 of *Numerical Linear Algebra and Optimization,* (Vol. I) (6:139-150).

*3.5.2 Rank-k Updating.* This subsection discusses the generalization to rank-$k$ updates. To determine a rank-$k$ update of $B$, let $B_{leave}$ be the submatrix formed from the column vectors of $B$ which are to be replaced (updated). Let $B_{enter}$ be the submatrix formed from the $k$ entering column vectors which are replacing the $k$ vectors leaving $B$. Compute the matrix difference, D, resulting from the vector replacements:

$$D = B_{enter} - B_{leave}$$

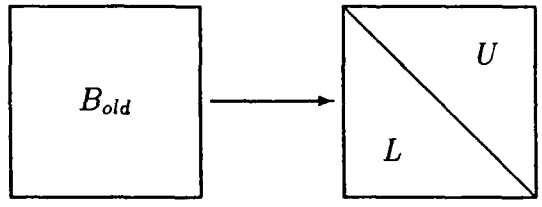
Let the 'change' subscripting denote the positions in $B$ where columns will be replaced. Let $E_{change}$ be a submatrix formed from the column vectors of an identity matrix subscripted by the change array. Then, letting $V = E_{change}$, column replace-

ment is described by the following matrix algebra:

$$B_{new} = B_{old} + DV^T$$

*3.5.3 Explicit LU.* By following the example of Gill, Murray and Wright, the following rank-$k$ update technique may be constructed from the explicit form of the $LU$ factorization. Suppose that we have the factorization $B_{old} = LU$. The two triangular factors may be formed in the storage provided for the original formation of $B_{old}$ if it is understood that the unit diagonal of $L$ is overwritten by the diagonal of $U$ (Figure 3.2):

Figure 3.2. Original Decomposition of $B_{old}$




It follows from the Sherman-Morrison update formula that,

$$B_{new} = L(U + WV^T)$$

where the matrix $W$ is solved according to the lower triangle system:

$$LW = D$$

Updating the factor $U$ to reflect the changes incurred in the formation of $B_{new}$ (Figure 3.3):

$$U^c = U + WV^T$$

where $U^c$ is an upper triangular matrix with 'spikes' in the columns indexed by

'change'. A spike in an upper triangular matrix occurs when a column has a non-zero value below the main diagonal. Direct elimination of the 'spikes' destroys the structure that can provide expedient solutions for updated linear systems. Therefore, to preserve useful structure, we crowd the 'spikes' over to the right side of $U^c$. The application of the column permutation matrix, $Q$, causes no loss of numerical precision and results in the formation of a generalized upper Hessenberg matrix, $U^H$ with $k$ subdiagonals (Figure 3.4):

$$U^H = U^c Q$$

The $U^H$ may then be triangularized by elimination *with partial pivoting* to annihilate the subdiagonal elements. The elimination and permutation factors used for partial pivoting may be stored compactly as a string of operators, denoted by $S$, and held in product form. Then the restored upper triangular matrix $U_{new}$ may be computed (Figure 3.5):

$$U_{new} = SU^H$$

For the explicit form of the $LU$ factorization, $L$ need not be updated. However, the accumulated product string must remain stored for use in computing solutions to the updated factorization. In general, an updated $LU$ factorization takes the form:

$$B_{new} = LS^{-1}U_{new}Q^T$$

This factorization takes the same form as the rank-1 case. Note that by exploiting triangular factors, $S^{-1}$ need never be computed to obtain an updated solution involving $B_{new}$.

Figure 3.3. Updating $U$ to Express the Formation of $B_{new}$

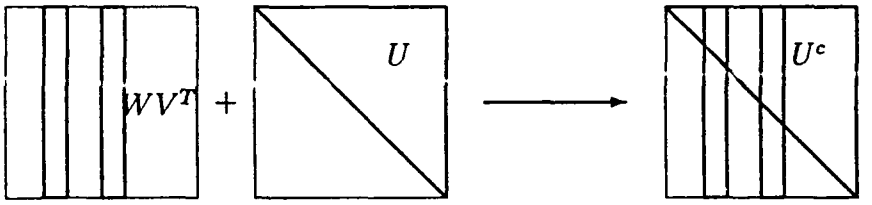



Figure 3.4. Permuting $U^c$ to Achieve Generalized Hessenberg Form

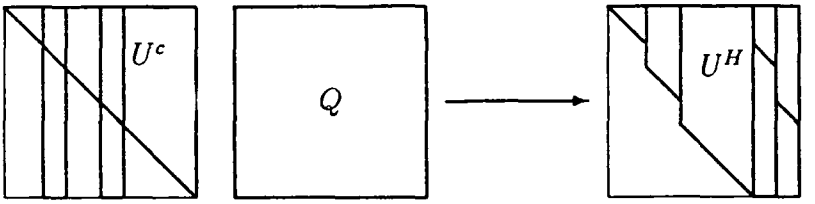
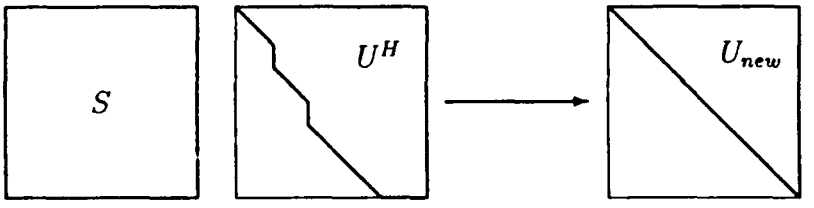



