Implementing interior point linear programming methods in the Optimization Subroutine Library

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

This paper discusses the implementation of interior point (barrier) methods for linear programming within the framework of the IBM Optimization Subroutine Library. This class of methods uses quite different computational kernels than the traditional simplex method. In particular, the matrices we must deal with are symmetric and, although still sparse, are considerably denser than those assumed in simplex implementations. Severe rank deficiency must also be accommodated, making it difficult to use off-the-shelf library routines. These features have particular implications for the exploitation of the newer IBM machine architectural features. In particular, interior methods can benefit greatly from use of vector architectures on the IBM 3090™ series computers and "super-scalar" processing on the RISC System/6000™ series.

The well-known simplex method for linear programming (LP) is nearly as old as LP itself, developed by Dantzig in 1947.[1] It is perhaps a coincidence that the forerunner of today's interior point methods was devised at about the same time as implementations of the simplex method, which could handle significant problems[2] in the mid-1950s. This forerunner was the logarithmic barrier method of Frisch.[3] Despite subsequent interest in barrier methods for nonlinear programming,[4] they were thought to be hopelessly impractical as competitors to the simplex method for LP. After all, these methods involved *non*-linearizing a linear problem as a first step—the reverse of the usual approach to hard problems. Thirty years later, with great fanfare, Karmarkar[5] devised an interior point method that could be shown to have nice theoretical properties, converging in fairly low-order polynomial time. Furthermore, claims were made that this method was orders of magnitude faster than the simplex method in practice. However, members of the LP profession were assured that the method was far too complicated for their understanding.

Undiscouraged, Gill et al.[6] were able to show that the method put forward by Karmarkar was actually equivalent to a form of the logarithmic barrier function method. They also described and gave computational experience with a primal barrier function method. Subsequently, Megiddo[7] developed an elegant theory for interior point methods involving barrier functions for both the primal and dual LP problems. This work led to the development of a number of "primal-dual" interior point

methods, which have gradually assumed dominance in interior point methodology. Outstanding work in this area has been done by Lustig, Marsten, and Shanno,[8] who developed the OB1 primal-dual optimizer, and by Mehrotra,[9] who developed a "predictor-corrector" variant of the primal-dual method, which appears to be even more efficient. This variant was subsequently incorporated into OB1,[10] and the superiority of this technique was confirmed by thorough computational testing. Currently it is the method of choice for most problems.

All three algorithmic approaches—the primal barrier, primal-dual, and predictor-corrector—have been implemented in Release 2 of the IBM Optimization Subroutine Library (OSL).

In the next section of this paper we give a brief mathematical background on barrier approaches to LP to bring out the fundamental compute-intensive steps common to all current barrier methods. It is followed by a discussion of how these steps can be implemented, with emphasis on two architectures—the System/370* Vector Facility and the IBM RISC System/6000*. This discussion includes some illustrative computational experience to show the effect of specially tailoring the code for the architectures.

Mathematical background

We take the linear programming problem to be expressed in the form:

$$\min_x \sum_j c_j x_j \tag{1}$$

subject to:

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

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

where A is an $m \times n$ $(m < n)$ matrix that is usually large and in practice almost always sparse, with perhaps four to ten nonzeros per column. What makes this problem different from those considered in classical optimization, via calculus, is the presence of inequalities [the bounds in (3)] and the linearity of all the functions. The way in which these characteristics are handled determines the features of the two classes of methods that have been successful in practice.

The simplex method makes explicit use of the fact that the linear constraints, together with the inequalities, describe a (convex) polyhedron, with the optimal solution at one of its vertices.[1] It uses a descent method, moving along edges of the polyhedron from one vertex to an adjacent one with a better solution value, until an optimum is obtained (or shown to be nonexistent). It is a *combinatorial* method in the sense that some of the bounds as shown in (3) are satisfied as equalities at each vertex (i.e., tight) and others are satisfied as strict inequalities (i.e., loose). The progress from vertex to vertex can then be viewed as a search for the correct set of loose and tight bounds. Alternatively, we can think of it as a method that follows a path on the *surface* of the polyhedron defining the set of feasible solutions to (2) and (3). Further details of the method, and its implementation in OSL, may be found in a companion paper by the authors.[11]

In contrast to the simplex method, barrier methods deal with the inequalities by introducing nonlinearities that prevent them from being violated and then applying calculus-based methods. In particular, logarithmic barrier methods work by considering a family of related problems:

$$\min_x F(x) = \sum_j (c_j x_j - \mu \ln (x_j - L_j) - \mu \ln (U_j - x_j)) \tag{4}$$

subject to:

$$Ax = b \tag{5}$$

where $\mu \to 0$ and $\mu > 0$.

Since the logarithmic function is not even defined for nonpositive arguments, then clearly the bounds in (3) cannot be violated. Furthermore, as μ is positive, the objective becomes large as variables approach their bounds. However, it can be shown (see, e.g., Fiacco and McCormick[4]) that as the *barrier parameter* μ tends toward zero the solution to (4) and (5) tends toward a solution of (1) through (3). Again in contrast to the simplex method, these solutions follow a path through the *interior* of the polyhedron.

A primal barrier method. One standard approach to solving a linear equality constrained optimization problem as expressed in (4) and (5) is to use a feasible-point descent method (see, e.g., Gill, Murray, and Wright[12]). Given a trial solution x that satisfies $Ax = b$, the next iterate $\bar{x}$ is defined as

$$\bar{x} = x + \alpha \Delta x \tag{6}$$

where Δx is the *search direction*, and α is the *step-length*. The computation of Δx and α must be such that $A\bar{x} = b$ and the value of F decreases.

