Implementing the simplex method for the Optimization Subroutine Library

by J. J. H. Forrest
J. A. Tomlin

In this paper we describe the simplex algorithm and briefly discuss the interaction of the detailed implementation of the algorithm with the changes in computer hardware over the last 30 years. Then we give one example of the design changes needed to implement the method efficiently for the IBM 3090™ vector architecture. For the later RISC System/6000™ implementation, it was necessary to rethink this yet again. Finally we discuss the issue of robustness and the steps that were taken to give maximum reliability in the simplex algorithm in the IBM Optimization Subroutine Library.

The linear programming (LP) problem can be expressed in a number of canonical forms. We express it in the very general form:

$$\min_{x} \sum_{j} c_j x_j \tag{1}$$

subject to:

$$Ax = b \tag{2}$$

$$L_j \leq x_j \leq U_j \tag{3}$$

If the number of variables, n, were equal to the number of constraints, m, Equation (2) would be a set of simultaneous equations with (at most) a single solution. There would be no possibility of optimization. Normally, n is significantly greater than m; a typical medium-sized LP model might have 20 000 variables and 5000 constraints. The matrix A is almost always very sparse, typically with less than ten nonzero elements in each column—a very important factor in the development of efficient algorithms. In geometric terms the constraints describe a (convex) polyhedron in the solution space of the x variables. Points on or inside the polyhedron are *feasible* solutions. Those outside the polyhedron are said to be *infeasible*.

One could imagine trying to solve the LP problem by choosing a subset of m columns of A, fixing the other variables at a bound and attempting to solve the equations. Not all choices of m variables will be valid, but many (perhaps very many) will, and for each of these variables we would have a solution and thus a value of the objective expressed in (1). Each such solution corresponds to a *vertex* of the polyhedron of feasible solutions, and it can be shown that an optimum solution must occur at a vertex. It follows that by evaluating all possible choices we could (in theory) obtain the optimal choice of variables—optimal in the sense that the function of (1) is minimized. Although such an approach would be extremely ill-advised in practice, an intelligent, ordered search of this set of solutions is in fact the underpinning of the simplex method to be discussed in this paper. Geometrically, this method moves from one vertex to an adjacent vertex with improved solution value and continues until no such improvement is possible.

The power of the linear programming model as a decision-making tool derives from a number of circumstances:

- Many real-world problems can be modeled in this way (see, e.g., Williams[1]), from problems as simple as blending animal feedstuffs to those as complex as modeling all of the flights of a large airline. The lower and upper bounds on the variables correspond to real constraints on the values that a decision variable may take, the most normal lower bound being zero. For example, the amount of oil pumped through a pipe may not be negative, nor may it exceed the capacity of the pipe.
- Simple extensions to the linear objective and linear constraints to allow for some nonlinearities make the technique even more powerful.
- In a competitive market a "best" solution is needed, and other heuristic techniques do not guarantee finding an optimal solution.
- Powerful algorithms exist for solving such linear programs (LPs), implemented in robust software packages, which not only solve them but have facilities for model manipulation and standard interfaces with model management systems.

Basic algorithmic approach

In this section we give a very simplified version of the primal simplex algorithm. Later we will give a version that is closer to modern implementations, but it is more important that the reader grasp the basic concepts than the details. Interested readers are referred to Dantzig.[2]

For most of this paper we will work with a slightly simpler representation of the model, where the general bounds in (3) are replaced by simple nonnegativity constraints:

$$x_j \geq 0 \tag{3a}$$

In practice many of the constraints are often in inequality form, e.g.,

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i$$

In such cases we introduce "slack" variables to convert the constraints into equalities. The columns corresponding to these variables are unit vectors that have a one in row i and a zero c_j coefficient in the objective, or cost, function. All other variables will be termed "structural."

Following the practice hinted at in the introduction, let us divide the variables into two groups—a set of $(n - m)$ *independent* or *nonbasic* variables x_N and m *dependent* or *basic* variables x_B. The latter are called basic because their corresponding m columns B are assumed to be linearly independent. The LP problem may be rewritten:

$$\min_x c_B^{\top} x_B + c_N^{\top} x_N \tag{4}$$

subject to:

$$Bx_B + Nx_N = b \tag{5}$$

$$x_B, x_N \geq 0 \tag{6}$$

where $A = (B, N)$. If we set x_N to some plausible feasible values, we may write:

$$x_B = B^{-1}b - B^{-1}Nx_N \tag{7}$$

and in particular when we choose $x_N = 0$, then $x_B = B^{-1}b$, which results in a value of the objective of $z = c_B^{\top} B^{-1} b$.

For simplicity, we assume that this solution is "feasible," i.e., that x_B is nonnegative. It is, however, unlikely to be optimal. To search for a better solution, we see what happens if we change one of the nonbasic (independent) variables. From (4) and (5) we may express the objective completely in terms of the nonbasic variables as $z + (c_N^{\top} - c_B^{\top} B^{-1} N) x_N$. Now consider the *"reduced costs"* for columns $a_{.j}$ in N:

$$d_j = c_j - c_B^{\top} B^{-1} a_{.j} \qquad (a_{.j} \in N) \tag{8}$$

For any $d_j < 0$ the objective will decrease as x_j is incrementally increased. If no such negative d_j exists, the solution is indeed optimal, and we are done. Otherwise we must find out how far this nonbasic variable can be increased without causing a basic variable to become negative. Let the value of the chosen x_j be a parameter $\theta \geq 0$. We see that:

$$x_B = B^{-1}b - \theta B^{-1}a_{.j} \tag{9}$$

In the event that all elements of $B^{-1}a_{.j}$ are nonpositive, the objective can be decreased without limit as θ increases from zero. This *unbounded* situation is normally only encountered for incorrectly formulated models. Otherwise we increase θ until some member of x_B reaches zero, move the corresponding column of B to N, and replace it by $a_{.j}$ to obtain a new basis. This process is repeated, possibly thousands of times, until we reach the optimal solution.