Figure 3.5. Restoring Upper Triangular Form



*3.5.4 Elimination Form of LU.* Another interesting $LU$ approach to the update is to arrange computations according to the elimination form. Suppose that we have the factorization $MB_{old} = U$ (elimination form where $M = L^{-1}$). It follows that:

$$MB_{new} = U + WV^T$$

where the matrix $W$ is solved according to the lower triangular system:

$$W = MD$$

Updating $U$:

$$U^c = U + WV^T$$

Where $U^c$ is an upper triangular matrix with 'spikes' in the columns indexed by 'change'. We may still take advantage of the remaining structure of the matrix. Specifically, permuting columns of $U^c$ into a generalized upper Hessenberg matrix, $U^H$ with $k$ subdiagonals, yields:

$$U^H = U^c Q$$

Note that the operation of the permutation matrix, $Q$, causes no loss of numerical precision. The $U^H$ may then be triangularized by elimination with partial pivoting to annihilate the subdiagonal elements. The elimination and permutation factors used for partial pivoting may be stored compactly as a string of operators, $S$, held in product form:

$$U_{new} = SU^H$$

$M$ is updated in the following manner:

$$M_{new} = SM$$

which provides the factorization:

$$B_{new} = M_{new}^{-1} U_{new} Q$$

Note that $M$ does not remain triangular since the permutations involved in the partial pivoting during elimination causes elements to spread above the diagonal. The form of this updated factorization is also the same as for the rank-1 case.

### *3.6 Applying General Methods*

Issues of computational performance include far more considerations than counts of floating point operations and memory requirements. Other important issues include memory access, stride, and other data movement overheads. The routines in the general numerical libraries (BLAS, Linpack, and etc) are highly refined to deal with all of these practical issues. Many versions exist which are tailored to specific machine architectures. However, the high level FORTRAN calling interface remains standard. Such generic routines enable reliable, modular implementations. Every effort was made to take advantage of these powerful computational kernals in constructing general $LU$ rank-$k$ update and solution routines.

There is little sense in re-inventing the wheel. Libraries of general numerical methods have become widely available for the solution of a number of numerical problems. Routines maintained in the Linpack, Quadpack, Minpack, and SLATEC libraries are known throughout the world's scientific and research communities. They are widely used within U.S. national laboratories and represent the state of the art in current mathematical software (11:4). Many of these routines have profitted from years of research, practical refinements and computational experience in organizing efficient and robust computations on both vector and scalar environments. These general methods provide FORTRAN access to high level, efficient and reliable computational kernals. Many of the kernals such as the level-1, level-2 and level-3 Basic

Linear Algebra Subroutines (BLAS), are already being implemented in hardware. The latest computational developments encourage the arrangement of computations to be rich in matrix multiplications (7). These computations can be performed efficiently on modern, high performance architectures.

Using either the explicit or elimination form of the $LU$ factorizations, the rank-$m$ linear equations

$$B_{new}x = b \quad \text{or} \quad B_{new}^T x = b$$

may be solved in $O(m^2)$ operations (6:143,146). This is the same order as the case for the explicit form of the inverse, but with improved numerical characteristics.

### *3.7 Choosing Explicit LU*

The update technique proposed in this study keeps an explict $L$ and the update stored in product form. The primary application of this form has been the solution of sparse linear programs (6:139–147). With sparse LPs, the growth of the product-string is retarded. However, the breadth-first tree search of the current ADBASE algorithm suggests other possible means for retarding the growth of the product-string by exploiting 'backtracking' types of algorithms (2). Without doing any computational backtracking, it is clear that breadth first arrangements minimize the likelihood of worst-case update situations where restoration of $U$ to trangular form requires the annihilation of $m-1$ subdiagonal elements. (6:143). The breadth-first tree traversal encourages the situation where the vectors which are likely to 'leave' will be the ones which have most recently 'entered'. This implies that very few elimination multipliers will be required for most updates to the basis.

# *IV. Results*

## *4.1 Decomposition of M-Simplex Algorithm*

The m-simplex algorithm has been decomposed into conceptually simpler elements. The bulk of numerical computations are concentrated in two elements:

1. Testing efficiencies of feasible edges.
2. Moving to the new vertex solutions.

Research and development centered upon the latter because that is where errors accumulate and stability is threatened. Standard numerical library routines, such as BLAS were incorporated where possible. As might be predicted, the lower level routines were the easiest to integrate. A convenient pair of Linpack routines were used to develop basic routines supporting m-simplex operations. Only one level-3 call appeared to be immediately practical. Incorporation of additional higher level BLAS routines would require the reformulation of algorithms to be richer in matrix multiplications. This would probably be the natural result of a second generation of development based upon the elimination form of the $LU$ factorization rather than the explicit form (as is case with the current routine).

## *4.2 Research Code*

This research has identified a need for implementations of general methods for solving updated systems of linear equations. An initial design based upon the explicit LU factorization has been implemented. Working code is listed in the appendices. It has achieved successful results using old factorizations produced by DGEFA (A Linpack PLU factorization routine). These initial routines are capable of solving regular and transposed systems of equations affected by rank-$k$ updates. The philosophy of the design was naturally influenced by some of the considerations in the

Linpack design. The processing is strongly oriented towards efficient use of FORTRAN's column-based structures. If refined, these routines could probably be made to perform very efficiently upon many platforms. It, also, relies heavily on level-1 BLAS and may potentially be refined to be an extremely portable routine. The current version uses COMMON declarations and an INCLUDE statements. These may be dropped from subsequent versions. The interface may also be further refined for simpler application.

*4.2.1 Experimental Subroutine Implementation.* Routines were developed to support the operation of the m-simplex requirements to generate solutions for new linear systems which are related by rank-$k$ updates to previously solved systems. URANKN updates the $LU$ factorizations used to solve old systems to a form which can be used by VECSL or VECTSL to solve new systems. This update technique is expected to be more stable than straight application of Sherman-Morrison formula and possibly faster. The basis of the update routines are the stabilized triangular system solution strategy which underlies the development of the $LU$ factorization, itself.