If we define $g \equiv \nabla F(x)$, $H \equiv \nabla^2 F(x)$, and y is the vector of Lagrange multipliers for the linear constraints, the *Newton search direction* Δx for this problem satisfies the linear system

$$\begin{pmatrix} H & A^\top \\ A & 0 \end{pmatrix} \begin{pmatrix} -\Delta x \\ y \end{pmatrix} = \begin{pmatrix} g \\ 0 \end{pmatrix} \tag{7}$$

To see how this system can be solved let us define:

$s_j = x_j - L_j$, $t_j = U_j - x_j$,

$D = diag\{(1/s_j^2 + 1/t_j^2)^{-1/2}\}$, and

$\hat{D} = diag\{(1/s_j - 1/t_j)\}$

Then the g, H terms in (7) are defined by:

$g(x) = c - \mu \hat{D} e$ and $H(x) = \mu D^{-2}$

and it follows from (7) that Δx and y must satisfy

$$\begin{pmatrix} \mu D^{-2} & A^\top \\ A & 0 \end{pmatrix} \begin{pmatrix} -\Delta x \\ y \end{pmatrix} = \begin{pmatrix} c - \mu \hat{D} e \\ 0 \end{pmatrix}$$

Introducing a new vector r defined by $Dr = -\mu \Delta x$, we see that r and y then satisfy

$$\begin{pmatrix} I & DA^\top \\ AD & 0 \end{pmatrix} \begin{pmatrix} r \\ y \end{pmatrix} = \begin{pmatrix} Dc - \mu D \hat{D} e \\ 0 \end{pmatrix}$$

Equivalently we may say that y is the solution and r the optimal residual of the least-squares problem:

$$\underset{y}{\text{minimize}} \; \|Dc - \mu D \hat{D} e - DA^\top y\|$$

The Newton barrier direction is then obtained as

$$\Delta x = -(1/\mu) Dr$$

As we iterate, the vector y converges to the Lagrange multipliers for the constraints $Ax = b$ in the original problem. However, in practice it is better to work with a correction Δy at each iteration, updating y, d, and the reduced gradient r, where the latter two terms are defined by:

$d = c - A^\top y$, and

$r = Dd - \mu D \hat{D} e$

The details on how the barrier parameter μ is controlled and convergence is established are given in Gill et al.[6] and will not be repeated here. The core computational steps of an iteration are seen to be:

1. Solve a least-squares problem:

$$\min_{\Delta y} \|r - DA^\top \Delta y\| \tag{8}$$

2. Update the dual values:

$y \leftarrow y + \Delta y$, and

$d \leftarrow d - A^\top \Delta y$

3. Compute the search direction Δx from:

$r = Dd - \mu D \hat{D} e$, and

$\Delta x = -(1/\mu) Dr$

4. Calculate the steplength α and update x

The most critical step in each iteration is solving the least-squares problem in (8). Although Gill et al.[6] advocated using an iterative method for this step, in the great majority of cases it may be done directly by solving the normal equations

$$AD^2 A^\top \Delta y = ADr$$

which for reasons to become apparent below will be written

$$A \Theta A^\top \Delta y = v \tag{9}$$

where $v = ADr$, $\Theta = D^2 = (S^{-2} + T^{-2})^{-1}$ and $S = diag\{s_j\}$, $T = diag\{t_j\}$. This system is solved by computing the *Cholesky factors* (triangular factors) $LL^\top$ of $A\Theta A^\top$ and then backsolving to obtain $\Delta y = L^{-\top}L^{-1}v$.

In practice, the system in (9) must be permuted to reduce fill-in of new nonzeros in the relatively sparse $A\Theta A^\top$, so that the Cholesky factors are actually computed from

$$LL^\top = PA\Theta A^\top P^\top$$

where P is the permutation matrix that defines the new row ordering. This implies that some nontrivial procedures must be carried out before the barrier algorithm can be even begun. In particular, the permutation P and the nonzero structure of L must be determined.

To determine P we require the nonzero structure of $AA^\top$. It may be obtained in OSL in one of two ways. The first method considers the Boolean matrices derived from A and $A^\top$ (which have a one in each position where A or $A^\top$ have a nonzero and zeros elsewhere). The Boolean product of A and $A^\top$ then has the desired nonzero structure. The second method uses only the Boolean matrix derived from A and sums the outer products of the columns with themselves. Having obtained the nonzero structure of $AA^\top$, processing must be done by some ordering method. OSL and many other codes use some version of what is known as a "minimum degree" ordering algorithm.[13] Once the permutation P is available, a "symbolic factorization"[13] is carried out to compute where the nonzero elements will be in the Cholesky factor L. When this nonzero structure for L is available, the memory requirements for the algorithm can be computed, and if insufficient space is available, the procedure may be halted.

Appropriate data structures are required so that we may compute

$$PA\Theta A^\top P^\top = \sum_j \theta_j Pa_{.j} a_{.j}^\top P^\top$$

This requires either the ability to access the matrix A row-wise or the computation of the outer products of the columns of A and their addition to the appropriate elements of the data structure that will contain L. The implementation in OSL allows either alternative, with the outer product form much preferred for vector platforms.

By far, the greatest computational effort in most cases now comes in computing the Cholesky factors at each iteration. This effort is the principal topic of the next section. For simplicity we ignore the permutation P from now on.