In practice the task of finding the set of j where $d_j < 0$ is done in two parts. First we compute the *shadow prices* or *pi values*:

$$\pi^{\mathsf{T}} = c_B^{\mathsf{T}} B^{-1} \tag{10}$$

and then

$$d_j = c_j - \pi^{\mathsf{T}} a_{.j} \qquad (a_{.j} \in N)$$

The other major computational effort is in finding $B^{-1}a_{.j}$.

History

The simplex method was invented by George Dantzig about 1947. Its usefulness and suitability for the then newly invented electronic computer was soon recognized and exploited. By 1951 the method had been implemented on an IBM card programmable calculator. This machine had 16 to 80 registers and used punched cards. Initially a new copy of B^{-1} had to be punched in cards for each iteration, but it was seen by 1954 that the algorithm could be adapted to suit the computer architecture.

A crucial advance was to maintain B^{-1} as a product of *elementary transformations*, or

$$B^{-1} = E_l E_{l-1} \cdots E_2 E_1 \tag{11}$$

where the E_k are $m \times m$ identity matrices with just one column $\eta_{.k}$ modified, i.e.,

$$E_k = \begin{bmatrix} 1 & & & \eta_{1k} & & & \\ & \ddots & & \vdots & & & \\ & & 1 & & & & \\ & & & \eta_{p_k k} & & & \\ & & & & 1 & & \\ & & & \vdots & & \ddots & \\ & & & \eta_{mk} & & & 1 \end{bmatrix} \tag{12}$$

These transformations are often referred to as "column etas" or just "etas," and only one E_k need be punched out per iteration and added to the factorization of B^{-1}. Periodically B^{-1} was computed from scratch to eliminate the buildup of round-off error and reduce the length of the eta file.

A problem with 45 constraints and 70 variables was considered large at that time and could take about eight hours to solve. However, the basic concepts of simplex implementation had been set for the next 30 years. As we have said, the model matrix A is usually very sparse. The elementary transformation vectors may be denser but still only of the order of hundreds of nonzeros for a problem with thousands of rows.

The work involved in the major tasks of each simplex iteration are then:

1. Form a dense m-vector c_B^{T} and apply a sequence of sparse transformation vectors $c_B^{\mathsf{T}} \leftarrow c_B^{\mathsf{T}} E_k$ for $k = \{l, l-1, \cdots 2, 1\}$ to obtain π.
2. Compute $d_j = c_j - \pi^{\mathsf{T}} a_{.j}$ for some columns j. This involves taking the inner product of a series of sparse vectors $a_{.j}$ with a dense m-vector π.
3. Expand the chosen (sparse) $a_{.j}$ into a full m-vector and apply the sequence of elementary transformation vectors $a_{.j} \leftarrow E_k a_{.j}$ for $k = \{1, 2, \cdots l-1, l\}$.

Each of these major operations consists of sequentially applying sparse inner products or additions to a dense vector of length m; so if we have enough registers or high-speed memory to contain m values, we can achieve a reasonable speed. Also note that while operations 1 and 3 involve the basis inverse, step 2 could scan the entire matrix N. If $n >> m$, this scan may be time-consuming.

In 1956 an LP code for the IBM 704 computer appeared. The IBM 704 had 4 to 8K 36-bit words of memory, plus tapes for backing storage. Tapes were ideally suited to the algorithm as described so far, since the original matrix could be stored on one or more tapes, while the elementary transformations could also be created on one or more tapes. Operation 3 read the tapes forward (and was called FTRAN, for forward transformation), a new transformation was added to the end, and the tapes could be read backwards for operation 1 (called BTRAN). The size of problem that could be solved with this system was now up to 256 rows.

From 1960 to 1966, LP/90 and C-E-I-R LP/90/94 were written for the more powerful IBM 7090 and 7094 computers, which had 32K 36-bit words, hardware floating point, and tapes. These codes were very sophisticated and could solve problems of up to 1023 rows.[3] With a sufficient number of tapes there was a good match between the floating-point unit and the I/O devices. (It also made LP visually more interesting than it is now.)

When IBM moved to the dramatically different System/360* and System/370* architectures, with much larger memory and disks, the first IBM LP code (MPS/360) did not take real advantage of the new hardware. In the UMPIRE code for the Univac 1108, the authors[4] did take advantage of the fact that disks were random access, and that updating therefore did not have to occur only at the end of the file. Use of this knowledge allowed elementary transformations with fewer elements, with a corresponding increase in speed. These ideas were subsequently incorporated in IBM's Mathematical Programming System Extended/370 (MPSX/370).[5] By the mid-1980s, with use of large real and virtual memory, the only change in overall design was that virtual memory became the "slow" device taking the place of disks, whereas the cache became the high-speed memory.

In summary, the simplex method has evolved with computers and has, sometimes tardily, taken advantage of each advance in computer architecture.[6]

Restatement of the simplex algorithm

This restatement of the algorithm is not completely general—it only deals with the case where all variables have zero lower bounds and infinite upper bounds. However, it is sufficient for understanding the remainder of this paper. For brevity we also omit discussions of some preliminaries. These include "presolving" to remove redundancies from the LP model and "crashing" an initial basis.[7] The steps involved often allow a "good" basis to be chosen, which can reduce the number of iterations needed by a substantial amount.

The steps of the simplex method (and their associated jargon) may be described as follows:

0. (INVERT and Initialization) Choose (or read in) a basis B from the columns of A and define x_B and x_N, the vectors of basic and nonbasic variables. Set $x_N = 0$. Compute the product form inverse of B. In practice we factorize (some permutation of) B as LU where L is lower triangular and U is upper triangular. The product form of their inverses may be written trivially as:

$$B^{-1} = U_1^{-1}U_2^{-1} \cdots U_m^{-1}L_m^{-1} \cdots L_1^{-1}$$

where each of the L_k^{-1} and U_k^{-1} are elementary transformation vectors, of the same type as E_k, constructed from the columns of L and U. For convenience we will usually continue to write:

$$B^{-1} = E_l E_{l-1} \cdots E_2 E_1$$

After completing this factorization, historically known as INVERT, we may compute:

$$x_B = \beta = B^{-1}b$$

1. (FORMC) If the solution is feasible (x_B nonnegative), we wish to minimize the true objective $c^{\top}x$ and form a vector containing the cost of each basic variable: $q^{\top} = c_B^{\top}$. If the solution is not feasible, in order to move toward feasibility we try to minimize the "sum of infeasibilities." To accomplish this we try to increase all basic variables below zero by giving them "costs" of -1.0 and by giving feasible variables a cost of zero. We then define $q^{\top}$ using these costs instead of $c_B^{\top}$.