*4.2.2 Linpack Source Code.* The $LU$ factorization code which provides the initial decomposition (and could be used for periodic reinversons) is the Linpack routine DGEFA (See Appendix A.1). The ordinary Linkpack solution routine, DGESL (See Appendix A.2), was found to be a little too coarse for application to an update formula. It was modified by the introduction of additional entry and exit points to allow the direct and independent solution of equations involving explicit $L$ and $U$ factors. The added entry calls, DLSL, DUSL, DLSLT, DUSLT use the same formal parameters (identically defined) as the original DGESL parameters, except for JOB (JOB is treated as a dummy variable). Modification was implemented in a manner so that ordinary DGESL processing will not be affected. A more elegant solution would be to break DGESL into four component routines. The level-2 BLAS has a triangular

matrix system solver routine which will directly solve equations involving $U$, but $L$ would have to first be inverted to its elimination form. Further, permutations would have to be applied to ensure correct ordering of solutions. Modifying the DGESL routine appeared to be the most straight forward option to keep compatibility with DGEFA.

*4.2.3 BLAS Utilization.* DGEFA and DGESL are supported by calls to level-1 BLAS. The level-1 calls are also instrumental in supporting the lower level functions of the factorization update routines.

### *4.3 ADBASE and System Integration*

In order to develop a test bed for the factorization update routines, a version of ADBASE was modified with the intent of eventual integration of the research code, without disrupting ADBASE functionality. The organization of ADBASE code appears modular. However, subroutines have been discovered to be tightly integrated. The complete structure of the control elements is not clear and attempts at integration has been difficult and time consuming. Partial integration of the routines has been accomplished. Full integration continues to be viewed with optimism.

# Appendix A. *Linpack A = PLU Routines*

## *A.1 DGEFA*

```
c
c     ********************* LINPACK DGEFA ************************
c     *                                                          *
c     *   This Software has been obtained via "netlib ornl.gov." *
c     *                                                          *
c     ************************************************************
c
c
      subroutine dgefa(a,lda,n,ipvt,info)
      integer lda,n,ipvt(1),info
      double precision a(lda,1)
c
c     dgefa factors a double precision matrix by gaussian elimination.
c
c     dgefa is usually called by dgeco, but it can be called
c     directly with a saving in time if  rcond  is not needed.
c     (time for dgeco) = (1 + 9/n)*(time for dgefa) .
c
c     on entry
c
c        a       double precision(lda, n)
c                the matrix to be factored.
c
c        lda     integer
c                the leading dimension of the array  a .
c
c        n       integer
c                the order of the matrix  a .
c
c     on return
c
c        a       an upper triangular matrix and the multipliers
c                which were used to obtain it.
c                the factorization can be written  a = l*u  where
c                l  is a product of permutation and unit lower
c                triangular matrices and  u  is upper triangular.
c
c        ipvt    integer(n)
```

```
c                an integer vector of pivot indices.
c
c        info    integer
c                = 0  normal value.
c                = k  if  u(k,k) .eq. 0.0 .  this is not an error
c                     condition for this subroutine, but it does
c                     indicate that dgesl or dgedi will divide by zero
c                     if called.  use  rcond  in dgeco for a reliable
c                     indication of singularity.
c
c     linpack. this version dated 08/14/78 .
c     cleve moler, university of new mexico, argonne national lab.
c
c     subroutines and functions
c
c     blas daxpy,dscal,idamax
c
c     internal variables
c
      double precision t
      integer idamax,j,k,kp1,l,nm1
c
c
c     gaussian elimination with partial pivoting
c
      info = 0
      nm1 = n - 1
      if (nm1 .lt. 1) go to 70
      do 60 k = 1, nm1
         kp1 = k + 1
c
c        find l = pivot index
c
         l = idamax(n-k+1,a(k,k),1) + k - 1
         ipvt(k) = l
c
c        zero pivot implies this column already triangularized
c
         if (a(l,k) .eq. 0.0d0) go to 40
c
c           interchange if necessary
c
```

```
            if (l .eq. k) go to 10
               t = a(l,k)
               a(l,k) = a(k,k)
               a(k,k) = t
   10       continue
c
c           compute multipliers
c
            t = -1.0d0/a(k,k)
            call dscal(n-k,t,a(k+1,k),1)
c
c           row elimination with column indexing
c
            do 30 j = kp1, n
               t = a(l,j)
               if (l .eq. k) go to 20
                  a(l,j) = a(k,j)
                  a(k,j) = t
   20          continue
               call daxpy(n-k,t,a(k+1,k),1,a(k+1,j),1)
   30       continue
         go to 50
   40    continue
            info = k
   50    continue
   60 continue
   70 continue
      ipvt(n) = n
      if (a(n,n) .eq. 0.0d0) info = n
      return
      end
```