The primal-dual barrier method. In discussing the primal-dual method we shall follow the description of the method by Lustig, Marsten, and Shanno[8] with the difference that variables are treated as having arbitrary lower bounds (as in OSL), rather than assuming that they have been normalized to zero (as in their OB1 system). It is convenient to rewrite the primal LP problem [(1)–(3)] so that the upper and lower bound constraints are separate:

$$\min_x c^\top x$$

subject to:

$$Ax = b,$$

$$x \geq L,$$

$$-x \geq -U$$

and its dual problem:[1]

$$\max_{y,z,w} b^\top y + L^\top z - U^\top w$$

subject to:

$$A^\top y + z - w = c$$

$$z \geq 0,\ w \geq 0$$

Now, let us write the barrier function form of the primal, to convert it to an equality constrained problem, with the slack variables s and t on the upper and lower bounds (as defined in the previous subsection) explicitly included:

$$\min_x \sum_j (c_j x_j - \mu \ln s_j - \mu \ln t_j)$$

subject to:

$$Ax = b,$$

$$x - s = L,$$

$$-x - t = -U$$

This has as its classical Lagrangian:

$$L(x, s, y, z, w, \mu) = \sum_j (c_j x_j - \mu \ln s_j - \mu \ln t_j) - y^T(Ax - b) - z^T(x - s - L) - w^T(U - x - t) \quad (10)$$

The first order necessary conditions for (10) can be written:

$$\begin{aligned} SZe &= \mu e, \\ TWe &= \mu e, \\ Ax &= b, \\ x - s &= L, \\ x + t &= U, \\ A^T y - w + z &= c \end{aligned} \quad (11)$$

where the multipliers z and w correspond to the slack variables in the dual LP above and $S = diag\{s_i\}$, $T = diag\{t_i\}$, $Z = diag\{z_i\}$ and $W = diag\{w_i\}$. Notice that these conditions comprise the primal and dual equality constraints plus the nonlinear equations $SZe = \mu e$, $TWe = \mu e$. These terms become the LP complementarity conditions[1] as μ approaches zero.

The straightforward primal-dual method proceeds by taking, for a decreasing sequence of values of μ, a step of Newton's method for these nonlinear equations to compute a search direction $(\Delta x, \Delta s, \Delta t, \Delta y, \Delta z, \Delta w)$. After some tedious but elementary algebra, the computational steps are seen to require solving a set of equations:

$$A\Theta A^T \Delta y = A\Theta\tau(\mu) - Ax + b \quad (12)$$

where $\Theta = (S^{-1}Z + T^{-1}W)^{-1}$ and $\tau(\mu) = c - A^T y + \mu(T^{-1} - S^{-1})e$. The value of Δy may now be used to compute:

$$\begin{aligned} \Delta x &= \Theta(A^T \Delta y - \tau(\mu)), \\ \Delta s &= \Delta x, \\ \Delta t &= -\Delta x, \\ \Delta z &= S^{-1}(\mu e - SZe - Z\Delta s), \\ \Delta w &= T^{-1}(\mu e - TWe + W\Delta t) \end{aligned}$$

Two steplengths, α_p and α_D, in the primal and dual spaces, are chosen to preserve feasibility, and then a new approximate minimizing solution is determined as $\hat{x} = x + \alpha_P \Delta x$, $\hat{s} = s + \alpha_P \Delta s$, $\hat{t} = t + \alpha_P \Delta t$, $\hat{y} = y + \alpha_D \Delta y$, $\hat{z} = z + \alpha_D \Delta z$, $\hat{w} = w + \alpha_D \Delta w$. These steps are repeated until the relative gap between the (feasible) primal and dual solutions falls below some user-specified tolerance.

This rather terse outline of the primal-dual method is sufficient to point out its relationship with the barrier primal method. Most importantly, the major computational step is seen to be the solution of a linear system, whose matrix $A\Theta A^T$ has the same structure as the matrix whose Cholesky factors we needed in the primal barrier method. Thus the bulk of the work, and the preprocessing, are very similar. The potential advantage of the primal-dual approach is that feasible dual information is available, and the number of steps may be smaller. Lustig, Marsten, and Shanno[8] present extensive computational experience with this method, and our own experience with OSL is that it takes fewer iterations and is more robust than the primal method on most problems.

The predictor-corrector method. The predictor-corrector method is a variant, due to Mehrotra,[9] of the primal-dual barrier method. Once again we follow an excellent description given by Lustig, Marsten, and Shanno[10] suitably modified to consider general lower as well as upper bounds. The essence of the method is a more ambitious approach to solving for the first order necessary conditions of (11). Instead of routinely applying Newton's method to this nonlinear system, Mehrotra asked whether it might not be possible to derive modifications Δx, Δs, Δt, Δy, Δz, Δw to the current trial solution x, s, t, y, z, w by directly solving for them. Substituting $x + \Delta x$ for x, etc., in (11), one obtains the system:

$$\begin{aligned} A\Delta x &= b - Ax, \\ \Delta x - \Delta s &= L - x + s, \\ \Delta x + \Delta t &= U - x - t, \\ A^T \Delta y + \Delta z - \Delta w &= c - A^T y - z + w, \\ S\Delta z + Z\Delta s &= \mu e - SZe - \Delta S \Delta Z e, \\ T\Delta w + W\Delta t &= \mu e - TWe - \Delta T \Delta W e \end{aligned} \quad (13)$$

where $\Delta S = diag\{\Delta s_i\}$, etc. Unfortunately this system is also nonlinear because of the product terms $\Delta S \Delta Z$ and $\Delta T \Delta W$ in the last two equations. A direct approach is to derive approximations for these product terms, plug them into the right side of (13), and then solve the system.