2. (BTRAN) Apply the transformations in reverse order:

$$\pi^{\top} = -q^{\top}E_l \cdots E_2 E_1$$

Note that these are the negative of the values described earlier.

3. (PRICE) Compute the reduced costs for some or all of the nonbasic variables:

$$d_j = c_j + \pi^T a_{.j} \qquad (a_{.j} \in N)$$

where $a_{.j}$ denotes column j of the matrix. If the problem is infeasible, we omit the c_j term. Choose one j, say g, among those with negative reduced costs. If there are no negative reduced costs, we have finished, with an optimal solution if the problem is feasible, or we have proved that there is no feasible solution otherwise.

4. (FTRAN) Apply the transformations in forward order:

$$\alpha = B^{-1}a_{.g} = E_l \cdots E_2 E_1 a_{.g}$$

5. (CHUZR) Choose which variable will leave the basis because it reaches zero first. When the solution is feasible, we may describe this in terms of a ratio test of finding:

$$\theta = \frac{\beta_p}{\alpha_p} = \min_{\alpha_i > 0} \frac{\beta_i}{\alpha_i}$$

where p becomes the pivot row and x_{Bp} leaves the basis. If there is no such row, the problem is unbounded. When infeasible, the test is similar but somewhat more complicated.

6. (UPDATE) The factorization of B^{-1} is modified. In the product form of the simplex algorithm, this involves creating a new eta E_{l+1} so that

$$\bar{B}^{-1} = E_{l+1} E_l \cdots E_2 E_1$$

where (letting $p_{l+1} = p$) the nonunit column of E_{l+1} is defined by

$$\eta_i^{(l+1)} = \frac{-\alpha_i}{\alpha_p} \qquad (i \neq p)$$

$$\eta_p^{(l+1)} = \frac{1}{\alpha_p}$$

The new values of the basic variables are then:

$$\bar{x}_B = \bar{\beta} = E_{l+1}\beta$$

More sophisticated implementations of the simplex method may directly update the triangular factors L and U.

We then repeat from step 1. Periodically we must INVERT (refactorize), or repeat from step 0, for speed (since the representation of B^{-1} is becoming longer and longer) and to avoid the buildup of round-off error.

Exploiting the 3090 Vector Facility

Up until the late 1980s, the way to a fast simplex implementation was to reduce the amount of data moved into high-speed memory and get the data there as fast as possible by minimizing disk I/O times or cache misses. Once the data were in high-speed memory, the time for the floating-point operations dominated the algorithm, so reducing the number of such operations was of paramount concern. This situation changed with the introduction of the IBM 3090* Vector Facility, as some floating-point operations became more important than others. This comparison is particularly apparent when we consider the pricing operation.

There are many ways to choose a variable with a negative reduced cost. Variants (termed *pricing strategies*) include:

1. Passing through the entire matrix N, computing d_j, and choosing the column with the most profitable reduced cost (called "full pricing")
2. Only scanning part of the matrix to obtain some reasonably profitable reduced costs ("partial pricing")
3. As above for 1 or 2 but selecting several columns at a time ("multiple pricing"). All of these could then be updated simultaneously, which involved only one pass through the transformation data.
4. Using the Devex method (from the Latin "steepest") due to Harris,[8] which uses full pricing but requires additional work to compute weights w_j and to select the column on the basis of the best *weighted* reduced cost d_j/w_j

Method 3 was preferred from the mid-1950s to the mid-1980s, as it allowed several iterations for a partial pass through the matrix and one pass

Table 1 Sparse inner product speedup

Nonzero Elements per Column	Vector/Scalar Ratio
10	0.89
20	1.44
30	1.85
40	2.18
50	2.41
60	2.61
70	2.77
80	2.91
100	3.35

through the transformation data. Method 4 typically took fewer iterations but more time on most models, as the iterations were considerably more expensive. With changes in computer architecture, the advantage of being able to make several updates in one pass through the transformation data declined. For a single update better use could be made of the registers, and the balance between cache overhead and floating-point operations was reasonable.

Let us re-examine work involved in pricing in more detail, writing the core computational step in full as:

$$d_j = c_j + \sum_{i=1}^{m} \pi_i a_{ij} \qquad (a_{.j} \in N) \tag{13}$$

In general, nearly all of the a_{ij} are zero—normally less than ten in any column will be nonzero—and only the nonzero elements are stored in a data structure consisting of the following arrays:

- Elements—a 64-bit array of floating-point numbers giving the nonzero elements. The size of this structure is the total number of nonzeros in the matrix.
- Indices—an integer array of similar size giving the row of the matrix on which the corresponding element lies
- Start—an integer array giving the position in the above two arrays of the first element and row index for each column
- Number—an integer array giving the number of nonzero elements in each column

The double loop that can be defined for computing (13), and then using d_j for several columns, can be translated into FORTRAN as:

```
      DO Column = Start,End
C       only do if not basic
         IF(......) THEN
           Inner_Product=0.0D0
           DO I = Start(Column),Start(Column) +
                  Number(Column) -1
             Row_index = Indices(I)
             Inner_Product = Inner_Product +
             Pi(Row_index)*Elements(I)
           ENDDO
C       use Inner_Product
           ......
         ENDIF
      ENDDO
```

The FORTRAN compiler for the 3090 can vectorize the inner loop as:

```
C       load number of nonzero elements
        L      R1,Number_of_Nonzeros
C       enter smaller of section size and number
C       remaining to be done
LOOP   VLVCU R1
C       row numbers where there are nonzero elements
       VL     V1,Indices
C       load corresponding values from dense PI
       VLID  V2,V1,Pi
C       multiply and accumulate inner product
C       using the nonzero elements of column
       VMCD  V4,V2,Elements
C       back to loop if any more elements
       BP     LOOP
```

The speed of this vectorization is approximately four cycles per element as compared to approximately 18 cycles per element for the corresponding scalar coding. This comparison looks promising until we find that there is a fixed overhead of approximately 180 cycles for the vector loop as compared to 15 cycles for the scalar loop. For different numbers of nonzero elements we obtain the results in Table 1.