## *A.2 Modified DGESL*

```
c
c
c     ****************** LINPACK DGESL ***************************
c     *                                                          *
c     *           (MODIFIED FOR 'UPDATE' SOLN ENTRY)             *
c     *                                                          *
c     *   This Software has been obtained via "netlib ornl.gov."  *
c     *                                                          *
c     *   This is a LINPACK routine.  It relies upon level-1 and  *
c     *   level-2 BLAS.  If BLAS is not available at your site,   *
c     *   it can be obtained through the same source.             *
c     *                                                          *
c     *   All modifications are shown between "*-lines" which     *
c     *   are bounded by double dashed lines.                     *
c     *                                                          *
c     ************************************************************
c


      subroutine dgesl(a,lda,n,ipvt,b,job)
C=================================================================
C*****************************************************************
C Routines allow clean insertion of LU update code for LP
C simplex application. (Especially since I used standard
C DGECO or DGEFA code for initialization and re-inversions):
C
c
c     Solve  l*y = b:
C      entry dlsl(a,lda,n,ipvt,b,job)
C
C =======> LU update code may be inserted here!
c
c     Solve  u*x = y:
C      entry dusl(a,lda,n,ipvt,b,job)
C
C Modified by M. Shields, Jan '91:
C
C This program is hash of the standard LinPack double precision
C 'solve' routine for DGEFA factorizations.

C I have added entry and return codes to obtain use of
```

```
C fragments for solving triangular systems of LU factorizations:
C    -- Specifically created an integer flag called FRAG to
C       determine if want fragment or not ...
C
      integer frag

C*********************************************************************
C=====================================================================
      integer lda,n,ipvt(1),job
      double precision a(lda,1),b(1)
c
c     dgesl solves the double precision system
c     a * x = b  or  trans(a) * x = b
c     using the factors computed by dgeco or dgefa.
c
c     on entry
c
c        a       double precision(lda, n)
c                the output from dgeco or dgefa.
c
c        lda     integer
c                the leading dimension of the array  a .
c
c        n       integer
c                the order of the matrix  a .
c
c        ipvt    integer(n)
c                the pivot vector from dgeco or dgefa.
c
c        b       double precision(n)
c                the right hand side vector.
c
c        job     integer
c                = 0         to solve  a*x = b ,
c                = nonzero   to solve  trans(a)*x = b  where
c                            trans(a)  is the transpose.
c
c     on return
c
c        b       the solution vector  x .
c
c     error condition
```

```
c
c        a division by zero will occur if the input factor contains a
c        zero on the diagonal.  technically this indicates singularity
c        but it is often caused by improper arguments or improper
c        setting of lda .  it will not occur if the subroutines are
c        called correctly and if dgeco has set rcond .gt. 0.0
c        or dgefa has set info .eq. 0 .
c
c     to compute  inverse(a) * c  where  c  is a matrix
c     with  p  columns
c           call dgeco(a,lda,n,ipvt,rcond,z)
c           if (rcond is too small) go to ...
c           do 10 j = 1, p
c              call dgesl(a,lda,n,ipvt,c(1,j),0)
c        10 continue
c
c     linpack. this version dated 08/14/78 .
c     cleve moler, university of new mexico, argonne national lab.
c
c     subroutines and functions
c
c     blas daxpy,ddot
c
c     internal variables
c
      double precision ddot,t
      integer k,kb,l,nm1
c
      frag = 0
      nm1 = n - 1
      if (job .ne. 0) go to 50
c
c        first solve  l*y = b
c==========================================
c**********************************************
      goto 5
      entry dlsl(a,lda,n,ipvt,b,job)
         frag = 1
         nm1 = n - 1
    5    continue
C**************************************************
c==========================================
```

```
         if (nm1 .lt. 1) go to 30
         do 20 k = 1, nm1
            l = ipvt(k)
            t = b(l)
            if (l .eq. k) go to 10
               b(l) = b(k)
               b(k) = t
   10       continue
            call daxpy(n-k,t,a(k+1,k),1,b(k+1),1)
   20    continue
   30    continue
c=============================================
c*********************************************
C     -- Return from DLSL entry:
         if (frag.eq.1) return

      entry dusl(a,lda,n,ipvt,b,job)
c*********************************************
c=============================================
c
c        now solve  u*x = y
c
         do 40 kb = 1, n
            k = n + 1 - kb
            b(k) = b(k)/a(k,k)
            t = -b(k)
            call daxpy(k-1,t,a(1,k),1,b(1),1)
   40    continue

      go to 100
c=============================================
C*********************************************
      entry duslt(a,lda,n,ipvt,b,job)

       frag = 1
C*********************************************
c=============================================
   50 continue
c        job = nonzero, solve  trans(a) * x = b
c        first solve  trans(u)*y = b
c
         do 60 k = 1, n
```

```
            t = ddot(k-1,a(1,k),1,b(1),1)
            b(k) = (b(k) - t)/a(k,k)
   60     continue
c=============================================
c*********************************************
C      -- Return from DUSLT entry:

        if (frag.eq.1) return

        goto 65

         entry dlslt(a,lda,n,ipvt,b,job)

         frag = 1
         nm1 = n - 1

 65     continue
c*********************************************
c=============================================
c
c         now solve trans(l)*x = y
c
          if (nm1 .lt. 1) go to 90
          do 80 kb = 1, nm1
             k = n - kb
             b(k) = b(k) + ddot(n-k,a(k+1,k),1,b(k+1),1)
             l = ipvt(k)
             if (l .eq. k) go to 70
                t = b(l)
                b(l) = b(k)
                b(k) = t
   70        continue
   80     continue
   90     continue

  100 continue
      return
      end
```