The *predictor* step in the predictor-corrector method solves the *affine* variant of the model, that is, it omits the μ and Δ-product terms from the right sides of the last two expressions:

$$\begin{aligned} A\Delta\hat{x} &= b - Ax, \\ \Delta\hat{x} - \Delta\hat{s} &= L - x + s, \\ \Delta\hat{x} + \Delta\hat{t} &= U - x - t, \\ A^T\Delta\hat{y} + \Delta\hat{z} - \Delta\hat{w} &= c - A^T y - z + w, \\ S\Delta\hat{z} + Z\Delta\hat{s} &= -SZe, \\ T\Delta\hat{w} + W\Delta\hat{t} &= -TWe \end{aligned} \tag{14}$$

Just as in the ordinary primal-dual method, the essential step is the solution of a system of the form:

$$A\Theta A^T\Delta\hat{y} = \hat{v}$$

where Θ is defined as for (12) and only the right side $\hat{v}$ is slightly different.

With $\Delta\hat{s}$, $\Delta\hat{z}$, $\Delta\hat{t}$, and $\Delta\hat{w}$ available, we are ready to perform the *corrector* step, which solves the linear system:

$$\begin{aligned} A\Delta x &= b - Ax, \\ \Delta x - \Delta s &= L - x + s, \\ \Delta x + \Delta t &= U - x - t, \\ A^T\Delta y + \Delta z - \Delta w &= c - A^T y - z + w, \\ S\Delta z + Z\Delta s &= \mu e - SZe - \Delta\hat{S}\Delta\hat{Z}e, \\ T\Delta w + W\Delta t &= \mu e - TWe - \Delta\hat{T}\Delta\hat{W}e \end{aligned} \tag{15}$$

The essential step is again the solution of a system of the form:

$$A\Theta A^T\Delta y = v$$

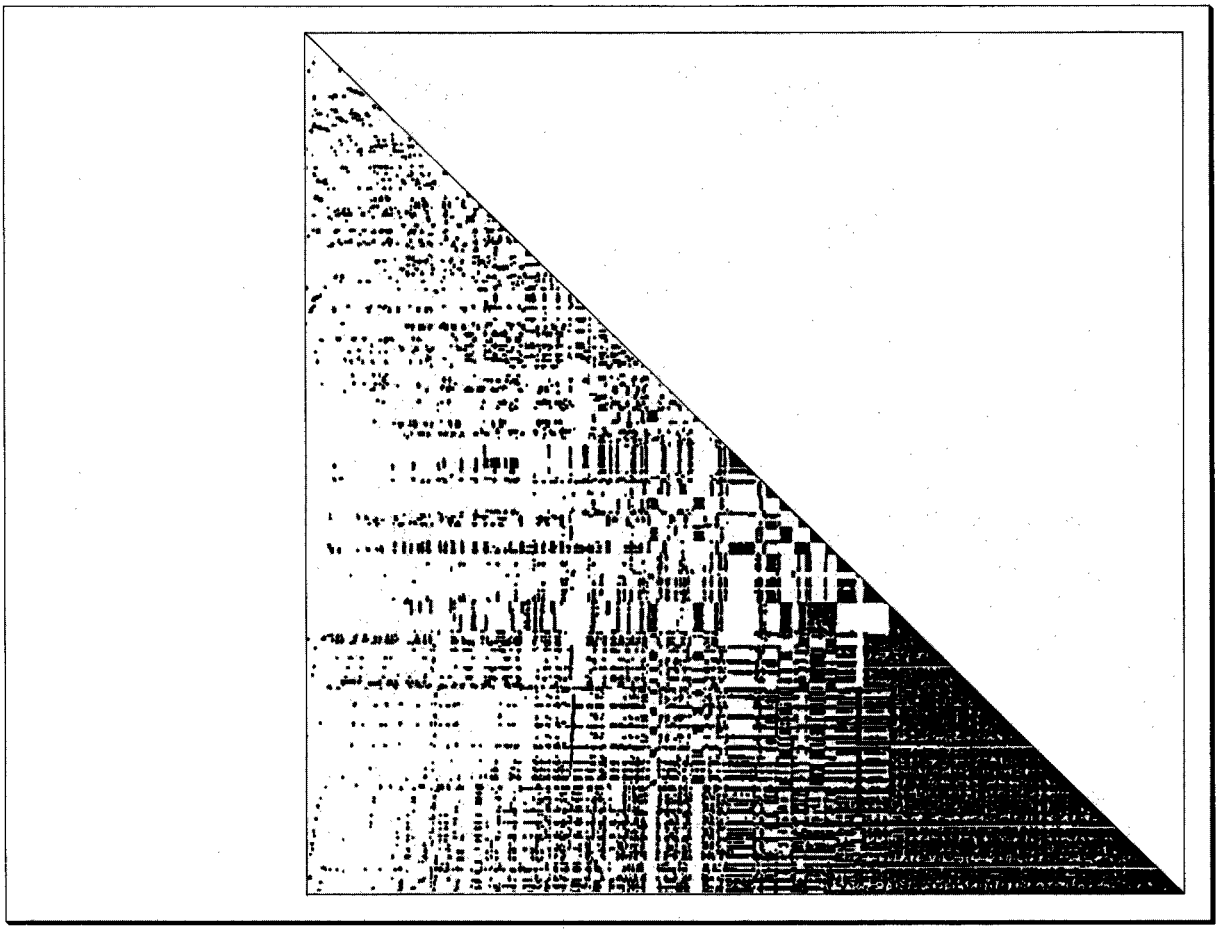
where the right side v is different again from that in the predictor step, but the diagonal Θ is identical. Thus the same factorization of $A\Theta A^T$ is used for both predictor and corrector steps, and only back substitution for Δy, etc., must be done twice. Since the factorization normally dominates the computation, we would expect a net decrease in time if the number of iterations is reduced. Lustig et al.[10] report that this is indeed almost always the case (often by a substantial factor) with their OB1 code, and our experience with OSL Release 2 is similar. In view of these results and the stability of the method, the predictor-corrector method must be considered the current method of choice.