A speedup of two is not seen until a column has more than 30 elements, which is not common. The break-even point is between 11 and 12.

Let us examine a medium-sized, fairly dense LP which, after "reduction" to remove redundancies, has 3526 rows, 9625 variables, and 70 560 elements, giving an average of 7.33 nonzero elements per column (the "slack" variables are not included in the statistics). No column has more than 25 entries, and we can expect no significant increase in speed if we vectorize the inner loops.

The 3090 Vector Facility has a relatively large startup time, and so we need to use it for large numbers of homogeneous operations. There are no large numbers in the first column of Table 2, but there *are* plenty of large numbers. For example, there are 588 columns with exactly seven elements.

The solution to using the Vector Facility efficiently is to reorder the matrix so that all columns with the same number of entries are grouped together and then vectorize the outer loop of the inner product loop, rather than the inner loop. If we have T columns with exactly K nonzero elements, we can store this as two dense blocks—Z_{kt} for $k = 1, \cdots, K, t = 1, \cdots, T$ for the elements and a similar one, I_{kt}, for the corresponding row indices.

With this scheme the FORTRAN loop above can be restructured for each such block as:

```
DO k = 1 , K
  DO t = 1 , T
    Row_index = I(t,k)
    Inner_Product(t) = Inner_Product(t) +
                       Pi(Row_index)*Elements(t,k)
  ENDDO
ENDDO
```

the inner loop of which the FORTRAN compiler can vectorize (for the kth nonzero in each of the columns) as:

1. LOAD the vector of indices

   ```
   VL    V1,Indices
   ```

   to get $\langle i_{k1}, i_{k2}, \cdots, i_{kT} \rangle$

2. Use the LOAD INDIRECT vector instruction

   ```
   VLID  V2,V1,Pi
   ```

   to get $PI = \langle \cdots \pi_{i_{k1}} \cdots \rangle$

3. Denoting the kth row of Z (in memory) as

   $Z\ ROW = \langle a_{i_{k1}}, a_{i_{k2}}, \cdots, a_{i_{kT}} \rangle$

   use the MULTIPLY AND ADD instruction

   ```
   VMAD  V4,V2,Elements
   ```

   to add the element-by-element product of this vector with PI to update the reduced costs:

   $$DJ \leftarrow DJ + PI * Z\ ROW$$

Table 2 Frequency of nonzeros per column

Number of Nonzeros in Column	Number of Structural Columns
1	569
2	1,438
3	938
4	84
5	520
6	124
7	588
8	2,656
9	1,364
10	84
11	188
. . . Vector Break Even . . .	
12	100
13	56
14	156
15	204
16	80
17	84
18	8
20	4
21	8
22	40
23	216
24	88
25	24

To use stride one and avoid cache misses, these two blocks of data are stored row-wise. We also sort the blocks so that all of the nonbasic columns are at the beginning. This ordering means that we need not look at the basic columns. The only drawback is that for each iteration we may have to modify two blocks of the matrix, one to adjust the block for the incoming column and one for the outgoing. For problems of any size this overhead is trivial.

One obvious question is whether we can do this sorting for every matrix. If we decide to sort only for columns with less than a certain number of nonzero entries, as problems become larger we can achieve a success rate approaching 100 percent. All columns with enough nonzeros vectorize in the normal (unblocked) way; hence, if we arbitrarily set 30 as an acceptable number for a vector length, the number of columns that fall in a block with too few columns must be less than 30×30. In practice we find that this reordering of the matrix for vectorization is worthwhile for all matrices with more than a few hundred columns.

Table 3 Example with scalar PRICE

Method	PRICE Work per Iteration	Non-PRICE Work per Iteration	Number of Iterations	Total Work
Standard	0.04	0.24	20,000	5,600
Devex	0.32	0.28	10,000	6,000

Table 4 Example with vectorized PRICE

Method	PRICE Work per Iteration	Non-PRICE Work per Iteration	Number of Iterations	Total Work
Standard	0.01	0.24	20,000	5,000
Devex	0.08	0.28	10,000	3,600

With this reordering, we achieve a performance improvement of a factor of four, which is close to the asymptotic improvement 18/4. This improvement only applies to PRICE. For the other time-consuming operations, we are unable to do much reordering, partly because of the order dependency of the operations and partly because these structures are continually changing (but see Eldersveld and Saunders[9] for a quite different approach).

Now we must consider the effect of being able to improve this particular portion of the code by a significant amount. As we mentioned, the conventional wisdom (based on tests when IBM's MPSX/370 was being developed in the early 1970s) was that partial pricing was preferable, even though it might take more iterations. To see why, consider the effect of two different pricing strategies (in Table 3) on a hypothetical problem, using an arbitrary unit of work (very approximately one million floating-point operations). The economy of the Devex method in number of iterations is clear even though the time is longer.

Now let us see what is the effect of being able to do some floating-point operations (especially in PRICE) four times as fast. The new times are shown in Table 4. They represent a significant gain.

The point to be made is that the gain came about after a multistage process of examining what a new architecture has to offer:

1. Identify which parts of the current implementation of the algorithm could expect a gain.
2. Identify which parts of the algorithm could gain if the data structures were modified.
3. Identify any modifications to the algorithm, or even totally different algorithms, which are more suited to the new architecture.
4. Balance the last three points to obtain the maximum improvement.