# Appendix B. *LU Factorization Update and Solution Routines*

## *B.1 Update LU Factorization*

```
%********************** CSWP.FOR **********************************

C To support multiplying strings of elementary matrices in sweeps:
      integer currpv,fullpv,overpv,strtswp,
     :        currmu,nxtuswp,fullmu,overmu
      common /cswp/
     :        currpv,fullpv,overpv,strtswp,
     :        currmu,nxtuswp,fullmu,overmu



%********************** URANKIN ****************************

      subroutine urankn(m,n,a,u,lu,ib,piv,iwk,p,
     :                  mu,swp,changes,enters,leaves)

C     This routine updates an m by m LU factorization to
C     reflect a rank-n update.  "A" contains the column
C     vectors from which the original factorization and
C     update vectors are taken.  "U" is a workspace where
C     intermediate...

      implicit none

      include 'cswp.for/list'

      integer m,n,h,kept,i,kk

      double precision a(m,*),lu(m,*),u(m,*)

      integer enters(*),leaves(*),changes(*)

      double precision mu(*)

      integer  swp(1:2,*)
      integer ib(*),piv(*),iwk(*),p(*)

C Update U without reference to L:
```

```
C     check if full: then do reinversion...
C******** add code for triggering reinversions:
C

C Determine entering vectors:
      do 10 i = 1,n

C      'copy vector to U:'
       call dcopy(m,a(1,enters(i)),1,u(1,i),1)

C      'vector difference:'
       call daxpy(m,-1.d0,a(1,leaves(i)),1,u(1,i),1)

C      'Solution ::dlsl'
       call dlsl(lu,m,m,piv,u(1,i),kk)

       call daxpy(changes(i),1.d0,lu(1,changes(i)),1,u(1,i),1)

10    continue

      strtswp = currpv + 1
      nxtuswp = currmu + 1

C Save links to changes to be made in the basis:
      call markupd(m,ib,n,iwk,changes,kept)

C Sweep matrix into permuted triangle form:
      call permu(m,lu,mu,swp,p,kept,n,iwk)

C Establish the new index:
      call permidx(m,p,ib)

C Update matrix and ib:
      call updmu(m,n,lu,mu,swp,ib,kept,u,enters)

C Update of U complete.

      end
```

```
      subroutine updmu(m,n,lu,mu,indices,ib,kept,u,enters)

      implicit none

      include 'cswp.for/list'

      integer m,n,i,kept,last
      double precision lu(m,*),u(m,*)
      integer enters(*),ib(*)

C  "pivot-strings" ...
      double precision mu(*)
      integer  indices(1:2,*)

      integer ibi,kk,zz

C For update i: (i indexes the update event and is associated with U)
C    enters(i) is the index of A that enters
C    changes(i) is the index of LU vector that is updated and temporarily
C       stored in the index set, ib(m).

      last = kept + n

      do 10 i = kept+1, last
C        (For the changed U vectors).

         ibi = ib(i)
         call vecswp(indices,mu,u(1,ibi))
C--------Eliminate last h elements to fit triangular form:
         if (i.lt.m)
     :     call annih(
     :        m,
     :        m - i,
     :        u(1,ibi),
     :        indices,
     :        mu          )

C--------Place U vector into original LU:
         call dcopy(i,u(1,ibi),1,lu(1,i),1)
```

```
C--------Record 'enters' index over 'changes' index in index vector, ib:
          ib(i) = enters(ibi)
10     continue
       if ( m.gt.last) then

         call vecswp(
     :          indices,
     :          mu,
     :          lu(1,m) )
       end if

       end


       subroutine permidx(n,ip,ib)

C This routine reorders an index set matrix, ib, by a permutation
C matrix, ip:
C   It computes the product (ip*ib).
C (Permutation matrix packed into the vector ip.)

       implicit none

       integer n, ip(n), ib(n), i, k, index,kk

       if (n.lt.2) return

C--Do until all items in ib are mapped by ip:
       i = n
10     continue
         k = ip(i)
         if (k .ne. i) then
           index = ib(k)
           ib(k) = ib(i)
           ib(i) = index
         end if
         i = i - 1
       if (i.gt.1) go to 10

       end
```

```
      subroutine markupd(m,ib,n,iwk,changes,kept)

C This routine marks changes in basic indices:
C Must be called only after the update vectors have
C been formed (But not yet reduced).

      implicit none

      integer m,n,ib(*),changes(*),iwk(*),kept,i,j,kk

      logical ibubswp

      do 5 i = 1, n
C------Put indices into work vector for sorting and working with...
        j = changes(i)
        iwk(i) = j
C------These indices are to the change matrix outside of LU:
        ib(j) = i
5     continue

C
C Ensure ascending order for iwk:
      i = n
10    if (ibubswp(i,iwk)) then
        i = i - 1
      goto 10
      end if

C Identify the right most-vector column vector of U,
C (For after permutation!) that goes unchanged.
      kept = m - n - 1
      if (iwk(n).eq.m) kept = kept + 1

      end




      subroutine permu(m,lu,mu,indices,p,kept,n,iwk)

      implicit none
```

```
      integer m,n,p(*),iwk(*),kk,zz,cnt
      integer left,i,k,h,j,kept,kp1
      integer indices(1:2,*)
      double precision lu(m,*),mu(*)


C Reduce and reorder the 'kept' basic vectors into a triangular form:
C Also, track permutations of the columns in packed-permutation matrix, p.
      cnt = 0
      h = 0
      j = iwk(1)
      do 20 i = 1,m
        if (i.eq.j) then
          h = h+1
          if (h.lt.n) j = iwk(h)
        else
C        (For the kept U vectors).
          cnt = cnt + 1
C--------Eliminate last h elements to fit left-shifted triangular form:

          if (h.gt.1)  then

            call vecswp(indices,mu,lu(1,i))
          end if
          call annih(
     :          i,
     :          h,
     :          lu(1,i),
     :          indices,
     :          mu           )

C--------Shift U vector left to crowd into original U:
          left = cnt
          call dcopy(left,lu(1,i),1,lu(1,left),1)

C--------Record column permutation of kept U in P:
          p(cnt) = cnt + h
        end if
      if (cnt.eq.kept) goto 25
20    continue
25    continue
```

```
C Done with permuting unchanged, basic column vectors...
C
C Now, finish p:
C
C Initialize p:
      do 30 i = kept+1,kept+n
        p(i) = 0
30    continue

C----Avoid permuting columns originally right of the last kept vector:
C    Put non-permuted change columns where before the permutations started.
      do 40 i = h+1,n
        k = iwk(i)
        p(k) = k
40    continue
      p(m) = m

C----Fill-in rest of p with remaining permuted, update columns:
      i = kept
      do 60 j = 1,h
C      -- Find a free column to accept iwk(j):
50      continue
          i = i + 1
        if (p(i).ne.0) goto 50
        p(i) = iwk(j)
60    continue

      end

      logical function ibubswp(n,p)
C Tests for bubble swaps ... used in a bubble sorts.
C Caution: Overwrites array.
C Last array elem. largest after 1 pass.
      implicit none
      integer p(*),n,i,h

      ibubswp = .false.
      do 30 i = 1,n - 1
        h = p(i+1)
        if (p(i) .gt. h) then
          p(i+1) = p(i)
```

```
          p(i) = h
          ibubswp = .true.
        end if
30    continue

      end



      subroutine annih(n,h,v,ip,mu)
C-- Description:
C-- Assumes element V(n) is non-zero: Constructs Gaussian elimination
C-- operations to annihilate from V the last H elements, pivoting
C-- on a single element.  This pivot element, at position (N - H)
C-- is the largest of the last ( H + 1 ) of the original elements in V.
C
C  MU is an array which records the string of multipliers involved
C  in the annihilation of V.
C
C  IP is an array which records descriptions of permutations and the
C  number of multipliers involved during the annihilation.
C

C-- 1.  Construct a stabilized transformation (partial pivoting and
C--     gaussian elimination), ...
C-- 2.  Use it to eliminate h components of v ...
C--      --- A swap adds to list ip(1:2,*) only.
C--      --- and elimination adds to mu as well.
C-- 3.  Pass essentials out to join string of pivot matrices for later
C--     update sweeps.
C--
C-- Works in association with VECSWP and VECTSWP.  These routines
C-- decode any series of annihilations made with this routine
C-- and applies them to update a vector.  The updated vector
C-- may then be applied to an appropriate triangular system.
C--

      implicit none

      double precision v(*), mu(*),swap
```

```
      integer n,h,ip(1:2,*),i,k,l,idamax,inc


      include 'cswp.for/list'

            external idamax

C-----h is the count of elements to elim. from v :
      if (h.eq.0) return
      i = n - h
      l = idamax(h+1,v(i),1) + i - 1
C-----(If singularity probable, need test for pivot tolerance here!)
      if (l.gt.i) then
C------Pivot involves a row swap!
        swap = v(i)
        v(i) = v(l)
        v(l) = swap
C------Signal: upper triangular form.
        currpv = currpv + 1
        ip(2,currpv) = i
        if (h.eq.1) then
         ip(1,currpv) = i
C--------find single elimination factor and return early:
C        (implicitly, n = i+1 = l ).
          currmu = currmu + 1
          mu(currmu) = v(l)/v(i)
          return
        else
          ip(1,currpv) = l
        end if
      end if

C----(Swap, if was necessary, is complete ...)
C    Now for the elimination ...

C----Signal: Lower triangular form for this pivot.
      currpv = currpv + 1
      ip(1,currpv) = i
      ip(2,currpv) = n

C----Compute elimination factors: Annihilates remaining v...
      do 10 k = 1, h
```

```
        mu(k+currmu) = v(i+k)/v(i)
10    continue
C----Update ptr to last elimination multiplier computed.
      currmu = currmu + h

      end
```