Solving the fundamental system

The previous section has emphasized that the key computational step in all of the interior point methods considered here is the solution of a symmetric system $A\Theta A^T\Delta y = v$. This sort of calculation is quite different from the key computational steps in the simplex method, which mostly involve working with sparse triangular factors obtained by Gaussian elimination from a square submatrix B (the *basis*) of the coefficient matrix A.[14] These factors used in the simplex method can usually be manipulated in such a way that they are not too dense in comparison with B (or A) itself. However, $A\Theta A^T$ is itself usually much denser than any basis B, and its Cholesky factors LL^T are normally much denser again—often by a factor of three or four or more—even after reordering to minimize the fill-in. Despite the fact that interior methods typically only require a few dozen iterations, compared to about $2m$ (where m is the number of rows) for the simplex method, the difference in the central computational step means that the efficiency of the Cholesky factorization is crucial.

Considerable progress has been made in efficient sparse Cholesky factorization[13,15] since barrier methods were first proposed, to the point where interior methods would compete successfully with the simplex method on many classes of LP problems—particularly large ones (see Tomlin[16] for further details). Furthermore, sparse Cholesky factorization is able to make more efficient use of vectorization[17] than the simplex method because the densities are relatively higher, giving longer vector calculations and lower overheads. However, it is the exploitation of dense processing that has

Figure 1 Cholesky factor structure for model PILOTS



made quite significant gains possible in situations where the sheer number of nonzeros to be processed might make interior point methods look unpromising. To see why this might be important, consider Figure 1, where each dot represents a nonzero in the Cholesky factor L (after a minimum degree permutation of the rows) for the model PILOTS (shown later in Tables 1–3).

Clearly, the lower right corner is very close to being completely dense, and there are distinct advantages to treating it as such.[15]

Exploiting the Vector Facility. The form of Cholesky factorization commonly used for sparse processing is of the "pull from behind" variety. That is, each column j of L is computed from the original column $F_{\cdot j}$ of $F = A\Theta A^{\top}$ and the already calculated columns $L_1, \cdots, L_{j-1}$:

$$l_{jj} = \left(f_{jj} - \sum_{k=1}^{j-1} l_{jk}^2 \right)^{1/2} \tag{16}$$

$$l_{ij} = \frac{f_{ij} - \sum_{k=1}^{j-1} l_{jk} l_{ik}}{l_{jj}} \quad (i > j) \tag{17}$$

While (17) obviously requires actual arithmetic only for columns k with nonzero l_{jk} and only for nonzero l_{ik} within such columns, the number of such floating-point operations may still be very

Table 1 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
L Nonzeros			
No dense	30,464	201,464	1,513,330
With Dense	30,805	202,988	1,609,375
Number of			
Dense columns	141	417	1,532
Clique columns	340	425	375

Table 2 Times (in seconds) for an IBM 3090J with Vector Facility

Model	25FV47	PILOTS	DFL001
No dense, no cliques			
Total Cholesky time	4.80	107.74	2,462.76
Dense Cholesky time	0.0	0.0	0.0
Solution time	9.01	131.21	2,535.83
Iterations	26	37	47
With dense, no cliques			
Total Cholesky time	3.35	74.18	1,387.87
Dense Cholesky time	0.40	9.97	462.33
Solution time	7.56	97.72	1,469.52
Iterations	26	37	47
With dense and cliques			
Total Cholesky time	2.73	43.33	819.42
Dense Cholesky time	0.40	10.00	462.87
Solution time	6.95	67.00	901.34
Iterations	26	37	47

Table 3 Times (in seconds) for a RISC System/6000 Model 530

Model	25FV47	PILOTS	DFL001
No dense, no cliques			
Total Cholesky time	9.08	255.79	6,234.21
Dense Cholesky time	0.0	0.0	0.0
Solution time	20.28	326.08	6,410.53
Iterations	26	37	40
With dense, no cliques			
Total Cholesky time	5.56	161.31	3,307.60
Dense Cholesky time	0.71	21.63	1,295.67
Solution time	16.66	230.82	3,495.13
Iterations	26	36	47
With dense and cliques			
Total Cholesky time	4.02	90.28	2,133.43
Dense Cholesky time	0.70	21.62	1,240.45
Solution time	15.14	159.61	2,313.36
Iterations	26	37	45

large, and it only vectorizes moderately well. If it is assumed that the current $L_{\cdot j}$ is being kept and updated in full (unpacked) array form, then for every nonzero l_{jk}, the row indices for the nonzero l_{ik} must be LOADed into a vector register, the corresponding elements of $L_{\cdot j}$ must be gathered (with a LOAD INDIRECT), updated with a MULTIPLY AND SUBTRACT, then restored with a STORE INDIRECT to $L_{\cdot j}$. Ignoring startup times, we see that the Vector Facility requires six cycles per element to perform this innermost loop. If, on the other hand, L can be treated as dense, each $L_{\cdot j}$ (or more correctly, segment of $L_{\cdot j}$) can be LOADed once, and the inner loop consists only of MULTIPLY AND SUBTRACTs (one cycle per element), with a final STORE of the updated column $L_{\cdot j}$. Again if startup times are ignored, this process is nearly six times faster per element (though, of course, there may be more elements).

