SIAM J. SCI. STAT. COMPUT.
Vol. 5, No. 3, September 1984

005

SPARSE MATRIX METHODS IN OPTIMIZATION*

PHILIP E. GILL†, WALTER MURRAY†, MICHAEL A. SAUNDERS†
AND MARGARET H. WRIGHT†

Abstract. Optimization algorithms typically require the solution of many systems of linear equations $B_k y_k = b_k$. When large numbers of variables or constraints are present, these linear systems could account for much of the total computation time.

Both direct and iterative equation solvers are needed in practice. Unfortunately, most of the off-the-shelf solvers are designed for single systems, whereas optimization problems give rise to hundreds or thousands of systems. To avoid refactorization, or to speed the convergence of an iterative method, it is essential to note that B_k is related to B_{k-1}.

We review various sparse matrices that arise in optimization, and discuss compromises that are currently being made in dealing with them. Since significant advances continue to be made with single-system solvers, we give special attention to methods that allow such solvers to be used repeatedly on a sequence of modified systems (e.g., the product-form update; use of the Schur complement). The speed of factorizing a matrix then becomes relatively less important than the efficiency of subsequent solves with very many right-hand sides.

At the same time, we hope that future improvements to linear-equation software will be oriented more specifically to the case of related matrices B_k.

Key words. large-scale nonlinear optimization, sparse matrices, sparse linear and nonlinear constraints, linear and quadratic programming, updating matrix factorizations

1. Introduction.

1.1. Background. The major application of sparse matrix techniques in optimization up to the present has been in the implementation of the simplex method for *linear programming* (LP) (see, e.g., Dantzig (1963)). In fact, commercial codes for large LP problems seem to have predated codes for sparse linear equations (even though solving a sparse LP problem requires solving many sparse linear systems). In the commercial world today, more sparse matrix computation is probably expended on linear programs than on any other type of problem, and linear programs involving thousands of unknowns can be solved routinely. Because of the great success of the simplex algorithm and the wide availability of LP codes, many large-scale optimization problems tend to be formulated as purely linear programs. However, we shall see that this limitation is often unnecessary.

Before considering particular methods, we emphasize that methods for large-scale optimization have a special character attributable in large part to the critical importance of linear algebraic procedures. Since dense linear algebraic techniques tend to become unreasonably expensive as the problem dimension increases, it is usually necessary to compromise what seems to be an "ideal" strategy. (In fact, an approach that would not even be considered for small problems may turn out to be the best choice for some large problems.) Furthermore, the relative cost of the steps of many optimization methods changes when the problem becomes large. For example, the performance of

* Received by the editors December 22, 1982, and in final form August 22, 1983. This research was supported by the U.S. Department of Energy under contract DE-AM03-76SF000326, PA no. DE-AT03-76ER72018, the National Science Foundation under grants MCS-7926009 and ECS-8012974; the Office of Naval Research under contract N00014-75-C-0267; and the U.S. Army Research Office under contract DAAG29-81-K-0156.

† Systems Optimization Laboratory, Department of Operations Research, Stanford University, Stanford, California 94305.

unconstrained optimization algorithms is often measured by the number of evaluations of the objective function required for convergence. Although simplistic, this is a reasonable gauge of effectiveness for most problems of low dimension because the number of arithmetic operations per iteration tends to be small, and the amount of work required for storage manipulation is negligible. However, as the size of the problem grows, the "housekeeping" (cost of arithmetic and data structures) becomes comparable to, and may even dominate, the cost of function evaluations.

Most optimization methods are iterative; we shall consider algorithms in which the $(k+1)$th iterate is defined as

$$x_{k+1} = x_k + \alpha_k p_k, \tag{1.1}$$

where α_k is a nonnegative scalar, and the n-vector p_k is called the search direction. One of the primary applications of sparse matrix techniques in optimization is in solving one or more systems of linear equations to obtain p_k.

It is usual for thousands of iterations to be required to solve a single large optimization problem, and hence it might appear that the computation time required would be enormous, even with the best available sparse matrix techniques. Fortunately, the linear systems that define p_{k+1} are usually closely related to those that define p_k (and the degree of closeness can be controlled to some extent by the choice of algorithm). In addition, the sequence $\{x_k\}$ will often converge to the solution with only mild conditions on $\{p_k\}$. Consequently, there is a certain flexibility in the definition of p_k. The design of algorithms for large-scale optimization problems involves striking a balance between the effort expended at each iteration to compute p_k and the number of iterations required for convergence.

1.2. Summary. The three main subdivisions of optimization are discussed in turn (unconstrained, linearly constrained, and nonlinearly constrained). A common denominator is the need to solve many systems of linear equations, and the need to *update* various factorizations in order to deal with sequences of related equations. We indicate situations where off-the-shelf software can be applied. Symmetric positive-definite solvers are mainly useful for unconstrained problems, while unsymmetric solvers are essential for dealing with linear constraints. There is an inevitable emphasis on the latter because most large optimization problems currently being solved involve sparse linear constraints.

The principal updating problem is that of replacing one column of a square matrix. However, there exists only one generally available package for updating sparse factors *in situ*. We therefore focus on methods that allow an off-the-shelf solver to be used repeatedly on the same matrix with different right-hand sides. Such methods facilitate more general updates to sparse matrices. In one instance, a sparse indefinite solver is needed.

The final section on nonlinear constraints covers methods that solve a sequence of simpler subproblems, to which the preceding comments apply.

2. Unconstrained optimization.

2.1. Methods for dense problems. The unconstrained optimization problem involves the minimization of a scalar-valued objective function, i.e.

$$\underset{x \in \mathfrak{R}^n}{\text{minimize}}\ F(x).$$

We assume that F is smooth; let $g(x)$ and $H(x)$ denote the gradient vector and Hessian matrix of F.

Many techniques are available for solving unconstrained problems in which n is small (for recent surveys, see, e.g., Brodlie (1977), Fletcher (1980), Gill, Murray and Wright (1981)). The most popular methods compute the search direction as the solution of a system of linear equations of the form

$$H_k p_k = -g_k, \tag{2.1}$$

where g_k is the gradient of F at x_k, and H_k is a suitable symmetric matrix that is most often intended to represent (in some sense) $H(x_k)$. If H_k is positive definite, the solution of (2.1) is the step to the minimum of the local quadratic approximation to F at x_k:

$$\underset{p \in \Re^n}{\text{minimize}}\ g_k^T p + \tfrac{1}{2} p^T H_k p. \tag{2.2}$$

The major distinctions among algorithms involve the definition of H_k.

When H_k is the exact Hessian at x_k or a finite-difference approximation, the algorithm based on solving (2.1) for p_k is called a *Newton-type* method. Newton-type methods tend to be powerful and robust when properly implemented, and exhibit quadratic convergence under mild conditions. However, certain difficulties arise when H_k is indefinite, since the quadratic function (2.2) is unbounded below and the solution of (2.1) may be undefined. Numerous strategies have been suggested for this case, and often involve defining p_k as the solution of a linear system with a positive-definite matrix that is closely related to the Hessian. These techniques include the modified Cholesky factorization of Gill and Murray (1974) and various trust-region strategies (see, e.g., Moré and Sorensen (1982)).

When an exact or finite-difference Hessian is unavailable or too expensive, a popular alternative is to use a *quasi-Newton method* (see Dennis and Moré (1977) for a survey). In a quasi-Newton method, the matrix H_k is an approximation to the Hessian that is *updated* by a low-rank change at each iteration, based on information about the change in the gradient. The hope is that the approximation will improve as the iterations proceed. Quasi-Newton methods typically display a superlinear rate of convergence in practice, and are often more efficient (in terms of computation time) than Newton-type methods.

When n becomes very large, two related difficulties can occur with methods that solve (2.1) directly: excessive computation time and insufficient storage for the $n \times n$ matrix H_k. Fortunately, the Hessian matrices of many large unconstrained problems are quite sparse, and density tends to decrease as n increases. Large problems can thus be solved efficiently using techniques that exploit sparsity in H_k to save work and/or storage, or that do not require storage of H_k.

2.2. Newton-type methods. When the Hessian is sparse and can be computed analytically, a Newton-type method can be implemented by applying standard sparse procedures to solve $H_k p_k = -g_k$. In particular, when H_k is positive definite, any efficient technique for computing a sparse Cholesky factorization may be applied in this context (for a survey of available software, see Duff (1982)). Although many linear systems may need to be solved before the method converges, all of them have the same sparsity pattern, and hence the structure needs to be analyzed only once.

Indefiniteness in a sparse Hessian may be treated using the procedures mentioned for the dense case. The modified Cholesky factorization (Gill and Murray (1974)) has been adapted in a straightforward fashion to treat sparsity (see Thapa (1980)). One advantage of the modified Cholesky approach is that indefiniteness can be detected

and corrected while constructing the factorization of the positive-definite matrix to be used in computing p_k; hence, only one sparse factorization needs to be computed at each iteration. With trust-region methods, p_k may be obtained using off-the-shelf software for a sparse Cholesky factorization; however, these methods typically require more than one factorization per iteration.

When the gradient is available, but the exact Hessian is not, a finite-difference approximation to the Hessian may be used as H_k. In the general case, this requires n gradient evaluations. However, if the sparsity pattern of the Hessian is known a priori it is possible to choose special vectors that allow a finite-difference approximation to $H(x)$ to be computed with many fewer than n evaluations of the gradient.