Much the same general approach is advocated by Zenios and Mulvey.[10] This discussion has concerned the simplex algorithm and its modifications, but interior point algorithms currently seem able to make more extensive use of many of these architectural changes.[11]

The initial work on vectorizing PRICE was done for MPSX/370. On easy problems (where the scalar code had given satisfactory performance) there was little speedup, whereas the best result achieved reduced the running time on one problem from 574 minutes to 150 minutes, a factor of 3.8. An experimental IBM Yorktown code gave us the opportunity to repeat the steps outlined above with a new design so that every aspect could be considered and every opportunity taken to vectorize the code. This code evolved into the simplex component of the IBM Optimization Subroutine Library (OSL). Table 5 (part of Table 1 of Forrest and Tomlin[11]) summarizes some of the resulting improvements compared to scalar MPSX code (times are given in minutes on a 3090).

The last entry in Table 5 is the problem we used as our earlier example but showing the unreduced dimensions. Also note that the speedup ratios are better for larger, denser problems. These results were obtained in 1987 and 1988; since then the OSL simplex algorithm has been further improved.

Another opportunity afforded by improved PRICE-type calculations is in the implementation of another variant of the simplex algorithm that is even more expensive on scalar machines—the *dual simplex* algorithm. This algorithm makes even more extensive use of pricing calculations and is well-suited to many problems. Using the new data structures, we obtain the results in Table 6 for the 4422 (original) rowed problem. (All times are scaled to 3090E minutes, except for the RISC System/6000* time.)

Table 5 Speed improvement of vectorized research code over scalar MPSX/370

Rows	Columns	Nonzeros	Iterations	Time	Speedup
147	2,655	14,005	976	0.10	1.6
1,152	2,763	10,941	1,045	0.05	1.0
4,981	6,221	34,895	5,988	1.30	2.9
930	3,523	14,173	1,562	0.12	1.3
398	2,750	11,334	1,753	0.17	1.5
3,829	8,216	108,968	7,418	3.51	3.3
822	1,571	11,127	2,699	0.40	3.1
3,526	9,625	74,090	13,273	7.92	4.0
1,442	3,652	43,220	6,092	3.71	5.0
5,564	6,181	46,578	10,018	4.90	2.8
2,357	11,004	128,286	21,523	17.83	4.0
2,942	11,717	99,121	16,679	9.65	3.7
4,422	6,711	110,342	42,580	47.21	12.2

RISC and the RISC System/6000

After several years spent writing a code from scratch to take advantage of the 3090 vector architecture, we were then faced with the new RISC architecture in the RISC System/6000. To understand the consequences of this architecture for the simplex method, we need to look at the comparative characteristics of some important classes of operations in terms of machine cycles:

- Dense processing for simple (one-dimensional) arrays. The critical (composite) operation here is LOAD – MULTIPLY – ADD. The relevant statistics are:
  - 3090 scalar—14 cycles
  - 3090 vector—2 cycles (asymptotic)
  - RISC System/6000—2 cycles
- Sparse processing for simple sparse/dense array operations. The critical sequence of instructions here is LOAD (an index) – LOAD (a value) – MULTIPLY – ADD. The statistics are:
  - 3090 scalar—18 cycles
  - 3090 vector—4 cycles (asymptotic)
  - RISC System/6000—4 cycles
- For sparse work the floating operations can be free. The bottleneck is in getting the numbers to and from registers. This situation suggests that extra work should be done once the numbers are in the registers.
- There is no special advantage to restructuring the matrix, in contrast to the Vector Facility.
- Cache misses are considerably more expensive.

In an earlier paper, we stated that "One lesson from this (vectorization) exercise is that it may be advantageous to search for other column selection schemes which are better than Devex even if they need more computation, including schemes which were considered and rejected some years ago."[11]

Table 6 Solution times for 4422 row problem

System	Time	Iterations
MPSX Version 1	517	302,357
MPSX Version 2	83	48,858
OSL Primal	25	36,050
OSL Dual	11	11,410
On RISC System/6000 520		
OSL Dual	32	11,449

On the RISC System/6000, it is possible to do extra floating-point operations at small (or sometimes no) extra cost, and some of these ideas looked even more promising. In particular, taking two inner products simultaneously turns out to be only 20 percent more expensive than one. At the suggestion of Don Goldfarb, we experimented with the steepest-edge pricing, to which Devex pricing can be regarded as an approximation.[12] Although this approach did not lead to universal success, it did lead to a substantial reduction in iterations for many problems. Tables 7–9 show the results, on a set of nontrivial test problems, of using the steepest-edge method in comparison to several other strategies on the mainframe computer and in comparison to the standard Harris Devex method on a RISC machine (all times exclude input).

Table 7 Test problem characteristics

Model	25FV47	PILOTS	DFL001
Original			
Rows	821	1,441	6,071
Columns	1,571	3,652	12,230
Nonzeros	10,400	43,167	35,632
Reduced			
Rows	715	1,374	4,840
Columns	1,484	3,361	10,999
Nonzeros	9,994	40,757	33,146

Table 8 Times (in seconds) on a 3090S with Vector Facility

Model	25FV47	PILOTS	DFL001
MPSX (no Devex)			
Iterations	3,681	8,432	>600,000
Solution time	47.5	892.8	>54,000.0
OSL—Full pricing			
Iterations	2,917	11,165	964,893
Solution time	19.6	338.9	31,580.4
OSL—Harris Devex			
Iterations	2,183	5,483	50,327
Solution time	15.3	176.4	1,342.4
OSL—Steepest edge			
Iterations	1,361	3,376	21,117
Solution time	14.1	148.3	740.9

Table 9 Times (in seconds) for a RISC System/6000 Model 540

Model	25FV47	PILOTS	DFL001
OSL—Harris Devex			
Iterations	2,191	5,211	43,012
Solution time	26.1	273.2	3,829.9
OSL—Steepest edge			
Iterations	1,184	3,297	17,343
Solution time	16.6	201.8	1,514.6

Returning to the 4422-row problem, and applying these steepest-edge ideas to the *dual* algorithm, we obtained a new "best time" of 7.5 minutes to solve it in 4827 iterations on the RISC System/6000 Model 540 (see Table 6).

Stability

One of the objectives of the IBM Research simplex code which became part of OSL was to be as fast as possible. The other main objective was to solve *all* problems presented to the code. With finite precision arithmetic this last objective is impossible to achieve, but it remains a worthy target. This section describes the steps taken to approach this target as closely as possible and, if the target is unreachable for any reason, to fail in as controlled a manner as possible.