The approach used in OSL and elsewhere is to define a sparse/dense "cutoff" and to partition F:

$$A\Theta A^{\mathsf{T}} = \begin{pmatrix} F_{11} & F_{21}^{\mathsf{T}} \\ F_{21} & F_{22} \end{pmatrix}$$

so that F_{22} corresponds to the submatrix that will be treated as dense. The standard cutoff in OSL is to partition at the first column encountered with a nonzero density of 0.7, though this may be varied by the user. The technique then is to use standard sparse matrix processing to comute L_{11} and L_{21}, where $L_{11}L_{11}^{\mathsf{T}} = F_{11}$ and $L_{21} = F_{21}L_{11}^{-\mathsf{T}}$, then pass through the columns of L_{21} to produce the matrix:

$$\bar{F}_{22} = F_{22} - L_{21}L_{21}^{\mathsf{T}} \tag{18}$$

which is treated as dense while producing the remainder of the Cholesky factor:

$$L_{22}L_{22}^{\mathsf{T}} = \bar{F}_{22}$$

Dense Cholesky factorization is a well-studied subject, and some efficient routines exist, notably in the IBM Engineering and Scientific Subroutine Library (ESSL).[18] Unfortunately, we were unable

to directly use the ESSL routine, partly because of a conflict in data structures, but most importantly because the systems encountered in LP interior point methods become not only ill-conditioned but rank-deficient, despite strenuous attempts (in OSL "presolve") to remove redundant constraints from the model by inspection. This situation must be dealt with gracefully by identifying and omitting the offending rows from the algorithm as they are encountered. This process is not one that standard library routines are equipped to handle, thus we produced custom vector assembler code.

Agarwal and Gustavson[19] discussed several efficient schemes for factorization using "blocks" or submatrices of the matrix to be factored. Generally they favor subdividing the matrix into dense subblocks which will fit into the high-speed cache memory ("blocking for cache"), an approach we encounter below in discussing the RISC System/6000. This approach is designed to minimize the number of "cache misses" (references to data not currently in the cache), which degrade the computation speed. We chose to "block for registers,"[17] that is, to subdivide the matrix into blocks that will fit into the 16 vector registers (or eight vector double registers). This is a "pull from behind" method that proceeds as follows:

1. For each group of eight consecutive columns, load a block of elements with row size equal to at most the section size (number of elements per vector register). They begin with the immediate subdiagonal element of the first column in position 0 of the first pair of registers, the immediate subdiagonal of the second in position 1 of the second pair of registers, and so on. Thus, successive pairs of registers after the first have one more dummy element at the top than the previous pair, leaving the elements properly aligned (necessary only for the immediate subdiagonal block).
2. Update the block by the same subset of the rows from all previous (updated) columns. These vector operations multiply the same portion of a column resident in memory by (nonzero) scalars and add it to the vector registers. Since we potentially use the memory-resident data eight times, we expect delays caused by cache misses to be slight.
3. Perform the pivots, update these eight column segments entirely within the registers (storing them only when fully updated), and update the appropriate diagonals. The vector mask register may be used here to avoid operating on the dummy elements.
4. Load the next section of (or remaining) rows of the group and update them by all preceding columns, as in step 2.
5. Complete the updates, using only register-to-register arithmetic and the new diagonal values for the columns computed in step 3. Store the updated column segments and modified diagonals for this set of rows.
6. Repeat steps 4 and 5 until all rows have been processed for this group of eight columns.

Some indication of the advantages of this approach over a naive dense Cholesky implementation is given in Forrest and Tomlin.[17] Other computational results are given in the next section.

Clique (supernode) processing. Given the advantages of dense processing, when applicable, it is reasonable to search for other opportunities for sustained vector (or at any rate floating-point) computation. This search has been done for some time in other applications that require factorization of sparse symmetric matrices.[15] A key observation is that after ordering a matrix using some minimum degree algorithm,[13] one encounters groups of columns with identical nonzero structure (except immediately under the diagonal), as shown schematically in Figure 2. These groups of columns are known as *cliques* or *supernodes*—terms derived from the graph theoretic processes of the minimum degree algorithm.

Use of the cliques can significantly improve the speed of at least part of the Cholesky inner loop in (17). When a clique is encountered as $L_{\cdot j}$ is updated, the amount of gathering and scattering of data can be reduced. As before, when the first column of the clique is encountered, the row indices for the nonzero l_{ik} must be LOADed into a vector register and the corresponding elements of $L_{\cdot j}$ must be gathered (with a LOAD INDIRECT). However, after this initialization, for each column in the clique it is only necessary to update with a MULTIPLY AND SUBTRACT. After the clique has been processed, the appropriate elements of $L_{\cdot j}$ are restored with a single STORE INDIRECT to $L_{\cdot j}$. This method resembles a miniature dense update with some pre- and post-processing overhead.