For example, suppose that $H(x)$ is tridiagonal:

$$H(x) = \begin{pmatrix} \times & \times & & & & & \\ \times & \times & \times & & & & \\ & \times & \times & \times & & & \\ & & \times & \times & \times & & \\ & & & \cdot & \cdot & \cdot & \\ & & & & \times & \times & \times \\ & & & & & \times & \times \\ & & & & & & \times \end{pmatrix}.$$

Consider the vectors

$$y_i = \frac{1}{h\|z_i\|_2}(g(x_k + hz_i) - g(x_k)), \qquad i = 1, 2,$$

where $z_1 = (1, 0, 1, 0, \cdots)^T$, $z_2 = (0, 1, 0, 1, \cdots)^T$, and h is an appropriate finite-difference interval. Let $y_{1,i}$ denote the ith component of y_1, and similarly for y_2. The vectors y_1 and y_2 are approximations to the sums of odd and even columns of H_k, respectively. Therefore,

$$y_{1,1} \approx \frac{\partial^2 F}{\partial^2 x_1}, \quad y_{2,1} \approx \frac{\partial^2 F}{\partial x_1 \partial x_2}, \quad y_{1,2} \approx \frac{\partial^2 F}{\partial x_1 \partial x_2} + \frac{\partial^2 F}{\partial x_2 \partial x_3}, \quad \text{and so on.}$$

Thus, for example,

$$y_{1,2} - y_{2,1} \approx \frac{\partial^2 F}{\partial x_2 \partial x_3}.$$

In this fashion, all the elements of H_k can be approximated with only two evaluations of the gradient, *regardless of the value of n.*

The idea of analyzing the sparsity pattern of the Hessian in order to determine suitable finite-difference vectors has been the subject of much recent interest. An algorithm for finding finite-difference vectors for a general sparse (*unsymmetric*) matrix is given by Curtis, Powell and Reid (1974), and is based on grouping together columns in which there are no overlapping elements. In the unsymmetric case, the problem of finding a minimum set of vectors can be viewed as a graph coloring problem in the directed graph that represents the sparsity pattern. A proof that finding the minimum set is NP-hard is given in Coleman and Moré (1983), along with practical algorithms (see also Coleman and Moré (1982a)).

A similar relationship with graph coloring can be developed for the case of a symmetric matrix. For example, the requirement of symmetry for a sparse matrix

means that the associated column-interaction graph will be *undirected.* The problem of finding a minimum set of finite-difference vectors for a symmetric matrix is NP-complete (a proof for a particular symmetric problem is given in McCormick (1983); see also Coleman and Moré (1982b)). Nonetheless, effective algorithms have been developed based on graph-theoretic heuristics. The algorithms are based on principles similar to those for the unsymmetric case, but are considerably complicated by exploiting symmetry.

A finite-difference Newton-type method for sparse problems thus begins with a procedure that analyzes the sparsity pattern in order to determine suitable finite-difference vectors. Algorithms for finding these vectors have been given by Powell and Toint (1979) and Coleman and Moré (1982b). Once a sparse finite-difference Hessian approximation has been computed, a sparse factorization can be computed as with the exact Hessian.

2.3. Sparse quasi-Newton methods. Because of the great success of quasi-Newton methods on dense problems, it is natural to consider how such methods might be extended to take advantage of sparsity in the Hessian. This extension was suggested first for the case of sparse nonlinear equations by Schubert (1970), and was analyzed by Marwil (1978). Discussions of sparse quasi-Newton methods for optimization and nonlinear equations are given in Toint (1977), Dennis and Schnabel (1979), Toint (1979), Shanno (1980), Steihaug (1980), Thapa (1980), Powell (1981), Dennis and Marwil (1982) and Sorensen (1982). In the remainder of this section we give a brief description of sparse quasi-Newton methods applied to unconstrained optimization.

In quasi-Newton methods for dense problems, the Hessian approximation H_k is *updated* at each iteration by the relationship

$$H_{k+1} = H_k + U_k.$$

The update matrices U_k associated with many dense quasi-Newton methods are of rank two, and can be shown to be the minimum-norm symmetric change in H_k, subject to satisfying the quasi-Newton condition

$$H_{k+1} s_k = y_k, \tag{2.3}$$

where $s_k = x_{k+1} - x_k$ and $y_k = g_{k+1} - g_k$ (see, e.g., Dennis and Moré (1977)). By suitable choice of the steplength α_k in (1.1), the property of hereditary positive-definiteness can also be maintained (i.e., H_{k+1} is positive definite if H_k is). However, the update matrices U_k do not retain the sparsity pattern of the Hessian.

The initial approach to developing sparse quasi-Newton updates was to impose the additional constraint of retaining sparsity on the norm-minimization problem (Powell (1976); Toint (1977)). Let $\mathcal{N}$ be defined as the set of indices $\{(i, j) | H_{ij}(x) = 0\}$, so that $\mathcal{N}$ represents the specified sparsity pattern of the Hessian, and assume that H_k has the same sparsity pattern. A sparse update matrix U_k is then the solution of

$$\begin{aligned} &\underset{U}{\text{minimize}} \ \|U\| \\ &\text{subject to } (H_k + U) s_k = y_k, \\ &\qquad U = U^T, \\ &\qquad U_{ij} = 0 \quad \text{for } (i, j) \in \mathcal{N}. \end{aligned} \tag{2.4}$$

Let $\sigma^{(j)}$ denote the vector s_k with the sparsity pattern of the jth column of H_k imposed. When the norm in (2.4) is the Frobenius norm, the solution is given by

$$U_k = \sum_{j=1}^{n} \lambda_j (e_j \sigma^{(j)T} + \sigma^{(j)} e_j^T), \tag{2.5}$$

where e_j is the jth unit vector and λ is the vector of Lagrange multipliers associated with the subproblem (2.4). The vector λ is the solution of the linear system

$$Q\lambda = y_k - H_k s_k, \tag{2.6}$$

where

$$Q = \sum_{j=1}^{n} (\sigma_j^{(j)} \sigma^{(j)} + \|\sigma^{(j)}\|_2^2 e_j) e_j^T.$$

The matrix Q is symmetric and has the same sparsity pattern as H_k; Q is positive-definite if and only if $\|\sigma^{(j)}\| > 0$ for all j. (The sparse analogue of *any* quasi-Newton formula may be obtained using a similar analysis; see Shanno (1980) and Thapa (1980).)

Thus far, sparse quasi-Newton methods have not enjoyed the great success of their dense counterparts. First, there are certain complications that result from the requirement of sparsity. In particular, note that the update matrix U_k (2.5) is of rank n, rather than of rank two; this means that the new approximate Hessian cannot be obtained by a simple update of the previous approximation. Second, an additional sparse linear system (2.6) must be solved in order to compute the update. Finally, it is not possible in general to achieve the property of hereditary positive-definiteness in the matrices $\{H_k\}$ if the quasi-Newton condition is satisfied (see Toint (1979) and Sorensen (1982)); in fact, positive-definiteness may not be retained even if H_k is taken as the exact (positive definite) Hessian and the initial x_k is very close to the solution (see Thapa (1980)).

In addition to these theoretical difficulties, computational results have tended to indicate that currently available sparse quasi-Newton methods are less effective than alternative methods (in terms of the number of function evaluations required for convergence). However, hope remains that their efficiency may be improved—for example, by relaxing the quasi-Newton condition (2.3), or by finding only an approximate solution of (2.6) (Steihaug (1982)). For a discussion of some possible new approaches, see Sorensen (1982).

2.4. Conjugate-gradient methods. The term *conjugate-gradient* refers to a class of optimization algorithms that generate directions of search without storing a matrix. They are essential in circumstances when methods based on matrix factorization are not viable because the relevant matrix is too large or too dense. We emphasize that there are *two* types of conjugate-gradient methods—linear and nonlinear.

The *linear* conjugate-gradient method was originally derived as an iterative procedure for solving positive-definite symmetric systems of linear equations (Hestenes and Stiefel (1952)). It has been studied and analyzed by many authors (see, e.g., Reid (1971)). When applied to the positive-definite symmetric linear system

$$Hx = -c, \tag{2.7}$$

it computes a sequence of iterates using the relation (1.1). The vector p_k is defined by

$$p_k = -(Hx_k + c) + \beta_{k-1} p_{k-1}, \tag{2.8}$$

and the step length α_k is given by an explicit formula. The matrix H need not be stored explicitly, since it appears only in matrix-vector products.

With exact arithmetic, the linear conjugate-gradient algorithm will compute the solution of (2.7) in at most m ($m \leqq n$) iterations, where m is the number of distinct eigenvalues of H. Therefore, the number of iterations required should be significantly reduced if the original system can be replaced by an equivalent system in which the matrix has clustered eigenvalues. The idea of *preconditioning* is to construct a transformation to have this effect on H. One of the earliest references to preconditioning for linear equations is Axelsson (1974). See Concus, Golub and O'Leary (1976) for details of various preconditioning methods derived from a slightly different viewpoint.

The *nonlinear* conjugate-gradient method is used to minimize a nonlinear function without storage of any matrices, and was first proposed by Fletcher and Reeves (1964). In the Fletcher–Reeves algorithm, p_k is defined as in the linear case by (2.8), where the term $Hx_k + c$ is replaced by g_k, the gradient at x_k. For a nonlinear function, α_k in (1.1) must be computed by an iterative step-length procedure. When the initial vector p_0 is taken as the negative gradient and α_k is the step to the minimum of F along p_k, it can be shown that each p_k is a direction of descent for F.

Many variations and generalizations of the nonlinear conjugate-gradient method have been proposed. The most notable features of these methods are: β_k is computed using different definitions; p_k is defined as a linear combination of *several* previous search directions; p_0 is not always chosen as the negative gradient; and α_k is computed with a relaxed linear search (i.e., α_k is not necessarily a close approximation to the step to the minimum of F along p_k). Furthermore, the idea of preconditioning may be extended to nonlinear problems by allowing a preconditioning matrix that varies from iteration to iteration.

It is well known that rounding errors may cause even the linear conjugate-gradient method to converge very slowly. The nonlinear conjugate-gradient method displays a range of performance that has not yet been adequately explained. On problems in which the Hessian at the solution has clustered eigenvalues, a nonlinear conjugate-gradient method will sometimes converge more quickly than a quasi-Newton method, whereas on other problems the method will break down, i.e. generate search directions that lead to essentially no progress. For recent surveys of conjugate-gradient methods, see Gill and Murray (1979), Fletcher (1980) and Hestenes (1980).

2.5. The truncated linear conjugate-gradient method. Much recent interest has been focussed on an approach to unconstrained optimization in which the equations (2.1) that define the search direction are "solved" (approximately) by performing a limited number of iterations of the *linear* conjugate-gradient method.

Consider the case in which the exact Hessian is used in (2.1). Dembo, Eisenstat and Steihaug (1982) note that the local convergence properties of Newton's method depend on p_k being an accurate solution of (2.1) *only near the solution of the unconstrained problem.* They present a criterion that defines the level of accuracy in p_k necessary to achieve quadratic convergence as the solution is approached, and suggest systematically "truncating" the sequence of linear conjugate-gradient iterates when solving the linear system (2.1) (hence their name of "truncated Newton method"). (See also Dembo and Steihaug (1980) and Steihaug (1980).)