First let us discuss some of the ways in which the simplex algorithm can fail, bearing in mind that in every case the worst possibility is that the algorithm does not recognize that it is failing and never terminates. One well-known possibility is that the algorithm "cycles." This outcome is possible if the θ computed in step 5 is zero for several iterations, because then we are not making any progress in the objective, and it is possible to bring in and throw out the same sets of variables from the basis *ad infinitum*. Zero moves occur very frequently, and in such cases the problem is termed "degenerate." When this happens, the simplex algorithm is likely to take more iterations than usual, but luckily there are ways to avoid actual cycling in these cases. It turns out that some of the ideas discussed below, as well as the Devex method discussed earlier, help to reduce the number of iterations.

A more serious problem is oscillation. At step 0 the simplex algorithm first finds a factorization. The steps in finding LU such that $B = LU$ are similar to those involved in solving simultaneous equations by Gaussian elimination, and we may find that one equation is a linear combination of others. In this case we find ourselves attempting to divide by a zero pivot, which is illegal. Other forms of ill-conditioning may also lead to unacceptable pivots ("singularity") in INVERT. In this case the normal practice (as in MPSX/370) is to discard the offending column, replacing it by the "slack" column for that constraint, or an "artificial" variable if the constraint was an equality. This may change the objective value or make the solution infeasible when it had been feasible. The danger is that at one refactorization the problem is feasible so that the algorithm is in "phase 2"; then steps 1 to 6 are repeated several times. At each iteration the accuracy of the factorization will decrease. This decrease will not be recognized, and some bad iterations may occur. At the next refactorization the basis is discovered to be "singular," a column is replaced, and the problem becomes infeasible so that we are back in "phase 1." The algorithm now tries to become feasible, succeeds, but may or may not have a solution that is as good as the one it had before. On numerically difficult problems such a situation could and did happen with MPSX/370. Even if the new basis is not singular, the solution (β) used in step 5 may have become inaccurate, so that after refactorization the problem is slightly infeasible.

What can be done about such situations? The answer is many things, each of which helps, but where the cumulative effect is important, as the ideas often work better together than in isolation.

First let us look at the accuracy problem in isolation. Prevention being the best cure, we seek to make the operations as accurate as possible and then check their accuracy. The first step is to have the most accurate factorization that is possible, as this will be the keystone of all our efforts. The "numerically correct" approach to simultaneous equations will attempt to do two things:

1. Look for ways to implicitly reorder the rows to obtain a near triangular form, reduce the amount of work, and minimize the number of nonzeros. For the LP factorization this approach reduces the number of floating-point operations needed to transform B into U—which helps reduce the buildup of error.
2. Choose large elements as pivots to use in eliminating others in that row or column.

The best-known method to achieve these goals is called the "dynamic Markowitz[13] method"—*dynamic* because we are continually looking at the sizes of the elements. MPSX/370 and many other systems were designed for smaller memories and used a bit map representation of the basis (a 1 where there was an element and 0 where there was not) or some other symbolic representation. This allowed them to attempt to reduce the number of operations, but gave no control over the size of the pivots. If these pivots were subsequently found to be too small, it was necessary to resort to a fairly primitive method to continue. With current large memories and work done by Reid[14] we can implement the full dynamic Markowitz method. It is essentially identical to the one now in ESSL—the Engineering and Scientific Subroutine Library.[15]

Given a stable factorization, then in the worst case we could refactorize every iteration, assuming that we can recognize the buildup of inaccuracy. However, the update described in step 6 is known to be unstable. The most stable update is due to Bartels and Golub.[16] This method was not practicable before the days of large memories, and the authors implemented a more practicable method.[4] The latter lacks *a priori* theoretical stability but has greater stability in practice than the product form update *and* allows for a check on stability. If the check fails, there is no recovery except for a refactorization, but as it is more suited to most computer architectures than the Bartels-Golub method, it is used in OSL. We can accept the occasional premature refactorization; it is the average time on which we must judge performance. In OSL the stability checks were made considerably tighter than in MPSX/370.

After tolerably accurate basis factors have been obtained, the next step is modifying the algorithm itself to improve overall accuracy. Previously we mentioned the improvement in performance due to using the Devex ideas from Harris,[8] and columns chosen using this form of pricing do seem to be "better" for an accurate sequence of iterations. But the same paper contained ideas on pivot choice that were widely ignored. Pursuing some of them, we see that in the textbook nondegenerate case we only have a single pivot choice in step 5. However, most loss of accuracy in iterating comes from pivots where β_p is close to zero. To see why this should be so let us examine the following case:

Row	Alpha	Beta
1	1.0	0.0
2	0.0000001	0.0

Here the minimum value of θ is achieved for both rows, and we may choose the larger pivot on row 1, which will tend to give a more stable pivot in the UPDATE step than using the smaller value on row 2. But in practice β values are rarely exactly 0, sometimes because of buildup of rounding error and often because the data have been given with limited accuracy. For example, a value of 1/3 will normally be given as 0.333333. Thus, it is common to have β values of 10^{-8} to 10^{-10}.

Now in the following case:

Row	Alpha	Beta
1	1.0	1.0
2	0.0000001	0.000000099

the situation is not as clear. The minimum is achieved for row 2 with θ at 0.99, but this choice of pivot would be bad. Alternatively, we could simply reject all values no greater than 10^{-7}, but that would lead to even worse problems. Some re-examination is required. So far, we have defined a variable as "infeasible" if it is less than zero. However, since in practice we are working to fixed accuracy, there must always be a feasibility tolerance (the default value in OSL is 10^{-8}), and "infeasible" is defined as below minus this value. Some tolerance is necessary from an elementary numerical standpoint, but the slightly higher values used in LP are due to the inherent inaccuracy of the data referred to above.