The clique phenomenon can be exploited even more usefully in the process of updating the dense segment discussed above. Let us rewrite (18) in outer product form as:

$$\bar{F}_{22} = F_{22} - \sum_k L_{.k}^{(21)} (L_{.k}^{(21)})^T$$

where $L_{.k}^{(21)}$ is the k^{th} column of L_{21}. One way of implementing this update would be to extract the dense submatrix of $\bar{F}_{22}$, as computed so far, corresponding to the nonzero elements of $L_{.k}^{(21)}$ and then perform a dense outer product update. For a single column k, this method is no more efficient than vectorizing the sparse outer product update in a straightforward way with gather-scatter. However, when there are several columns with identical structure, it can be beneficial to think about blocking the extracted submatrix in the registers and then applying all of the outer product updates for the columns in the clique without further gather-scatter (except for the subsequent storage back into $L_{.j}$). This resembles the blocked dense processing described above except that:

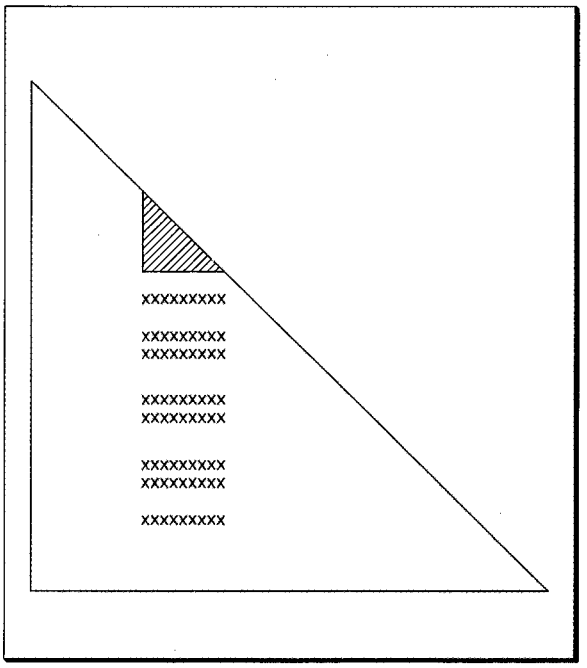
a. The columns of $\bar{F}_{22}$ from which (some) nonzero elements will be loaded are not necessarily contiguous.
b. Only seven dense columns can be included in a block since we need to maintain a vector register of the nonzero row indices to which the elements correspond.

Experience would seem to indicate that much of the advantage of clique processing derives from this updating of the dense segment. Some computational experience with clique processing is presented in the next section.

Exploiting the RISC System/6000 architecture. Most of the detailed techniques we have described for a vector processor are simply inappropriate for the RISC System/6000 architecture. The broad outline of the sparse/dense processing approach does, however, remain appropriate, with differences in algorithmic detail and software engineering.

With very careful coding, the RISC System/6000 can sustain computation of floating-point multiply-adds at one per cycle in dense matrix computations,[20] thus becoming remarkably competitive with even a vector processor. To achieve this rate, at least two conditions are necessary—avoidance of cache misses and coding to achieve instruction overlap. The latter is usually achieved

Figure 2 Clique (supernode) structure



by means of loop unrolling—a technique that places several consecutive calculations in an inner loop, with a corresponding increase in loop increment, so that the floating-point unit may be working on one set of operands while others are being fetched or pointers incremented.

The penalty for a cache miss on the RISC machine is much more severe than on the mainframe computer. Thus, since there are no vector registers to provide a "blocking for registers" strategy, it makes sense to adopt a strict "blocking for cache" policy for matrix computations in this environment (see the highly useful document in Reference 20). This blocking strategy implies a different approach to dense Cholesky factorization—the "push from behind" approach. With this scheme, each column is already fully updated by previous columns when it is encountered. It must then be pivoted on:

$$l_{jj} \leftarrow \sqrt{l_{jj}},$$

$$l_{ij} \leftarrow \frac{l_{ij}}{l_{jj}}, \quad (i > j)$$

and later columns $k = j + 1, j + 2, \cdots$ updated:

$$l_{kk} \leftarrow l_{kk} - l_{kj}^2,$$
$$l_{ik} \leftarrow l_{ik} - l_{ij} l_{kj} \quad (i > k) \tag{19}$$

Although we have written the formulae for individual elements, the same algorithm applies when we consider subblocks of the dense segment $\bar{F}_{22}$. We were fortunate to be able to modify the FORTRAN dense Cholesky code from ESSL/6000* (the RISC System/6000 version of ESSL) which implements this push-ahead block strategy very efficiently by using subblocks that will fit in cache and loop unrolling. The modifications involved a change in data structure for compatibility with OSL and a capability to gracefully survive loss of rank (see above). Computational results indicating the efficacy of this approach to the dense segment are given in the following section.

The use of clique (supernode) processing is also very important on the RISC System/6000, for any device which enables us to avoid indexing and storing is particularly beneficial. Thus, updating a sparse vector $L_{\cdot j}$ by a clique is done by essentially creating a miniature "vector register" from the floating-point registers of the RISC machine, loading it with elements of $L_{\cdot j}$ corresponding to the nonzero row indices in the clique, then performing an unrolled loop over the columns of the clique.

One of the most computationally beneficial steps for the RISC System/6000 is the updating of the dense segment by the cliques. This step follows the basic outline explained above for vector processing, except that the extraction of a subblock of the dense segment, corresponding to the rows of a clique, is not carried out explicitly, but implicitly, using loop unrolling. Great care is also taken to avoid cache misses. The computational effect of the clique processing is shown in the next section.

Some computational results

To illustrate the importance of the sparse/dense/clique processing in Cholesky factorization for both the mainframe Vector Facility and the RISC architecture we present comparative results with three quite well-known LP test models. These models are widely understood to be nontrivial, bellwether models. The characteristics of the models are displayed in Table 1. The "reduced" model that we actually work with is the result of the presolve routine EKKPRSL of OSL. We have run the problems with and without a dense segment (with the default cutoff density at 0.7) and with a dense segment and cliques (the current standard). As a result, we have two figures for the number of nonzeros in the Cholesky factor—that obtained with no dense segment defined and one with.