This idea has subsequently been applied in a variety of situations—for example, in computing a search direction from (2.1) when H_k is a sparse quasi-Newton approximation (Steihaug (1982)). We therefore prefer the more specific name of *truncated conjugate-gradient methods.* These methods are useful in computing search directions when it is impractical to store H_k, but it is feasible to compute a *relatively small* number of matrix-vector products involving H_k. For example, this would occur if H_k were the

product of several sparse matrices whose product is dense (see § 3.3.1). Truncated conjugate-gradient methods have also been used when the matrix-vector product $H_k v$ is *approximated* (say, by a finite-difference along v); in this case, the computation of p_k requires a number of gradient evaluations equal to the number of linear conjugate-gradient iterations (see, e.g., O'Leary (1982)). In order for these methods to be effective, it must be possible to compute a good solution of (2.1) in a small number of linear conjugate-gradient iterations, and hence the use of preconditioning is important.

With a truncated conjugate-gradient method, complications arise when the matrix H_k is not positive definite, since the linear conjugate-gradient method is likely to break down. Various strategies are possible to ensure that p_k is still a well-defined descent direction even in the indefinite case. For example, the conjugate-gradient iterates may be computed using the Lanczos process (Paige and Saunders (1975)); a Cholesky factorization of the resulting tridiagonal matrix leads to an algorithm that is equivalent to the usual iteration in the positive-definite case. If the tridiagonal matrix is indefinite, a related positive-definite matrix can be obtained using a modified Cholesky factorization. Furthermore, preconditioning can be included, in which case the linear conjugate-gradient iterates begin with the negative gradient transformed by the preconditioning matrix. If the preconditioning matrix is a good approximation to the Hessian, the iterates should converge rapidly. Procedures of this type are described in O'Leary (1982) and Nash (1982).

Further flexibility remains as to how the result of a truncated conjugate-gradient procedure may be used within a method for unconstrained optimization. Rather than simply being used as a search direction, for example, p_k may be combined with previous search directions in a *nonlinear* conjugate-gradient method (see Nash (1982)).

3. Linearly constrained optimization.

3.1. Introduction. The linearly constrained problem will be formulated as

$$\text{LCP} \qquad \underset{x \in \Re^n}{\text{minimize}}\ F(x)$$

$$\text{subject to } \mathcal{A}x = b,$$

$$l \leqq x \leqq u,$$

where the $m \times n$ matrix $\mathcal{A}$ is assumed to be large and sparse. For simplicity, we assume that the rows of $\mathcal{A}$ are linearly independent (if not, some of them may be removed without altering the solution).

The most popular methods for linearly constrained optimization are *active-set* methods, in which a subset of constraints (the *working set*) is used to define the search direction. The working set at x_k usually includes constraints that are satisfied exactly at x_k; the search direction is then computed so that movement along p_k will continue to satisfy the constraints in the working set.

With problem LCP, the working set will include the *general* constraints $\mathcal{A}x = b$ and some of the bounds. When a bound is in the working set, the corresponding variable is *fixed* during that iteration. Thus, the working set induces a partition of x into *fixed* and *free* variables.

We shall not be concerned with details of how the working set is altered, but merely emphasize that the fixed variables at a given iteration are effectively removed from the problem; the corresponding components of the search direction will be zero, and thus the columns of $\mathcal{A}$ corresponding to fixed variables may be ignored. Let A_k denote the submatrix of $\mathcal{A}$ corresponding to the free variables at iteration k; each

change in the working set corresponds to a change in the *columns* of A_k. Let n_V denote the number of free variables, and the vector p_k denote the search direction with respect to the free variables only.

By analogy with (2.2) in the unconstrained case, we may choose p_k as the step to the minimum of a quadratic approximation to F, subject to the requirement of remaining on the constraints in the working set. This gives p_k as the solution of the following quadratic program:

$$\begin{aligned} &\underset{p}{\text{minimize}}\ g_k^T p + \tfrac{1}{2} p^T H_k p \\ &\text{subject to } A_k p = 0, \end{aligned} \tag{3.1}$$

where g_k denotes the gradient and H_k the Hessian (or Hessian approximation) at x_k with respect to the free variables.

The solution p_k and Lagrange multiplier λ_k of the problem (3.1) satisfy the $n_V + m$ equations

$$\begin{pmatrix} H_k & A_k^T \\ A_k & \end{pmatrix} \begin{pmatrix} p_k \\ -\lambda_k \end{pmatrix} = \begin{pmatrix} -g_k \\ 0 \end{pmatrix}, \tag{3.2}$$

which will be called the *augmented system.*

One convenient way to represent p_k involves a matrix whose columns form a basis for the null space of A_k. Such a matrix, which will be denoted by Z_k, has $n_V - m$ linearly independent columns and satisfies $A_k Z_k = 0$. The solution of (3.1) may then be computed by solving the *null-space equations*

$$Z_k^T H_k Z_k p_Z = -Z_k^T g_k \tag{3.3}$$

and setting

$$p_k = Z_k p_Z. \tag{3.4}$$

Equations (3.3) and (3.4) define a *null-space* representation of p_k (so named because it explicitly involves Z_k). The vector $Z_k^T g_k$ and the matrix $Z_k^T H_k Z_k$ are called the *projected gradient* and *projected Hessian.*

3.2. Representation of the null space. The issues that arise in representing Z_k when A_k is sparse illustrate the need to compromise strategies that are standard for dense problems. In the rest of this section, we shall drop the subscript k associated with the iteration.

In dense problems, it is customary to use an explicit LQ or some other orthonormal factorization of A in order to define Z. If $AQ = (L\ 0)$, where the orthonormal matrix Q is partitioned as $(Y\ Z)$ and L is lower triangular, then $AZ = 0$. In this case, Z has the "ideal" property that its columns are orthonormal, so that formation of the projected Hessian and gradient does not exacerbate the condition of (3.3) and (3.4). Unfortunately, for large problems computation of such a factorization is normally too expensive. (Some current research is concerned with efficient methods for obtaining sparse orthogonal factorizations; see George and Heath (1981). However, the need to update the factors is an even more serious difficulty; see Heath (1982) and George and Ng (1982).)

If an orthogonal factorization is unacceptable, a good alternative is to reduce A to triangular form using Gaussian elimination (i.e., elementary transformations combined with row and column interchanges). This would give an LU factorization in the

form

$$P_1 A P_2 \begin{pmatrix} U & W \\ & I \end{pmatrix} = (L \quad 0), \tag{3.5}$$

where P_1 and P_2 are permutation matrices, U is unit upper triangular, and L is lower triangular. The matrices P_1 and P_2 would be chosen to make U well-conditioned and $\|W\|$ reasonably small. The required matrix

$$Z = P_2 \begin{pmatrix} W \\ I \end{pmatrix} \tag{3.6}$$

would no longer have orthonormal columns, but should be quite well conditioned, even if A is poorly conditioned.

Unfortunately, it is not known how to update the factorization (3.5) efficiently in the sparse case when columns of A are altered. However, (3.5) indicates the existence of a square, nonsingular submatrix drawn from the rows and columns of A. We shall assume for simplicity that this matrix comprises the left-most columns of A, i.e.

$$A = (B \quad S), \tag{3.7}$$

where B is nonsingular. (In practice, the columns of B may occur anywhere in A.) It follows from (3.7) and (3.5) (with P_1 and P_2 taken as identity matrices) that $BW + S = 0$, so that $W = -B^{-1}S$. Thus, Z has the form

$$Z = \begin{pmatrix} -B^{-1}S \\ I \end{pmatrix}. \tag{3.8}$$

As long as B in (3.7) is nonsingular, the matrix Z (3.8) will provide a basis for the null space of A. In the absence of the ideal factorization (3.5), the aim must be to choose a B that is as well conditioned as conveniently possible, since this will tend to limit the size of $\|W\|$ and hence the condition of Z.

The partition of the columns of A given by (3.7) induces a partition of the free variables, which will be indicated by the subscripts "B" and "S". The m variables x_B are called the *basic* variables. The remaining s free variables ($s = n_V - m$) are called the *superbasic* variables. For historical reasons, the fixed variables are sometimes called the *nonbasic* variables.

An advantage of the form (3.8) for sparse problems is that operations with Z and Z^T may be performed using a factorization of the matrix B; the matrix Z itself need not be stored. For example, the vector $Z^T g$ required in (3.3) may be written as

$$Z^T g = -S^T B^{-T} g_B + g_S. \tag{3.9}$$

(The vector on the right-hand side of (3.9) is called the *reduced gradient*; note that it is simply the projected gradient with a particular form of Z.) Thus, $Z^T g$ may be obtained by solving $B^T v = g_B$, and then forming $g_S - S^T v$. Similarly, to form $p = Zp_Z$, we have

$$p = \begin{pmatrix} -B^{-1}S \\ I \end{pmatrix} p_Z = \begin{pmatrix} -B^{-1}Sp_Z \\ p_Z \end{pmatrix},$$

which gives the system

$$Bp_B = -Sp_Z.$$

With the *reduced-gradient form of* Z (3.8), the problems of representing a null space and computing the associated projections reduce to the familiar operations of factorizing and solving with an appropriate square B.

3.3. Solving for the search direction. At each iteration of an active-set method for LCP, the search direction p with respect to the free variables solves the subproblem (3.1). We have seen that there are mathematically equivalent representations of p; the way in which p is *computed* for sparse problems depends on several considerations, which will be discussed below.

3.3.1. Solving the null-space equations. For sparse problems, it will generally not be possible to solve (3.3) by explicitly forming and then factorizing Z^THZ. Even if H and B are sparse, the projected Hessian will generally be dense. Thus, if a factorization of the projected Hessian is to be stored, the number of superbasic variables at each iteration must be sufficiently small (i.e., the number of fixed variables must be sufficiently large). Fortunately, for many large-scale problems there is an *a priori* upper bound on the number of free variables. For example, if only q of the variables appear nonlinearly in the objective function, the dimension of the projected Hessian matrix at the solution cannot exceed q.