## *B.2 Solve Ax = b Using Updated LU Factorization*

```
c
c
c     ************************************************************
c     *                                                          *
c     *    The following software routines have been written     *
c     *    to support and demonstrate the LU rank-n              *
c     *    factorization updating and solution procedures        *
c     *    described in this Thesis.  All of the routines in     *
c     *    these appendicies were tested together and function   *
c     *    properly to yield expected results.                   *
c     *                                                          *
c     *    They have not yet been fully integrated in a          *
c     *    complete M-Simplex package.                           *
c     *                                                          *
c     ************************************************************
c
c     Rank-n solution routine.
c     This routine replaces the solve routine, DGESL, when working
c     with factorizations updated by URANKIN.  VECSL solves the
c     updated problem: (Ax = b).
c

      subroutine vecsl(m,b,p,a,piv,swp,mu)
      implicit none
      integer m, kk, zz
      double precision b(*),a(m,*)
      double precision mu(*)
      integer  swp(1:2,*)
      integer piv(*),p(*)

      include 'cswp.for/list'

      call dlsl(a,m,m,piv,b,kk)
      call vecswp(swp,mu,b)
      call dusl(a,m,m,piv,b,kk)
      call vecperm(m,b,p)

      end
```

```
      subroutine vecperm(n,v,ip)

C This routine reorders a vector according to a permution matrix
C (matrix packed into the vector ip):

      implicit none

      integer n, ip(n), i, k
      double precision a, v(n)

      if (n.lt.2) return
      do 10 i=n,1,-1
        k = ip(i)
        if (k .lt. i) then
          a = v(k)
          v(k) = v(i)
          v(i) = a
        end if
10    continue

      end
```

```
      subroutine vecswp(indices,mu,v)
C
C Computes the product of a string of stabilized eliminations
C  (applies partial pivoting (pair-wise for Rank-$n$) and a dense vector, v.
C
C----To perform a complete sweep (correctly), must start at the beginning:
C--
C--  1.  The pivot counter PIVCNT must be reset to 1, and ...
C--  2.  The pointer, NXTUSWP from the COMMON /cswp/, must be
C--    reset to 1.
C--
C--  Explanation: There may be many multipliers to a pivot, and
C--  NXTUSWP points to the set of annihilators (multipliers) associated
C--  with a pivot, PVCNT.  The number of multipliers can be gleaned from
C--  the information in INDICES(.,pvcnt)
C

      implicit none

      integer pvcnt,i,j,h

      include 'cswp.for'

      double precision v(*), mu(*), a
      integer indices(1:2,*)

      nxtuswp = 1
      do 10 pvcnt = 1, currpv
        i = indices(1, pvcnt)
        j = indices(2, pvcnt)
        h = j - i
        a = v(i)
C Use form of update message to determine update action:
        if (h.gt.0) then
C    --   lower triangle -- Avoid permutation: annihilate.
          call daxpy(h,-a,mu(nxtuswp),1,v(i+1),1)
```

```
          nxtuswp = nxtuswp + h
        else
C        -- Upper triangle pivot element -- Complete permutation:
         if (h.eq.0) then
C         -- Since only one, annihilate the lower triangle element
C         -- after swap. (Operations combined, here:)
          h = i + 1
          v(i) = v(h)
          v(h) = a - v(h)*mu(nxtuswp)
          nxtuswp = nxtuswp + 1
         else
C         -- Just a swap: (No annihilation).
          v(i) = v(j)
          v(j) = a
         end if
        end if
10     continue
       end
```