Table 2 gives the results for running each of the three cases on an IBM 3090J system for each of the three problems. For each case we give the total time spent in the Cholesky routine, the amount of that time that was spent in the dense Cholesky factorization, the total time to reach a solution (including input and all preprocessing), and the number of iterations taken by the method, which in all cases is the predictor-corrector variant of the primal-dual method. It can be seen that the improvement in total Cholesky time with increasing use of dense processing itself improves with increasing model size from a factor of less than two for 25FV47 to a factor of three for DFL001. Furthermore, we see the importance of this improvement in reducing total solution time with increasing problem size—at least for this sample of models. The occasional small discrepancy in timings for the same operations is due to the coarseness of the timer.

Table 3 gives similar figures for running the same set of cases on a RISC System/6000 Model 530. We see here that the relative improvement from simplest to most sophisticated processing is just a little less than for the mainframe with the Vector Facility. This should not be a surprise. What is impressive is that the solution times for the RISC machine are all only a factor of less than three more than the mainframe times. The picture is clouded slightly by the variability in the number of iterations. It is the result of small changes in accuracy when the computations are performed in different order for the variations of the Cholesky factorization. It is perhaps more apparent on the RISC machine because of the greater accuracy of its arithmetic. However, even with these variations the pattern of the results is clear.

Conclusions

The computational results show that interior point methods can be implemented in a general-

purpose mathematical programming system such as OSL and solve very substantial problems rapidly on both mainframe computers and workstations. Furthermore, both types of computers are able to benefit from advances in both sparse and dense matrix processing techniques. Interior point LP methods are not yet (and may never be) the answer to all demands for rapid solution. In particular, there is at present no counterpart to the ability of the simplex method to efficiently exploit solutions to related problems (that is, a "warm start"). Applied interior point LP optimization is still in its infancy compared with the simplex method and its variants. However, it is rapidly becoming a standard tool for solving large, difficult problems from scratch.

Acknowledgments

We are indebted to Jenny Edwards for converting the ESSL/6000 dense Cholesky routines to function under OSL, to Fred Gustavson for helpful discussions, and to Irvin Lustig for generating, and giving us permission to use, Figure 1.

*Trademark or registered trademark of International Business Machines Corporation.

Cited references

1. G. B. Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ (1963).
2. 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.
3. K. R. Frisch, *The Logarithmic Potential Method of Convex Programming*, memorandum of May 13, 1955, University Institute of Economics, Oslo, Norway.
4. A. V. Fiacco and G. P. McCormick, *Nonlinear Programming: Sequential Unconstrained Minimization Techniques*, John Wiley & Sons, Inc., New York and Toronto (1968).
5. N. Karmarkar, "A New Polynomial-Time Algorithm for Linear Programming," *Combinatorica* **4**, 373–395 (1984).
6. P. E. Gill, W. Murray, M. A. Saunders, J. A. Tomlin, and M. H. Wright, "On Projected Newton Barrier Methods for Linear Programming and an Equivalence to Karmarkar's Projective Method," *Mathematical Programming* **36**, 183–209 (1986).
7. N. Megiddo, "Pathways to the Optimal Set in Linear Programming," in *Progress in Mathematical Programming*, N. Megiddo, Editor, Springer-Verlag, New York (1989), pp. 131–158.
8. I. J. Lustig, R. E. Marsten, and D. F. Shanno, *Computational Experience with a Primal-Dual Interior Point Method for Linear Programming*, Technical Report SOR 89-17, Princeton University, Princeton, NJ (1989). To appear in *Linear Algebra and Its Applications*.
9. S. Mehrotra, *On the Implementation of a (Primal-Dual) Interior Point Method*, Technical Report 90-03, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL (1990).
10. I. J. Lustig, R. E. Marsten, and D. F. Shanno, *On Implementing Mehrotra's Predictor-Corrector Interior Point Method for Linear Programming*, Technical Report SOR 90-03, Department of Civil Engineering and Operations Research, Princeton University, Princeton, NJ (1990).
11. J. J. H. Forrest and J. A. Tomlin, "Implementing the Simplex Method for the Optimization Subroutine Library," *IBM Systems Journal* **31**, No. 1, 11–25 (1992, this issue). Also, Research Report RJ 8174, IBM Almaden Research Center, San Jose, CA (1991).
12. P. E. Gill, W. Murray, and M. H. Wright, *Practical Optimization*, Academic Press, London and New York (1981).
13. J. A. George and J. W. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1981).
14. E. M. L. Beale, "The Evolution of Mathematical Programming Systems," *Journal of the Operational Research Society* **36**, 357–366 (1985).
15. I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, Oxford (1986).
16. J. A. Tomlin, "A Note on Comparing Simplex and Interior Methods for Linear Programming," in *Progress in Mathematical Programming*, N. Megiddo, Editor, Springer-Verlag, New York (1989), pp. 91–103.
17. 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).
18. *Engineering and Scientific Subroutine Library Guide and Reference*, SC23-0184-4, IBM Corporation (1990); available through IBM branch offices.
19. R. C. Agarwal and F. G. Gustavson, *Vector and Parallel Algorithms for Cholesky Factorization on IBM 3090*, Research Report RC 14901, IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598 (1989).
20. *IBM RISC System/6000 Performance Tuning for Numerically Intensive FORTRAN and C Programs*, GG24-3611, IBM Corporation (1990); available through IBM branch offices.

General references

IBM System/370 Vector Operations, SA22-7125-3, 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.

Accepted for publication September 10, 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 sev-

eral 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 postdoctoral 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. G321-5458.