Furthermore, even if the dimension of Z^THZ is small, *forming* the projected Hessian may involve a substantial amount of work; when Z is defined by (3.8), computation of Z^THZ requires the solution of $2s$ systems of size $m \times m$. For this reason, a Newton-type method in which the *projected* Hessian is recomputed at each iteration is not generally practical. By contrast, *quasi-Newton methods* can be adapted very effectively to sparse problems in which the dimension of the projected Hessian remains small, by updating a dense Cholesky factorization of a quasi-Newton approximation to the *projected Hessian*; this is the method used in the MINOS code of Murtagh and Saunders (1977), (1980).

When the projected Hessian cannot be formed or factorized, the null-space equations may be solved using an iterative method that does not require storage of the matrix, such as a truncated conjugate-gradient method (see § 2.5). In order for this approach to be reasonable, the computation of matrix-vector products involving Z and H must be relatively cheap (e.g., when H is sparse); in addition, a good approximation to the solution of (3.3) must be obtained in a small number of iterations. Even when the Hessian is not available, a truncated conjugate-gradient method may be applied to (3.3) by using a finite-difference of the gradient to approximate the vector HZv; an evaluation of the gradient is thus necessary for every iteration of the truncated conjugate-gradient method. Note that this is one of the few methods in which H is not required to be sparse.

Each of the above methods for solving the null-space equations can be adapted to allow for changes in the working set (§ 3.5).

3.3.2. Solving the range-space equations. The null-space equations provide one means of solving for p in the augmented system (3.2), by eliminating λ_k. When H is positive definite, a complementary approach is to solve for λ first, via the *range-space equations*

$$AH^{-1}A^T\lambda = AH^{-1}g, \qquad Hp = A^T\lambda - g.$$

This method would be appropriate if H were sparse, and if A had relatively few rows. The application of a range-space approach to quadratic programming is discussed by Gill et al. (1982).

3.3.3. Solving the augmented system. An alternative method for obtaining p involves treating the augmented system directly. (Variations of this idea have been proposed by numerous authors; see, e.g., Bartels, Golub and Saunders (1970)). The most obvious way to solve (3.2) is to apply a method for symmetric indefinite systems, such as the Harwell code MA27 (Duff and Reid (1982)). In order for the solution of (3.2) to be meaningful, the matrix Z^THZ must be positive definite. Verifying positive-definiteness in this situation is a nontrivial task, since of course the matrix Z^THZ is not computed explicitly. However, the result may sometimes be known a priori—for example, when H itself is positive-definite.

Both H and A change dimension when the working set is altered. Updating procedures for this case are discussed in § 3.6.2.

3.4. Factorizing and solving a square system. The linear systems involving B and B^T are typically solved today using a sparse LU factorization of B. Surveys of techniques for computing such a factorization are given in Duff (1982) and Duff and Reid (1983). The *analyze phase* of a factorization consists of an analysis of the sparsity pattern alone (independent of the values of the elements), and leads to a permutation of the matrix in order to reduce fill-in during the factorization. The *factor phase* consists of computation with the actual numerical elements of the matrix.

We shall mention a few features of certain factorization methods that have particular relevance to optimization (see Duff and Reid (1983) for more details). Since active-set algorithms include a sequence of matrices that undergo column changes, the factorization methods were typically developed to be used in conjunction with an update procedure.

The P^4 algorithm of Hellerman and Rarick (1971), (1972) performs the analyze phase separately from the factor phase, and produces the well-known "bump and spike" structure, in which B is permuted to block lower-triangular form with relatively few "spikes" (columns containing nonzeros above the diagonal). This procedure is very effective if B is nearly triangular. Also, the factor phase is able to use external storage, since it processes B one column at a time. Column interchanges are used to stabilize the factorization. (Row interchanges would destroy the sparsity pattern.) If an interchange is needed at the ith stage, it is necessary to solve a system of the form $L_{i-1}^T y = e_i$ and to compute the quantities $y^T a_j$ for all remaining eligible spike columns a_j. This involves significant work and also degrades the sparsity of the factors. Thus, a rather loose pivot tolerance must be used to avoid many column interchanges (e.g., $|\mu| \leqq 10^4$, where μ is the largest subdiagonal element in any column of L divided by the corresponding diagonal).

The *Markowitz* algorithm (Markowitz (1957)), on the other hand, performs the analyze and factor phases simultaneously, and hence must run in main memory. It computes dynamic "merit counts" in order to determine the row and column permutations to preserve sparsity and yet retain numerical stability. The Markowitz procedure can achieve a good sparse factorization even with a rather strict pivot tolerance (e.g., $|\mu| \leqq 10$).

In order to indicate how these factor routines perform on matrices that arise in optimization, we give results on five test problems. In the first three problems, the matrix B has "staircase" structure (see, e.g., Fourer (1982)); constraints of this form often arise in the modeling of dynamic systems, in which a set of activities is replicated over several time periods. The fourth and fifth problems arise from the optimal power flow (OPF) problem (see e.g., Stott, Alsac and Marinho (1980)). In this case, B is the Jacobian of the network equations of the power system, and has a *symmetric* sparsity

TABLE 1
Summary of problem characteristics.

	Stair 1	Stair 2	Stair 3	OPF 1	OPF 2
B rows	357	745	1,170	1,200	3,400
B nonzeros	3,500	3,600	7,100	9,000	29,000
P^4 blocks	1	5	13	1	—
P^4 spikes	66	101	157	715	—

pattern (which is not at all triangular!) Table 1 shows some of the relevant features of the problems described, including the results of factorization with the P^4 algorithm.

The number of nonzeros in the initial LU factorization of B is shown in the first two rows of Table 2. The P^4 algorithm is as implemented in the MINOS code of Murtagh and Saunders (1977), (1980); the Markowitz procedure is the Harwell code LA05 (Reid (1976), (1982)). Note that the large number of spikes in the first OPF problem is bound to cause difficulties for the P^4 algorithm.

TABLE 2
Number of nonzeros in initial LU factorization and after k updates.

	Stair 1	Stair 2	Stair 3	OPF 1	OPF 2
L_0U_0 with P^4 (MINOS)	9,400	16,200	32,000	30,400	—
L_0U_0 with Markowitz (LA05)	5,400	4,700	13,500	13,800	75,000
k	50	50	50	30	40
L_kU_k with LA05	7,800	6,000	17,100	15,300	83,000

3.5. Column updates. For problems of the form LCP, each change in the working set involves changing the status of a variable from fixed to free (or vice versa). When a previously fixed variable becomes free, a column of $\mathscr{A}$ is added to A; this poses no particular difficulty, since the new column can simply be appended to S. When a free variable is to become fixed, a column of A must be deleted, and complications arise if the column is in B. Since the number of columns in B must remain constant (in order for B to be nonsingular), it is necessary to *replace* a column of B with one of the columns of S.

Assume that we are given an initial B_0, which thereafter undergoes a sequence of column replacements, each corresponding to one of the free variables becoming fixed on a bound. Let l_k denote the index of the column to be replaced at the kth step, a_k denote the l_kth column of B, v_k denote the new column, and e_{l_k} denote the l_kth column of the identity matrix. After each replacement, we have

$$B_k = B_{k-1} + (v_k - a_k)e_{l_k}^T. \tag{3.10}$$

We shall consider several ways in which systems of equations involving B_k can be solved following a sequence of such changes.

3.5.1. The product-form update. The standard updating technique used in all early sparse LP codes was the product-form (PF) update (e.g., Dantzig and Orchard-Hays (1954)). It follows from the definition of B_k that

$$B_k = B_{k-1}T_k,$$

where

$$B_{k-1}y_k = v_k \quad \text{and} \quad T_k = I + (y_k - e_{l_k})e_{l_k}^T. \tag{3.11}$$

Note that T_k is a permuted triangular matrix (with only one nontrivial column); equivalently, T_k is a rank-one modification of the identity matrix. The matrix T_k can be represented by storing the index l_k and the vector y_k.

After k such updates we have

$$B_k = B_0T_1T_2 \cdots T_k. \tag{3.12}$$

Given a procedure to solve systems of equations involving B_0, (3.12) indicates that solving $B_kv = b$ is equivalent to solving the $k+1$ linear systems

$$B_0v_0 = b, \quad T_1v_1 = v_0, \quad \cdots, \quad T_kv_k = v_{k-1}, \tag{3.13}$$

where the systems involving T_j are easy to solve. As k increases, the solution process becomes progressively more protracted, and the storage required to store the updates is strictly increasing. Therefore it becomes worthwhile to compute a factorization of B_k from scratch. Most current systems use an initial triangular factorization $B_0 = L_0U_0$ (see § 3.4), and recompute the factorization after k updates (typically $k \leqq 50$).

The PF update has two important advantages for sparse problems. First, the vectors $\{y_j\}$ may be stored in a single sequential file, so that implementation is straightforward. Second, any advance in the methods for linear equations is immediately applicable to the factorization of B_0, since the update does not alter the initial factorization. Thus, B_0 may be represented by a "black box" procedure for solving equations (involving both B_0 and B_0^T).

Unfortunately, the PF update has two significant deficiencies. It is numerically unreliable if $|e_{l_k}^T y_k|$ is too small (since T_k is then ill-conditioned), and the growth of data defining the updates is significantly greater than for alternative schemes.

3.5.2. The Bartels–Golub update. The instability of the PF update was first made prominent by Bartels and Golub (1969), who showed as an alternative that an LU factorization can be updated in a stable manner (see also Bartels, Golub and Saunders (1970); Bartels (1971)). Given an initial factorization $B_0 = L_0U_0$, the updates to L are represented in product form, but the sparse triangular matrix U is stored (and updated) *explicitly*. Thus, instead of the form (3.12) we have

$$B_k = L_0T_1T_2 \cdots T_qU_k \equiv L_kU_k, \tag{3.14}$$

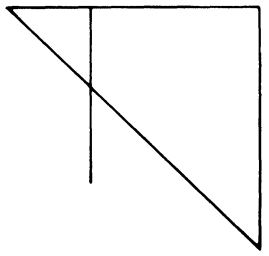
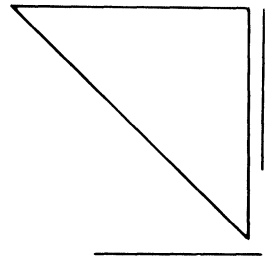
where each T_j represents an update whose construction will be discussed below.