Luckily, the solution to our pivoting problem lies in using the feasibility tolerance constructively. Suppose we pivot on row 2, then β_1 becomes 0.01, which is feasible, but if we pivot on row 1, then β_2 becomes 0.000000099-0.0000001, or -10^{-9}, which is also feasible, given our definition of feasibility. Thus, row 1 gives an acceptable and stable pivot. The problem normally arises when all of α, β, and θ are small, so it is likely that $\beta - \theta\alpha$ would not be very negative. In general then, we have a set of pivot choices with different θ all valid with respect to the user-defined feasibility tolerance. This set gives us more flexibility and allows us to choose larger pivots. Having larger pivots tends in turn to make the Devex weights for column selection more accurate. It can also allow the algorithm to escape degeneracy more quickly.

Unhappily, while these ideas help, we can still construct situations where difficulties occur. Having allowed small negative β values, we must expect them actually to occur. Most of them will not cause any problems, but from time to time we will have a situation similar to the following:

Row	Alpha	Beta
1	1.0	0.005
2	0.0001	−0.000000001

If we now pivot on row 1, β_2 becomes −0.000005001. This condition causes us to declare the problem to be back in an infeasible state. If we pivot on row 2, our θ will be negative, and the objective function will move in the wrong direction. In both cases we are allowing the state of the problem to get worse, thus bringing back the threat of oscillation.

The solution here was suggested by Gill et al.,[17] although we have not followed all of their ideas. The value of the variable pivoting on row 2 is currently -10^{-9}, and for this situation to occur the variable will normally be within the feasibility tolerance of zero. To overcome our problem we will leave it at that value and take it out of the basis, but instead of setting it to zero, we will leave it at its current nonzero value. Such a variable is called *superbasic* and naturally occurs in nonlinear programming. Now we make a move taking $\theta = 0$. It is still possible to make such a move for a very small α, but our changes to infeasibility are becoming smaller and smaller.

We have reduced the stability problem by these techniques, but not eliminated it. We still have the possibility of oscillating between feasibility and infeasibility and making no overall progress. The basic problem can be posed as a question—how would we rank the state of the solutions at three consecutive iterations, where the objective and sum of infeasibilities are as follows?

	Objective	Sum of infeasibilities
1	1000000.0	0.0
2	999999.0	0.00001
3	1000001.0	0.0

Since we are minimizing the objective, then obviously 3 is worse than 1, but is it the step from 1 to 2 or the one from 2 to 3 that is bad? The problem is that we have been differentiating between "phase 1" when the problem is infeasible and "phase 2" when it is feasible, but not giving any weights to the two phases. This situation brings us to the idea of a *composite* objective function.[18] This function exists (but is rarely used) in MPSX/370 and has been used in some other commercial codes. The initial idea was to work toward optimality while still infeasible. The only difference between the phase 1 and 2 algorithm is in the creation of a pricing vector, and we can use some combination of the two. If we use w_F as a weight for the feasible objective and w_I for the infeasible objective, it is only the ratio that matters.

When the problem is feasible there is no difference in algorithms, as the infeasible objective is empty. When the problem is infeasible there is the possibility that the algorithm may incorrectly terminate, because too much weight has been given to the feasible objective. It is easy to check for, and then the weights may be adjusted and the algorithm continued. However, it is considerably more efficient to use some heuristic ideas to modify the weights before this happens. OSL periodically adjusts the weights (so that w_F decreases if it changes at all) in an attempt to keep the best balance. Normally the weight given to the infeasible objective is several orders of magnitude greater than the weight given to the feasible one.

Returning to our example, we can now answer the question if we know the weights. If w_F/w_I is greater than 10^{-11}, then 1 to 2 was good (and bad otherwise), whereas if w_F/w_I is less than 5×10^{-12}, then 2 to 3 was good. If weights have been changed between 1 and 2, we would use the final weights for the test, and similarly for 2 and 3.

With given weights we now also know whether a refactorization is "worse" than the previous one. One simple idea implemented in OSL is a "history" array. This small array keeps a history of each iteration—which variables entered and left the basis. Unlike MPSX/370, which always continues iterating, OSL always tries to go in the correct direction, so if it appears that a refactorization is worse (or if the basis was singular), it strips off a certain number of iterations, using the history region, and tries again. This is repeated if necessary. There are two final possibilities:

1. The new basis is not the same as the original one. In this case we are making progress, and we continue, refactorizing more frequently than before, as it looks as if we were being too ambitious.
2. The first iteration was bad, so we are back to the original basis. In this case we mark the variable that entered on the first iteration as "dangerous" and carry on with great caution. Eventually we must make one good iteration, or all variables will have been rejected. In this case the code declares semifailure and returns a status of "optimal," but with some variables excluded from consideration.

We may use the idea of a composite objective function in a more constructive way to avoid problems, rather than just to detect them. If we deliberately allow a problem to become infeasible after being feasible, we can extend the Devex ideas to give us more flexibility and stability. The ideas we have just discussed would still force us to pivot on row 2 in the following case:

Row	Alpha	Beta
1	1.0	1.0
2	0.0000001	−0.000000001

and we know that we would prefer to pivot on row 1. But now suppose we are feasible, w_F is 10^{-9}, w_I is 1, *and* the gradient or reduced cost of the incoming variable is -200.0. Now pivoting on row 1 will give us a new infeasibility of 0.000000101. (To be absolutely accurate it will be 10^{-8} [the infeasibility tolerance] less than this.) But the objective will decrease by θ times the reduced cost, or 200.0, so the weighted objective function will decrease by $0.0000002 - 0.0000001$; thus we are still improving. Hence, we have yet more flexibility to choose larger pivots when we have a numeric penalty for the idea of infeasibility rather than an absolute feasible/infeasible switch. We may still encounter problems if w_F and d_j are both small. Luckily, if all of the feasible d_j are small, we are unlikely to have needed a small w_F. Since the pricing step tries to pick large reduced costs, the main area of danger is at the very end of the simplex method as the best reduced cost approaches zero. We note that LP models that have a basis in the real world seem to behave better in the neighborhood of the optimum than at some arbitrary point in the algorithm.

The approach to stability we have described may be summarized by the following points:

- Choose the best possible factorization process as the basis for the implementation.
- Make sure the algorithm knows if inaccuracies are likely to occur.
- Put in layers of strategies to encourage good iterations.
- Put in multiple layers of safety nets so that the algorithm can continue, even if slowly for some iterations, when difficulties are encountered.