## *B.3 Solve $A^T v = c$ Using Updated LU Factorization*

```
c
c
c     ************************************************************
c     *                                                          *
c     *    The following software routines have been written     *
c     *    to support and demonstrate the LU rank-n              *
c     *    factorization updating and solution procedures        *
c     *    described in this Thesis.  All of the routines in     *
c     *    these appendicies were tested together and function   *
c     *    properly to yield expected results.                   *
c     *                                                          *
c     *    They have not yet been fully integrated in a          *
c     *    complete M-Simplex package.                           *
c     *                                                          *
c     ************************************************************
c
c     Rank-n solution routine.  (transposed case)
c     This routine replaces the solve routine, DGESL, when
c     working with factorizations updated by URANKN.  VECTSL
c     solves the transposed problem: (transposed(A)x = b)
c
c
      subroutine vectsl(m,b,p,a,piv,swp,mu)

      implicit none
      integer m, kk, zz
      double precision b(*),a(m,*)
      double precision mu(*)
      integer  swp(1:2,*)
      integer piv(*),p(*)

      include 'cswp.for/list'
C Solve a transposed system
      call vectperm(m,b,p)
      call duslt(a,m,m,piv,b,kk)
      call vectswp(swp,mu,b)
      call dlslt(a,m,m,piv,b,kk)
```

```
      end


      subroutine vectswp(indices,mu,v)
C
C Computes the product of a transposed string of stabilized eliminations
C  pivots and a dense vector, v.  The sweep begins at currpiv and is
C  completed when have reached the first pivot matrix.
C
      implicit none

      integer pvcnt, i, j, h, bckswp

      include 'cswp.for'
      double precision v(*),mu(*),a,ddot
      integer indices(1:2,*)

      external ddot

      bckswp = nxtuswp
      do 10 pvcnt = currpv,1,-1
        i = indices(1, pvcnt)
        j = indices(2, pvcnt)
        h = j - i
        if (h.gt.0) then
C lower triangle -- Avoid permutation:
          bckswp = bckswp - h
          v(i) = v(i) - ddot(h,mu(bckswp),1,v(i+1),1)
        else
          a = v(i)
C Upper triangle pivot element -- Complete permutation:
          if (h.eq.0) then
C         -- (Operations combined, here:)
            h = i + 1
            v(i) = v(h)
            bckswp = bckswp - 1
            v(h) = a - v(h)*mu(bckswp)
          else
C         -- Just a swap:
            v(i) = v(j)
            v(j) = a
```

```
          end if
        end if

10    continue
      end



      subroutine vectperm(n,v,ip)

C This routine reorders a vector according to a permution matrix
C (matrix packed into the vector ip):

      implicit none

      integer n, ip(n), i, k
      double precision a, v(n)

      if (n.lt.2) return
      do 10 i= 1, n
        k = ip(i)
        if (k .gt. i) then
          a = v(i)
          v(i) = v(k)
          v(k) = a
        end if
10    continue

      end
```

# Appendix C. *Example Test Programs*

```
C-- Enters rank-1 update, solves system of equations.
C
      program test
C-- try out l.a. routines and LU- update code.: see page 144 Gill et. all.

      implicit none

      integer over,lda, m, kk, i , j
        parameter(
     :     over = 100,
     :     lda = 5,
     :     m = 3
     :     )
C Problem data:
      double precision a(m,lda) / 5.d0, 2.d0, 1.d0,
     :                            4.d0, 1.d0, 1.d0,
     :                            3.d0, 2.d0, 1.d0,
     :                            1.d0, 1.d0, 3.d0,
     :                            1.d0, 1.d0, 1.d0 /
      integer enters(m) / 4, 0, 0 /,
     :        leaves(m) / 1, 0, 0 /,
     :        changes(m) / 1, 0, 0 /

C Routine workspace:
      include 'cswp.for/list'

      double precision mu(1:over),lu(m,m),u(m,2)

      integer  swp(1:2,1:over),n
      integer piv(m),iwk(m),ib(m) / 1, 2, 3 /,p(m)

      double precision invcond, b(m) /-2.d0,-5.d0,-4.d0/, s
      double precision w(m)

      fullpv = 75
      fullmu = 75
      overpv = over
      overmu = over
      currpv = 0
      currmu = 0
```

```
      n = 1
C      write (*,10) ((a(i,j),i=1,m),j=1,m)
10    format (3f10.3,/)
      print *
      do 30 i = 1,m
        call dcopy(m,a(1,i),1,lu(1,i),1)
30    continue
      write (*,10) ((lu(i,j),j=1,m),i=1,m)
      call dgeco(lu,m,m,piv,invcond,w)
      write (*,10) ((lu(i,j),j=1,m),i=1,m)
      print *
      print *,'update LU:'

      call urankn     (m,n,a,u,lu,ib,piv,iwk,p,
     :                 mu,swp,changes,enters,leaves)
      write (*,10) ((lu(i,j),j=1,m),i=1,m)
C      print *
      print *, (piv(i),i=1,m)
      print *, (p(i),i=1,m)
      print *, invcond,(ib(i),i=1,m)
      print *,' swp:'
      print *,((swp(j,i), j=1,2),i=1,4)
      print *, 'mu: ' ,(mu(i),i=1,3)
      print *,'solve b:',b

      call vecsl(m,b,p,lu,piv,swp,mu)
      print *,b
      end
```

```
      program testr
C-- try out l.a. routines and LU- update code.: see page 144 Gill et. all.
C-- For transposed systems:
C-- Rank-1 update and transposed sol'n.
C

      implicit none
      integer over,lda, m, kk, i , j
        parameter(
     :     over = 100,
     :     lda = 5,
     :     m = 3
     :     )
C Problem data:
      double precision aa(m,m) /
     :                          1.d0, 1.d0, 3.d0,
     :                          4.d0, 1.d0, 1.d0,
     :                          3.d0, 2.d0, 1.d0  /

      double precision bb(m) /-2.d0,-5.d0,-4.d0/
      double precision bt(m) /-2.d0,-5.d0,-4.d0/
      double precision bbb(m) /-2.d0,-5.d0,-4.d0/

      double precision a(m,lda) / 5.d0, 2.d0, 1.d0,
     :                          4.d0, 1.d0, 1.d0,
     :                          3.d0, 2.d0, 1.d0,
     :                          1.d0, 1.d0, 3.d0,
     :                          1.d0, 1.d0, 1.d0 /
      integer enters(m) / 4, 0, 0 /,
     :        leaves(m) / 1, 0, 0 /,
     :        changes(m) / 1, 0, 0 /

C Routine workspace:
      include 'cswp.for/list'

      double precision mu(1:over),lu(m,m),u(m,2)
      integer  swp(1:2,1:over),n
      integer piv(m) iwk(m),ib(m) / 1, 2, 3 /,p(m)
      double precision invcond, b(m) /-2.d0,-5.d0,-4.d0/, s
      double precision w(m)
```

```
      fullpv = 75
      fullmu = 75
      overpv = over
      overmu = over
      currpv = 0
      currmu = 0
      n = 1
10    format (3f10.3,/)

      do 30 i = 1,m
        call dcopy(m,a(1,i),1,lu(1,i),1)
30    continue
      write (*,10) ((lu(i,j),j=1,m),i=1,m)
      call dgeco(lu,m,m,piv,invcond,w)
      call urankn(m,n,a,u,lu,ib,piv,iwk,p,
     :                  mu,swp,changes,enters,leaves)
      write (*,10) ((lu(i,j),j=1,m),i=1,m)
      print *
      print *, (piv(i),i=1,m)
      print *, (p(i),i=1,m)
      print *, invcond,(ib(i),i=1,m)
      print *,'solve x: b =  Btransposed*x, '

      call vectsl(m,b,p,lu,piv,swp,mu)
      print *,'factored x = ',b
      print *,'verification:'

C transposed system sol::
      call dgeco(aa,m,m,piv,invcond,w)
C      write (*,10) ((aa(i,j),j=1,m),i=1,m)
      print *,'transposed soln: '

      call duslt(aa,m,m,piv,bt,1)
      call dlslt(aa,m,m,piv,bt,1)
      print *,bt

      print *,'Straight dgesl soln:(transposed:) '
      call dgesl(aa,m,m,piv,bbb,1)
      print *,bbb

      end
```