At the kth step, replacing the l_kth column of B_{k-1} gives

$$B_k = L_{k-1}\tilde{U},$$

where $\tilde{U}$ is identical to U_{k-1} except for its l_kth column. Since U_{k-1} is stored as a sparse matrix, it is desirable to restore $\tilde{U}$ to upper-triangular form U_k without causing substantial fill-in. To this end, let P denote a cyclic permutation that moves the l_kth row and column of $\tilde{U}$ to the end, and shifts the intervening rows and columns forward.

We then have

$\tilde{U} =$  , $P^T\tilde{U}P =$  .

The nonzeros in the bottom row of $P^T\tilde{U}P$ may be eliminated by adding multiples of the other rows. However, it follows from the usual error analysis of Gaussian elimination (e.g., Wilkinson (1965)) that this procedure will not be numerically stable unless the size of the multiple is bounded in some way. Hence, we must allow the last row to be *interchanged* with some other row. Formally, the row operations are stabilized elementary transformations (Wilkinson (1965)), which are constructed from 2×2 matrices of the form

$$M = \begin{pmatrix} 1 & \\ \mu & 1 \end{pmatrix} \quad \text{or} \quad \tilde{M} = \begin{pmatrix} & 1 \\ 1 & \mu \end{pmatrix}. \tag{3.15}$$

(Note that the transformation $\tilde{M}$ includes a row interchange.) Each such transformation is represented by the scalar μ, and is unnecessary if the element to be eliminated is already zero. Numerical stability is achieved by choosing between M and $\tilde{M}$ so that the multiplier μ is bounded in size by some moderate number (e.g., $|\mu| \leqq 1$, 10 or 100). The matrices $\{T_j\}$ in (3.14) are constructed from sequences of matrices of the form (3.15).

Unfortunately, elimination of the nonzeros is "easier said than done" in the sparse case. Any transformation of type $\tilde{M}$ amounts to a form of fill-in, since the location of nonzeros in the interchanged rows is unlikely to be the same. A complex data structure is therefore needed to update U_k without losing efficiency during subsequent solves. (Holding individual nonzeros in a linked list, for example, would not be acceptable in a virtual-memory environment.)

The implementation of the BG update by Saunders (1976) capitalizes on the "bump and spike" structure revealed by the P^4 procedure (see § 3.4). Each triangular factor is of the form

$$U_k = \begin{pmatrix} I & E_k \\ & F_k \end{pmatrix},$$

and fill-in can occur only within F_k. If U_0 contains s spikes, the dimension of F_k will be at most $s+k$. Storing F_k as a dense matrix allows the BG update to be implemented with maximum stability ($|\mu| \leqq 1$ in (3.15)), and the approach is efficient as long as s is not unduly large (say, $s \leqq 100$). This implementation has been used for several years in the nonlinear programming system MINOS (Murtagh and Saunders (1977), (1980)). During that period, the number of spikes in U_0 has proved to be favorably small for many sparse optimization models. However, two important applications are now known to give unacceptably large numbers of spikes: time-period models (for which B has a staircase structure) and optimal power-flow problems (for which B has a symmetric sparsity pattern). Some statistics for these problems are given in Table 1 (§ 3.4).

Another implementation of the BG update has been developed by Reid (1976), (1982) as the Fortran package LA05 in the Harwell Subroutine Library. It strikes a compromise between dense and linked-list storage by using a whole row or column of U_k as the "unit" of storage. Thus, the nonzeros in any one row of U_k are held in contiguous locations of memory, as are the corresponding column indices, and an ordered list points to the beginning of each row. To facilitate searching, a similar data structure is used to hold just the sparsity pattern of each column (i.e., the row indices are stored, but not the nonzeros themselves; see Gustavson (1972)). This storage scheme is also suitable for computing an initial LU factorization using the Markowitz criterion and threshold pivoting—a combination that has been eminently successful in practice, particularly on the structures mentioned above. Table 2 (§ 3.4) shows the sparsity of various initial factorizations $B_0 = L_0 U_0$ computed by subroutine LA05A, and the moderate rate of growth of nonzeros following k calls to the BG update subroutine LA05C.

Given the row-wise storage scheme for the nonzeros of U_0, it was natural in LA05A for the stability test to be applied *row-wise*. (Thus, each diagonal of U_0 must not be too small compared to other nonzeros in the same row.) This standard threshold pivoting rule is appropriate for single systems, but unfortunately is at odds with the aim of the BG update. The effect is to control the condition of U_0, with no control on the size of the multipliers μ defining L_0.

A preferable alternative is to apply the threshold pivoting test *column-wise*, in order to control the condition of L_0. The resulting L_0, and hence all subsequent factors L_k, will then be a product of stabilized transformations T_j. It follows that *the factors of B_k are likely to be well conditioned if B_k is well conditioned, even if B_0 is not.*

In order to apply the column-wise stability test efficiently, the data structure for computing U_0 needs to be transposed. This and other improvements will be incorporated in a new version of LA05 (Reid, private communication).

At the Systems Optimization Laboratory we have recently implemented some analogous routines as part of a package LUSOL, which will maintain the factorization $L_k B_k = U_k$ following various kinds of updates. The matrices B_k may be singular or rectangular, and the updates possible are column replacement, row replacement, rank-one modification, and addition or deletion of a row or a column. The condition of L_k is controlled throughout for the reasons indicated above. We expect such a package to find many applications within optimization and elsewhere. One example will be to maintain a sparse factorization of the Schur-complement matrix C_k (see §§ 3.5.4–3.6.2), often called the *working basis* in algorithms for solving mathematical programs that have special structure. GUB rows and imbedded networks are examples of such structure; see Brown and Wright (1981) for an excellent overview.

3.5.3. The Forrest–Tomlin update. The update of Forrest and Tomlin (1972) was developed as a means of improving upon the sparsity of the PF update while retaining the ability to use external storage where necessary. In fact the FT update is a restricted form of the BG update, in which no row interchanges are allowed when eliminating the bottom row of $P^T \tilde{U} P$. This single difference removes the fill-in difficulty (but at the expense of losing guaranteed numerical stability).

Algebraically, a new column w_k is added to U_{k-1}, the l_kth column and row are deleted, and the transformations M are combined into a single "row" transformation $R_k = I + e_{l_k}(r_k - e_{l_k})^T$. It can be shown that the required vectors satisfy

$$L_{k-1} w_k = v_k, \quad \text{and} \quad U_{k-1}^T r_k = e_{l_k}, \tag{3.16}$$

and the new diagonal of U_k is $r_k^T w_k$. Most importantly, the multipliers μ are closely related to the elements of r_k, and these can be tested *a posteriori* to determine whether the update is acceptable (see also Tomlin (1975)). In practice a rather undemanding test such as $|\mu| \leqq 10^6$ must be used to avoid rejecting the update too frequently. The FT update is now used within several commercial mathematical programming systems.

3.5.4. Use of the Schur complement. The work of Bisschop and Meeraus (1977), (1980) has recently provided a new perspective on the problem of updating within active-set methods. Suppose that for each update a vector v_j replaces the l_jth column of B_0. A key observation is that the system $B_k x = b$ is equivalent to the system

$$\begin{pmatrix} B_0 & V_k \\ I_k & \end{pmatrix}\begin{pmatrix} y \\ z \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix}, \tag{3.17}$$

where

$$V_k = (v_1\ v_2 \cdots v_k), \qquad I_k = (e_{l_1}\ e_{l_2} \cdots e_{l_k})^T.$$

Note that the rectangular matrix I_k is composed of k rows of the identity matrix corresponding to indices of columns that have been replaced. Since the equations $I_k y = 0$ set k elements of y to zero, the remaining elements of y and z together give the required solution x. Similarly, the system $B_k^T y = d$ is equivalent to

$$\begin{pmatrix} B_0^T & I_k^T \\ V_k^T & \end{pmatrix}\begin{pmatrix} y \\ z \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \end{pmatrix} \tag{3.18}$$

if d_1 and d_2 are constructed from d appropriately (with the aid of k arbitrary elements, such as zero).

The matrix in (3.17) may be factorized in several different ways. In the next two sections we consider the simplest factorization

$$\begin{pmatrix} B_0 & V_k \\ I_k & \end{pmatrix} = \begin{pmatrix} B_0 & \\ I_k & C_k \end{pmatrix}\begin{pmatrix} I & Y_k \\ & I \end{pmatrix} \tag{3.19}$$

where

$$B_0 Y_k = V_k, \qquad C_k = -I_k Y_k. \tag{3.20}$$

The $k \times k$ matrix C_k is the *Schur complement* for the partitioned matrix on the left-hand side of (3.19). It corresponds to a matrix of the ubiquitous form $D - WB^{-1}V$ (e.g., see Cottle (1974)).

3.5.5. A stabilized product-form update. From (3.17) and (3.19) we see that the vectors y and z needed to construct the solution of $B_k x = b$ may be obtained from the equations

$$B_0 w = b, \tag{3.21a}$$

$$C_k z = -I_k w, \tag{3.21b}$$

$$y = w - Y_k z. \tag{3.21c}$$

Similarly, the solution of $B_k^T y = d$ is obtained from the two linear systems

$$C_k^T z = d_2 - Y_k^T d_1, \tag{3.22a}$$

$$B_0^T y = d_1 - I_k^T z. \tag{3.22b}$$

Assuming that Y_k is available, the essential operations in (3.21) and (3.22) are a solve

with B_0 and a solve with C_k. If k is small enough (say, $k \leqq 100$), C_k may be treated as a dense matrix. It is then straightforward to use an orthogonal factorization $Q_k C_k = R_k (Q_k^T Q_k = I, R_k$ upper triangular) or an analogous factorization $L_k C_k = U_k$ based on Gaussian elimination (L_k square, U_k upper triangular). These factorizations can be maintained in a stable manner as C_k is updated to reflect changes to B_k. (The updates involve adding and deleting rows and columns of C_k; see Gill et al. (1974).) The stability of the procedures (3.21) and (3.22) then depends essentially on the condition of B_0. In other words, if B_0 is well conditioned, we have a stable method for solving $B_k x = b$ for many subsequent k.