Never allow the possibility of the algorithm falling into a "black hole."
- Use strategies and algorithms that reinforce one another.

A final critical point in the development of an LP code is exhaustive testing. The range of test problems available is much wider now, but with modern computing power, OSL can be tested on more problems in one day than MPSX/370 could be tested on in a month.

Conclusion

Perhaps because of its rich history, the simplex method offers a multitude of opportunities for exploiting new computer architectures and numerical computing techniques. The size of problems being generated and presented for solution has grown enormously in the last few years. This growth has presented considerable challenges, not only in building LP codes with the requisite speed, but in robustness. With use of the techniques described in this paper, the simplex method, as implemented in OSL, has had considerable success in meeting these challenges.

Acknowledgments

The authors are indebted to Don Goldfarb and the anonymous referees for their constructive criticisms of the first draft of this paper.

*Trademark or registered trademark of International Business Machines Corporation.

Cited references

1. H. P. Williams, *Model Building in Mathematical Programming*, John Wiley & Sons, Inc., New York (1978).
2. G. B. Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ (1963).
3. E. M. L. Beale, *Mathematical Programming in Practice*, Wiley, New York, and Pitmans, London (1968).
4. J. J. H. Forrest and J. A. Tomlin, "Updating Triangular Factors of the Basis to Maintain Sparsity in the Product Form Simplex Method," *Mathematical Programming* **2**, 263–278 (1972).
5. M. Benichou, J. M. Gauthier, G. Hentges, and G. Ribière, "The Efficient Solution of Large-Scale Linear Programming Problems—Some Algorithmic Techniques and Computational Results," *Mathematical Programming* **13**, 280–322 (1977).
6. J. A. Tomlin, "The Influences of Algorithmic and Hardware Developments on Computational Mathematical Programming," in *Mathematical Programming: Recent Developments and Applications*, M. Iri and K. Tanabe, Editors, Kluwer Academic Publishers, Tokyo (1989), pp. 159–175.
7. W. Orchard-Hays, *Advanced Linear Programming Computing Techniques*, McGraw-Hill Book Co., Inc., New York (1968).
8. P. M. J. Harris, "Pivot Selection Methods of the Devex LP Code, *Mathematical Programming* **5**, 1–28 (1973).
9. S. K. Eldersveld and M. A. Saunders, *A Block-LU Update for Large-Scale Linear Programming*, Technical Report SOL 90-2, Department of Operations Research, Stanford University, CA (1990). To appear in *SIAM Journal of Matrix Analysis*.
10. S. A. Zenios and J. M. Mulvey, "Vectorization and Multitasking of Nonlinear Network Programming Algorithms," *Mathematical Programming* **42**, 449–470 (1988).
11. J. J. H. Forrest and J. A. Tomlin, "Vector Processing in Simplex and Interior Methods for Linear Programming," *Annals of Operations Research* **22**, 71–100 (1990).
12. D. Goldfarb and J. K. Reid, "A Practicable Steepest-Edge Algorithm," *Mathematical Programming* **12**, 361–371 (1977).
13. H. M. Markowitz, "The Elimination Form of the Inverse and Its Application to Linear Programming," *Management Science* **3**, 255–269 (1957).
14. J. K. Reid, *FORTRAN Subroutines for Handling Sparse Linear Programming Bases*, Report AERE-R.8269, Harwell Laboratory, Oxfordshire, England (1976).
15. *Engineering and Scientific Subroutine Library Guide and Reference*, SC23-0184-4, IBM Corporation (1990); available through IBM branch offices.
16. R. H. Bartels and G. H. Golub, "The Simplex Method of Linear Programming Using the *LU* Decomposition," *Communications of the ACM* **12**, 266–268 (1969).
17. P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright, "A Practical Anti-Cycling Procedure for Linearly Constrained Optimization," *Mathematical Programming* **45**, 437–474 (1989).
18. P. Wolfe, "The Composite Simplex Algorithm," *SIAM Review* **7**, 42–54 (1965).

General references

E. M. L. Beale, "The Evolution of Mathematical Programming Systems," *Journal of the Operational Research Society* **36**, 357–366 (1985).

IBM RISC System/6000 Performance Tuning for Numerically Intensive FORTRAN and C Programs, GG24-3611, IBM Corporation (1990); available through IBM branch offices.

IBM System/370 Vector Operations, SA22-7125-3, IBM Corporation (1988); available through IBM branch offices.

MPSX/370 Version 2 Users Guide, SH19-6552-0, IBM Corporation (1988); available through IBM branch offices.

Optimization Subroutine Library Guide and Reference, SC23-0519-2, IBM Corporation (1991); available through IBM branch offices.

W. Orchard-Hays, "History of Mathematical Programming Systems," in *Design and Implementation of Optimization Software*, H. J. Greenberg, Editor, Sijthoff and Noordhoff, The Netherlands (1978), pp. 1–26.

Accepted for publication September 13, 1991.

John J. H. Forrest *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598-0218.* Mr. Forrest is a research staff member at the Thomas J. Watson Research Center. He has overall responsibility for the contributions of IBM Research to the Optimization Subroutine Library. In this position he was responsible for much of the design of OSL and for the simplex and mixed-integer portions of the library. He has helped to develop several other mathematical programming packages, including UMPIRE and Sciconic. Mr. Forrest has a B.A. from Oxford University and an M.S. from the University of California at Berkeley.

John A. Tomlin *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120-6099.* Dr. Tomlin is a research staff member at the Almaden Research Center. He has been with IBM since 1987. He gained a B.Sc. (honors) in 1963 and a Ph.D. in mathematics in 1967, both from the University of Adelaide, South Australia. Since 1968 he has been involved in research and development in mathematical programming systems and their applications, first with Scicon, Ltd., London, then at Stanford University, and subsequently at Ketron, Inc. In 1970 he was awarded an IBM post-doctoral fellowship at Stanford University. He is a member of the Association for Computing Machinery and the Operations Research Society of America and is a charter member of the Mathematical Programming Society.

Reprint Order No. G321-5457.