## *Bibliography*


1. Bazaraa, Mokhtar S., John J. Jarvis and Hanif D. Sherali. *Linear Programming and Network Flows.* 2nd ed. New York: John Wiley & Sons, Inc., 1990.
2. Brassard, Gilles and Paul Bratley. *Algorithmics: Theory and Practice.* Englewood Cliffs: Prentice Hall, Inc., 1988.
3. Dongarra, J. J., C. B. Moler, J. R. Bunch, G. W. Stewart. "LINPACK User's Guide," SIAM, Philadelphia,1979.
4. Dougherty, Edward R. and Charles R. Giardina. *Mathmatical Methods for Artificial Intelligence and Autonomous Systems.* Englewood Cliffs: Prentice Hall, Inc., 1988.
5. Duff, I. S., A. M. Erisman and J. K. Reid. *Direct Methods for Sparse Matricies.* New York: Oxford University Press, 1986.
6. Gill, Phillip E., Walter Murray and Margaret H. Wright. *Numerical Linear Algebra and Optimization.* Vol. 1. Redwood City: Addison-Wesley Publishing Company, 1990.
7. Golub, Gene H. and Charles F. Van Loan. *Matrix Computations.* 2nd ed. Baltimore: Johns Hopkins University Press, 1990.
8. Johnson, Krista E. *Frequency Assignments for HFDF Receivers in a Search and Rescue Network.* Master Thesis. School of Engineering, Air Force Institute of Technology (AU). Wright-Patterson AFB OH, March 1990.
9. Karmarkar, N. "A New Polynomial-Time for Linear Programming." *Combinatorica,* 4(4) (1984), 373-395.
10. Mandell, Stephen L. *Principles of Data Processing.* 2nd ed. St. Paul: West Publishing Company, 1981.
11. Kahaner, David, Cleve Moler and Stephen Nash. *Numerical Methods and Software.* Englewood Cliffs: Prentice Hall, 1989.
12. Murty, Katta G. *Linear Programming.* New York: John Wiley & Sons, 1983.
13. Rice, John R. *Matrix Computations and Mathematical Software.* New York: McGraw-Hill, Inc., 1981.
14. Strang, Gilbert. *Linear Algebra and its Applications,* 3rd ed. San Diego: Harcourt Brace Jovanovich, 1988.
15. Steuer, Ralph E. *ADBASE Operating Manual.* Vers. 9/89, Part 1. Department of Management Science, Brooks Hall, University of Georgia. Athens GA, September 1989.
16. Steuer, Ralph E. *MUltiple Criteria Optimization: Theory, Computation, and Application.* New York: John Wiley & Sons, 1986.

17. Yu, Po-Lung. *Multiple-Criteria Decision Making: Concepts, Techniques, and Extensions*. New York: Plenum Press, 1985.

## *Vita*

Captain Michael A. Shields was born 18 April 1963 in Seattle, Washington. He graduated from Onteora Senior High School, Boiceville New York in 1981. In May of 1985 he graduated from the USAF Academy and was commissioned as a Second Lieutenant. After graduation he was assigned to the 4201 Test Squadron (now the 49TESTS), Barksdale AFB, Louisiana, where he worked as a operational test engineer and analyst for a variety of programs involving SAC air-launched weapon systems. He entered the School of Engineering, Air Force Institute of Technology at Wright Patterson AFB Ohio in 1990 and is currently assigned to the Air Force Center for Studies and Analysis at the Pentagon.