The method retains several advantages of the PF update. The vectors to be stored (columns of Y_k) satisfy $B_0 y_k = v_k$, which is analogous to (3.11). These vectors should have sparsity similar to those in the PF update, and they can be stored sequentially (in compact form on an external file, if necessary). A further advantage is that whenever a column of C_k is deleted, the corresponding vector y_k may be skipped in subsequent uses of (3.21c). This gain would tend to offset the work involved in maintaining the factors of C_k. Because of the parallels, the method described here amounts to a practical mechanism for stabilizing an implementation based on the PF update.

3.5.6. The Schur-complement update. One of the aims of Bisschop and Meeraus (1977), (1980) was to give an update procedure whose storage requirements were independent of the dimension of B_0. This is achievable because the matrix Y_k is not essential for solving (3.17) and (3.18), given V_k and a "black box" for B_0. For example, (3.21c) may be replaced by

$$B_0 y = b - V_k z, \tag{3.23}$$

and hence storage for Y_k can be saved at the expense of an additional solve with B_0. Similarly, (3.22a) is equivalent to

$$B_0^T w = d_1, \qquad C_k^T z = d_2 - V_k^T w,$$

again involving a second solve with B_0. Note that the original data V_k will usually be more sparse than Y_k, so that the additional expense may not be substantial.

The storage required for a dense orthogonal factorization of C_k ($\frac{3}{2}k^2$) is small for moderate values of k. As with the PF update, any advance in solving linear equations is immediately applicable to the equations involving B_0.

The method is particularly attractive when B_0 has special structure. For example, certain linear programs have the following form:

$$\text{minimize } c^T x$$

$$\text{subject to } (B_0 \quad N)x = b,$$

$$l \leqq x \leqq u,$$

where B_0 is a square block-diagonal matrix:

$$B_0 = \text{block-diag}\,(D_0\, D_1 \cdots D_N).$$

Assuming that the square matrices D_j are well conditioned, B_0 provides a natural starting basis for the simplex method.

With the Schur-complement (SC) update, an iteration of the simplex method on such a problem requires four solves with B_0, and hence four solves with each matrix D_j. In certain applications, the matrices D_j are closely related to D_0 (e.g., in time-dependent problems), in which case a further application of the Schur-complement technique would be appropriate. A simplex iteration then involves only solves with D_0.

This is a situation in which one factorization is followed by hundreds or even thousands of solves (involving both D_0 and D_0^T). Thus, it is useful for black-box solvers to be tuned to the case of multiple right-hand sides.

3.5.7. The partitioned *LU* update. Recall that the PF approach accumulates updates in a single file, while the BG and FT methods seek to reduce the storage required for the updates by updating two separate factors (one implicitly through a file of updates, the other explicitly). Here we suggest leaving L_0 and U_0 unaltered (in effect, treating them as two "black boxes" for solving linear systems), and accumulating *two* files of updates. In place of the block factorization (3.19) we can write

$$\begin{pmatrix} B_0 & V_k \\ I_k & \end{pmatrix} = \begin{pmatrix} L_0 & \\ R_k & C_k \end{pmatrix} \begin{pmatrix} U_0 & W_k \\ & I \end{pmatrix} \tag{3.24}$$

with the same definition (3.20) of C_k. After the kth update, the new column of W_k and row of R_k satisfy

$$L_0 w_k = v_k \quad \text{and} \quad U_0^T r_k = e_{l_k}. \tag{3.25}$$

The similarity of (3.25) with the equations (3.16) for the FT update leads us to suppose that the storage requirements would be at least as low as for the FT update. Apart from the need to store and update C_k, all implementation advantages are retained (in fact improved upon, since U_0 is not altered). As with the PF and SC updates, the stability depends primarily on the condition of B_0. We could therefore regard the factorization (3.24) as a practical and stable alternative to the FT update.

3.5.8. Avoiding access to B_0. In active-set methods, it is often necessary to solve the equations $B_k x = v$, where v is a column of the matrix $\mathscr{A}$. Although v will not be a column of B_k, it *could* be a column of B_0. If B_0 were not stored in main memory, it would be desirable to access its columns as seldom as possible. In this section we shall show that with the PF update or the Schur-complement updates, the elements of B_0 need not be accessed once the initial factorization has been completed.

Assume that v is the lth column of B_0, so that $v = B_0 e_l$ by definition. For the PF update it follows by substituting the expression for v in (3.13) that

$$T_1 \cdots T_k x = e_l,$$

which gives an equation for x that does not involve v or B_0. With the Schur-complement approach, (3.21a) reduces to $w = e_l$, while (3.23) can be rearranged to give $B_0(y - e_l) = -V_k z$. In either case, when solving for x we can avoid not only an explicit reference to the elements of B_0 but also a solve with B_0.

Similarly, it is often necessary to solve $B_k^T y = d$ and then to form $\gamma_j = y^T v_j$ for each column v_j that has been replaced in B_0. (The quantities γ_j are the reduced costs or reduced gradients for variables that have been removed from B_0.) If t denotes the product $B_0^T y$, then by definition of v_j it follows that $y^T v_j = t^T e_{l_j}$. With both the PF and the Schur-complement updates, t is a by-product of the procedure for computing y. Thus, t and all relevant values γ_j are available at no cost.

These results confirm that B_0 need exist only in the form of a "black box" for solving linear systems.

3.6. Other applications of the Schur-complement update. Historically, the formulation LCP has been used because it involves only column updates to B_k, which have appeared to be the least difficult kind of update to implement for sparse problems. However, the Schur-complement approach also applies to more general sequences of

related square systems. As with column replacement, the key is to solve a partitioned system that involves the original matrix.

3.6.1. Unsymmetric rank-one updates. Consider the case in which B_0 undergoes a sequence of rank-one modifications:

$$B_k = B_{k-1} + v_k s_k^T \equiv B_0 + V_k S_k^T.$$

The solution of $B_k x = b$ is part of the solution of the extended system

$$\begin{pmatrix} B_0 & V_k \\ S_k^T & -I \end{pmatrix} \begin{pmatrix} x \\ z \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix} \tag{3.26}$$

(Kron (1956), Bisschop and Meeraus (1977)). Given factorizations of B_0 and the Schur complement $C_k \equiv -I - S_k^T B_0^{-1} V_k$, the solution may be obtained from

$$C_k z = -S_k^T w, \qquad B_0 x = b - V_k z,$$

where $B_0 w = b$. An alternative that would require more storage but less work could be obtained by using $B_0 = L_0 U_0$ and storing the vectors defined by $L_0 w_k = v_k$, $U_0^T r_k = s_k$. Let R_k denote the matrix whose jth column is r_j, and similarly for W_k. In this case, the solution of (3.26) would be obtained from

$$C_k z = -R_k^T v \qquad U_0 x = v - W_k z,$$

where $L_0 v = b$. Either approach is an alternative to updating a factorization of B_k itself (e.g., Gille and Loute (1981), (1982)), which is even more difficult to implement than the BG update.

We emphasize that column or row replacements are best treated as a special case, *not* as a sequence of general rank-one modifications.

3.6.2. A symmetric Schur-complement update. It was observed in § 3.1 that in some circumstances the search direction can be computed by solving the linear system (3.2) involving the augmented matrix

$$M_k = \begin{pmatrix} H_k & A_k^T \\ A_k & \end{pmatrix}. \tag{3.27}$$

Within an active-set method, changes in the status of fixed and free variables lead to changes in H and A. When a variable becomes fixed, the corresponding row and column of M_k are *deleted*; when a variable is freed, a new row and column of M_k are *added.*

Instead of updating a factorization of M_k, we can start with some M_0 and work with an augmented system of the form

$$\begin{pmatrix} M_0 & S_k \\ S_k^T & \end{pmatrix}.$$

If a variable is fixed at the kth change, the kth column of S_k is an appropriate coordinate vector; if the lth variable is freed, the column is

$$s_k = \begin{pmatrix} h_l \\ a_l \end{pmatrix},$$

where h_l is obtained from the lth column of the full Hessian, and a_l is the lth column of $\mathscr{A}$. The solution of the augmented system corresponding to the kth working set can then be obtained using a factorization of M_0 and a factorization of the Schur complement $C_k = -S_k^T M_0^{-1} S_k$.

3.7. Linear and quadratic programming. Two important special cases of LCP are linear and quadratic programs. Since there are no user-supplied functions, the computation in linear and quadratic programming methods involves primarily linear algebraic operations.

3.7.1. Large-scale linear programming. Large-scale linear programs occur in many important applications, such as economic planning and resource allocation. Methods and software for large-scale LP have thus achieved a high level of sophistication, and many of the techniques discussed in § 3 were designed originally for use within the simplex method.

Much research has involved linear programs with special structure in the constraint matrix—for example, those arising from networks or time-dependent systems. It is impossible to summarize methods for specially-structured linear programs in a survey paper of this type. However, to illustrate the flavor of the work, we consider staircase linear programs (which were used in the examples of § 3.4). These arise in modeling time-dependent processes; the recent book edited by Dantzig, Dempster and Kallio (1981) is entirely devoted to such problems. It has long been observed that the simplex method tends to be less efficient on staircase problems than on general LPs. To correct this deficiency, work has tended to proceed in two directions. First, the simplex method can be adapted to take advantage of the staircase structure, by using special techniques for factorizing, updating, and pricing (Fourer (1982)). Second, special-purpose methods can be designed to exploit particular features of the problem. For staircase problems, several variations of the *decomposition* approach (Dantzig and Wolfe (1960)) have been suggested. The basic idea is to solve the problem in terms of smaller, nearly independent, subproblems.

3.7.2. Large-scale quadratic programming. A general statement of the quadratic programming problem is

$$\begin{aligned} &\underset{x\in\mathfrak{R}^n}{\text{minimize}}\ c^Tx+\tfrac{1}{2}x^THx \\ &\text{subject to } \mathscr{A}x=b, \\ &\qquad\qquad l\leqq x\leqq u, \end{aligned}$$

where H is a symmetric matrix.

An early approach to quadratic programming was to transform the problem into a linear program, which is then solved by a modified LP method (e.g., Beale (1967)). The most popular quadratic programming algorithms are now based on the active-set approach described in § 3.1 (for a comprehensive survey of QP methods, see Cottle and Djang (1979)), and the search direction is defined by the subproblem (3.1). Efficient methods for *sparse* quadratic programs thus involve specializing the techniques discussed in § 3.3 for the special case when the Hessian is *constant.*

4. Nonlinearly constrained optimization. The nonlinearly constrained optimization problem is assumed to be of the following form:

NCP

$$\begin{aligned} &\underset{x\in\mathfrak{R}^n}{\text{minimize}}\ F(x) \\ &\text{subject to } c(x)=0, \\ &\qquad\qquad l\leqq x\leqq u, \end{aligned}$$

where $c(x)$ is a vector of m nonlinear constraint functions. We shall assume that these

constraints are "sparse", in the sense that the $m \times n$ Jacobian matrix $A(x)$ of $c(x)$ is sparse. For simplicity, we shall usually not distinguish between linear and nonlinear constraints in $c(x)$. However, it is usually considered desirable to treat linear and nonlinear constraints separately.

Problems with nonlinear constraints are considerably more difficult to solve than those with only linear constraints. There is an enormous literature concerning methods for nonlinear constraints; recent overviews are given in Fletcher (1981) and Gill, Murray and Wright (1981). In this section, we shall concentrate on the impact of sparsity rather than attempt a thorough discussion of the methods.

One aspect of NCP that is directly relevant to sparse matrix techniques is that any superlinearly convergent algorithm must consider the curvature of the nonlinear constraint functions, and thus the Hessian of interest is the Hessian of the *Lagrangian function* rather than the Hessian of F alone. Let the Hessian of the Lagrangian function be denoted by $W(x, \lambda) \equiv H(x) - \sum_{i=1}^{m} \lambda_i H_i(x)$, where H_i is the Hessian of c_i. At first, it might appear unlikely that this matrix would be sparse, since it is a weighted sum of the Hessians of the objective function and the constraints. However, sparsity in the gradient of a nonlinear constraint *always* implies sparsity in its Hessian matrix. For example, if the gradient of $c_i(x)$ contains five nonzero components, the corresponding Hessian matrix $H_i(x)$ can have at most 25 nonzero elements. Furthermore, there is often considerable overlap in the positions of nonzero elements in the Hessians of different constraints. Thus, *in practice the Hessian of the Lagrangian function is often very sparse.*

The usual approach to solving NCP is to construct a sequence of unconstrained or linearly constrained subproblems whose solutions converge to that of NCP. Early methods included unconstrained subproblems based on penalty and barrier functions (see Fiacco and McCormick (1968)). Unfortunately, these methods suffer from inevitable ill-conditioning; they have for the most part been superseded by more efficient methods.

4.1. Augmented Lagrangian methods. Augmented Lagrangian methods were motivated in large part by the availability of good methods for unconstrained optimization. The original idea was to minimize an approximation to the Lagrangian function that has been suitably augmented (by a penalty term) so that the solution is a local *unconstrained* minimum of the augmented function (Hestenes (1969), Powell (1969)).

In particular, an augmented Lagrangian method can be defined in which x_{k+1} is taken as the solution of the subproblem

$$\begin{aligned} &\underset{x \in \mathfrak{R}^n}{\text{minimize}}\ L_A(x, \lambda_k, \rho_k) \\ &\text{subject to } l \leqq x \leqq u, \end{aligned} \tag{4.1}$$

where the *augmented Lagrangian function* L_A is defined by

$$L_A(x, \lambda, \rho) \equiv F(x) - \lambda^T c(x) + \frac{\rho}{2} c(x)^T c(x). \tag{4.2}$$

The vector λ_k is an estimate of the Lagrange multiplier vector, and ρ_k is a suitably chosen nonnegative scalar. Alternatively, it is possible to treat any general linear constraints by an active-set method (§ 3.1), and to include only *nonlinear* constraints in the augmented Lagrangian function. Whatever the definition of the subproblem, the algorithm has a *two-level* structure—"outer" iterations (corresponding to different subproblems) and "inner" iterations (within each subproblem).

The Hessian of interest when solving (4.1) is the Hessian of L_A (4.2), which is $W(x, \lambda_k)+\rho_k A(x)^T A(x)$. The sparsity patterns of $W(x, \lambda)$ and the Hessian matrix of L_A are sometimes very similar. Hence, techniques designed to use an explicit sparse Hessian may be applied to (4.1).

The Jacobian matrix $A(x)$ need not be stored explicitly in order to solve the subproblem (4.1). If a fairly accurate solution of (4.1) is computed, an improved Lagrange multipler estimate may be obtained without solving any linear systems involving $A(x)$. However, in several recent augmented Lagrangian methods, (4.1) is solved only to *low accuracy* in order to avoid expending function evaluations when λ_k is a poor estimate of the optimal multipliers; in this case, some factorization of the matrix $A(x_{k+1})$ *is* required to obtain an improved Lagrange multiplier estimate (by solving either a linear system or a linear least-squares problem). The relevance of the storage needed for the Jacobian and/or a factorization depends on the number of nonlinear constraints and the sparsity of the Jacobian.

4.2. Linearly constrained subproblems. The solution of NCP is a minimum of the Lagrangian function in the subspace defined by the gradients of the active constraints. This property leads to a class of methods in which linearizations of the nonlinear constraints are used to define a *linearly constrained subproblem*, of the form

$$\begin{aligned} &\underset{x\in\mathfrak{R}^n}{\text{minimize}}\ F(x)-\lambda_k^T(c(x)-A_k x) \\ &\text{subject to } A_k(x-x_k)=-c_k, \\ &l\leqq x\leqq u, \end{aligned} \tag{4.3}$$

where c_k and A_k denote $c(x_k)$ and $A(x_k)$ (Robinson (1972), Rosen and Kreuser (1972)). With this formulation, the Lagrange multipliers of the kth subproblem may be taken as the multiplier estimate λ_{k+1} in defining the next subproblem, and will converge to the true multipliers at the solution. When $c(x)$ contains both linear and nonlinear functions, only the nonlinear functions need be included in the objective function of (4.3). Under suitable assumptions, the solutions of the subproblems converge *quadratically* to the solution of NCP. A further benefit of the subproblem (4.3) is that linear constraints may be treated explicitly.

One of the important conditions for convergence with the subproblems (4.3) is a "sufficiently close" starting point; thus, some procedure must be used to prevent divergence from a poor value of x_0. Rosen (1980) suggested a two-phase approach, starting with a penalty function method. In the MINOS/AUGMENTED system of Murtagh and Saunders (1982), the objective function of the subproblem is defined as a *modified augmented Lagrangian* of the form

$$\tilde{L}_A(x, \lambda_k, \rho_k)\equiv F(x)-\lambda_k^T\tilde{c}_k(x)+\frac{\rho_k}{2}\tilde{c}_k(x)^T\tilde{c}_k(x), \tag{4.4}$$

where

$$\tilde{c}_k(x)=c(x)-(c_k+A_k(x-x_k)).$$

Methods based on solving (4.3) have several benefits for sparse problems. The ability to treat linear constraints explicitly is helpful for the many large problems in which most of the constraints are linear. As noted in the Introduction, it is often a feature of sparse problems that the cost of evaluating the problem functions is dominated by the sparse matrix operations. The superiority of SQP methods (§ 4.3.2) for dense problems results from the generally lower number of function evaluations

compared to methods based on (4.3); for sparse problems, however, the function evaluations required to solve (4.3) may be insignificant compared to the savings that would result from solving fewer subproblems. If an active-set method of the type described in § 3.3.1 is applied to (4.3), only the *projected Hessian* needs to be stored (rather than the full Hessian). Thus, methods based on (4.3) will tend to be more effective than augmented Lagrangian methods for problems in which the Hessian of the Lagrangian function is not sparse and the projected Hessian can be stored as a dense matrix.

4.3. Methods based on linear and quadratic programming. We now consider two classes of methods in which the subproblems are solved *without evaluation of the problem functions* (in contrast to the methods of §§ 4.1 and 4.2).

4.3.1. Sequential linear programming methods. Because of the availability and high quality of software for sparse linear programs, a popular technique for solving large-scale problems has been to choose each iterate as the solution of an LP subproblem; we shall call these *sequential linear programming* (SLP) methods. They were first proposed by Griffith and Stewart (1961); for a recent survey, see Palacios-Gomez, Lasdon and Engquist (1982).

One crucial issue in an SLP method is the definition of the linear functions in the subproblem. A typical formulation is

$$\begin{aligned} &\underset{x\in\Re^n}{\text{minimize}}\ g_k^T(x-x_k)\\ &\text{subject to } A_k(x-x_k)=-c_k,\\ &\qquad l\leqq x\leqq u. \end{aligned}$$

With some formulations, the LP may not be well posed—for example, there may be fewer constraints than variables. The usual way of ensuring a correctly posed subproblem is to include additional constraints on the variables, such as bounds on the change in each variable. In general, the latter are also needed to ensure convergence.

SLP methods have the advantage that the subproblems can be solved using all the technology of sparse LP codes. They tend to be efficient on two types of problems: those with nearly linear functions, particularly slightly perturbed linear programs; and those in which the functions can be closely approximated by piecewise linear functions (e.g., the objective function is separable and convex). Unfortunately, on general problems SLP methods are at best linearly convergent unless the number of active constraints at the solution is equal to the number of variables. Furthermore, the speed of convergence critically depends on the technique that defines each subproblem.

Recently, some of the techniques used in SQP methods (§ 4.3.2) have been applied to the SLP approach—such as the use of a merit function to ensure progress after each outer iteration. Such techniques cannot be expected to improve the asymptotic rate of convergence of SLP methods, but they should improve robustness and overall effectiveness.

Beale (1978) has given a method that is designed to make extensive use of an existing LP system. The nonlinearly constrained problem is assumed to be of the form

$$\begin{aligned} &\underset{x,y}{\text{minimize}}\ c(x)^T y\\ &\text{subject to } A(x)y=b(x), \qquad (4.5)\\ &\qquad l\leqq x\leqq u,\\ &\qquad v\leqq y\leqq w. \end{aligned}$$

A special nonlinear algorithm is then used to adjust x; for each value of x, a new estimate y is determined by solving an LP.

4.3.2. Sequential quadratic programming methods. The most popular methods in recent years for dense nonlinearly constrained problems are based on solving a sequence of quadratic programming subproblems (see Powell (1982) for a survey). At iteration k, a typical QP subproblem has the form

$$\underset{p\in\Re^n}{\text{minimize}}\ \tfrac{1}{2}p^T H_k p + g_k^T p$$

$$\text{subject to } A_k p = -c_k$$

$$l - x_k \leqq p \leqq u - x_k,$$

where H_k is an approximation to the Hessian of the Lagrangian function. The solution of the QP subproblem is then used as the *search direction* p_k in (1.1). The step α_k is chosen to achieve a suitable reduction in some *merit function* that measures progress toward the solution. In the dense case, the most popular method is based on taking H_k as a positive-definite quasi-Newton approximation to the Hessian (Powell (1977)). However, the many options in defining the QP subproblem have yet to be fully understood and resolved (see Murray and Wright (1982), for a discussion of some of the critical issues).

Further complex issues are raised when applying an SQP method to sparse problems (see, e.g., Gill et al. (1981)). The general development of methods has been hampered because methods for sparse quadratic programming are only just being developed, and are not yet generally available for use within a general nonlinear algorithm. However, Escudero (1980) has reported some success with an SQP implementation in which a sparse quasi-Newton approximation is used for H_k (see also § 3.7.2).

REFERENCES

J. ABADIE AND J. CARPENTIER (1969), *Generalization of the Wolfe reduced-gradient method to the case of nonlinear constraints*, in Optimization, R. Fletcher, ed., Academic Press, London and New York, pp. 37–49.

O. AXELSSON (1974), *On preconditioning and convergence acceleration in sparse matrix problems*, Report 74-10, CERN European Organization for Nuclear Research, Geneva.

R. H. BARTELS (1971), *A stabilization of the simplex method*, Numer. Math., 16, pp. 414–434.

R. H. BARTELS AND G. H. GOLUB (1969), *The simplex method of linear programming using the LU decomposition*, Comm. ACM, 12, pp. 266–268.

R. H. BARTELS, G. H. GOLUB AND M. A. SAUNDERS (1970), *Numerical techniques in mathematical programming*, in Nonlinear Programming, J. B. Rosen, O. L. Mangasarian and K. Ritter, eds., Academic Press, London and New York, pp. 123–176.

E. M. L. BEALE (1967), *An introduction to Beale's method of quadratic programming*, in Nonlinear Programming, J. Abadie, ed., Academic Press, London and New York, pp. 143–153.

——— (1978), *Nonlinear programming using a general mathematical programming system*, in Design and Implementation of Optimization Software, H. J. Greenberg, ed., Sijthoff and Noordhoff, Netherlands, pp. 259–279.

J. BISSCHOP AND A. MEERAUS (1977), *Matrix augmentation and partitioning in the updating of the basis inverse*, Math. Prog., 13, pp. 241–254.

——— (1980), *Matrix augmentation and structure preservation in linearly constrained control problems*, Math. Prog., 18, pp. 7–15.

K. W. BRODLIE (1977), *Unconstrained optimization*, in The State of the Art in Numerical Analysis, D. Jacobs, ed., Academic Press, London and New York, pp. 229–268.

G. C. BROWN AND W. G. WRIGHT (1981), *Automatic identification of embedded structure in large-scale optimization models*, in Large-Scale Linear Programming, G. B. Dantzig, M. A. H. Dempster and M. J. Kallio, eds., IIASA, Laxenburg, Austria, pp. 781–808.

T. F. COLEMAN AND J. J. MORÉ (1982a), *Software for estimating sparse Jacobian matrices*, Report ANL-82-37, Argonne National Laboratory, Argonne, IL.

——— (1982b), *Estimation of sparse Hessian matrices and graph coloring problems*, Report 82-535, Dept. Computer Science, Cornell Univ., Ithaca, New York.

——— (1983), *Estimation of sparse Jacobian matrices and graph coloring problems*, SIAM J. Numer. Anal., 20, pp. 187–209.

P. CONCUS, G. H. GOLUB AND D. P. O'LEARY (1976), *A generalized conjugate-gradient method for the numerical solution of elliptic partial differential equations*, in Sparse Matrix Computations, J. R. Bunch and D. J. Rose, eds., Academic Press, London and New York, pp. 309–322.

R. W. COTTLE (1974), *Manifestations of the Schur complement*, Linear Algebra Appl., 8, pp. 189–211.

R. W. COTTLE AND A. DJANG (1979), *Algorithmic equivalence in quadratic programming*, J. Optim. Theory Appl., 28, pp. 275–301.

A. R. CURTIS, M. J. D. POWELL AND J. K. REID (1974), *On the estimation of sparse Jacobian matrices*, J. Inst. Maths. Applics., 13, pp. 117–119.

G. B. DANTZIG (1963), *Linear Programming and Extensions*, Princeton Univ. Press, Princeton, NJ.

G. B. DANTZIG, M. A. H. DEMPSTER AND M. J. KALLIO, eds., (1981), *Large-Scale Linear Programming (Vol. 1)*, IIASA Collaborative Proceedings Series, CP-81-51, IIASA, Laxenburg, Austria.

G. B. DANTZIG AND W. ORCHARD-HAYS (1954), *The product form of the inverse in the simplex method*, Math. Comp., 8, pp. 64–67.

G. B. DANTZIG AND P. WOLFE (1960), *The decomposition principle for linear programs*, Oper. Res., 8, pp. 110–111.

R. S. DEMBO, S. C. EISENSTAT AND T. STEIHAUG (1982), *Inexact Newton methods*, SIAM J. Numer. Anal., 19, pp. 400–408.

R. S. DEMBO AND T. STEIHAUG (1980), *Truncated-Newton algorithms for large-scale unconstrained optimization*, Working Paper # 48, School of Organization and Management, Yale Univ., New Haven, CT.

J. E. DENNIS, JR. AND E. S. MARWIL (1982), *Direct secant updates of matrix factorizations*, Math. Comp., 38, pp. 459–474.

J. E. DENNIS, JR. AND J. J. MORÉ (1977), *Quasi-Newton methods, motivation and theory*, SIAM Rev., 19, pp. 46–89.

J. E. DENNIS, JR. AND R. B. SCHNABEL (1979), *Least change secant updates for quasi-Newton methods*, SIAM Rev., 21, pp. 443–469.

I. S. DUFF (1982), *A survey of sparse matrix software*, Report AERE-R.10512, Atomic Energy Research Establishment, Harwell, England; in Sources and Development of Mathematical Software, W. R. Cowell, ed., Prentice-Hall, Englewood Cliffs, NJ, 1984.

I. S. DUFF AND J. K. REID (1982), *The multifrontal solution of indefinite sparse symmetric linear systems*, Report CSS 122, Atomic Energy Research Establishment, Harwell, England; ACM Trans. Math. Software, 9 (1983), pp. 302–305.

——— (1983), *Direct methods for solving sparse systems of linear equations*, presented at the Sparse Matrix Symposium, Fairfield Glade, Tennessee, 1982; I. S. DUFF, this Journal, this issue, pp. 605–619.

L. F. ESCUDERO (1980), *A projected Lagrangian method for nonlinear programming*, Report G320-3407, IBM Scientific Center, Palo Alto, CA.

A. V. FIACCO AND G. P. MCCORMICK (1968), *Nonlinear Programming: Sequential Unconstrained Minimization Techniques*, John Wiley, New York and Toronto.

R. FLETCHER (1974), *Methods related to Lagrangian functions*, in Numerical Methods for Constrained Optimization, P. E. Gill and W. Murray, eds., Academic Press, London and New York, pp. 219–240.

——— (1980), *Practical Methods of Optimization, Volume 1, Unconstrained Optimization*, John Wiley, New York and Toronto.

——— (1981), *Practical Methods of Optimization, Volume 2, Constrained Optimization*, John Wiley, New York and Toronto.

R. FLETCHER AND C. M. REEVES (1964), *Function minimization by conjugate gradients*, Comput. J., 7, pp. 149–154.

J. J. H. FORREST AND J. A. TOMLIN (1972), *Updating triangular factors of the basis to maintain sparsity in the product form simplex method*, Math. Prog., 2, pp. 263–278.

R. FOURER (1982), *Solving staircase linear programs by the simplex method, 1: Inversion*, Math. Prog., 23, pp. 274–313.

J. A. GEORGE AND M. T. HEATH (1980), *Solution of sparse linear least squares problems using Givens rotations*, Linear Algebra Appl., 34, pp. 69–83.

J. A. GEORGE AND E. NG (1982), *Solution of sparse underdetermined systems of linear equations*, Report CS-82-39, Dept. Computer Science, Univ. Waterloo, Waterloo, Ontario, Canada.

P. E. GILL, G. H. GOLUB, W. MURRAY AND M. A. SAUNDERS (1974), *Methods for modifying matrix factorizations*, Math. Comp., 28, pp. 505–535.

P. E. GILL AND W. MURRAY (1974), *Newton-type methods for unconstrained and linearly constrained optimization*, Math. Prog., 28, pp. 311–350.

——— (1979), *Conjugate-gradient methods for large-scale nonlinear optimization*, Report SOL 79-15, Dept. Oper. Res., Stanford Univ., Stanford, CA.

P. E. GILL, N. I. M. GOULD, W. MURRAY, M. A. SAUNDERS AND M. H. WRIGHT (1982), *Range-space methods for convex quadratic programming*, Report SOL 82-14, Department of Operations Research, Stanford Univ., Stanford, CA.

P. E. GILL, W. MURRAY, M. A. SAUNDERS AND M. H. WRIGHT (1981), *QP-based methods for large-scale nonlinearly constrained optimization*, in Nonlinear Programming 4, O. L. Mangasarian, R. R. Meyer and S. M. Robinson, eds., Academic Press, London and New York, pp. 57–98.

P. E. GILL, W. MURRAY AND M. H. WRIGHT (1981), *Practical Optimization*, Academic Press, London and New York.

P. GILLE AND E. LOUTE (1981), *A basis factorization and updating technique for staircase structured systems of linear equations*, Discussion Paper 8113, CORE, Université Catholique de Louvain, Louvain-la-Neuve, Belgium.

——— (1982), *Updating the LU Gaussian decomposition for rank-one corrections; application to linear programming basis partitioning techniques*, Cahier No. 8201, Séminaire de Mathématiques Appliquées aux Sciences Humaines, Facultés Universitaires Saint-Louis, Brussels, Belgium.

R. E. GRIFFITH AND R. A. STEWART (1961). *A nonlinear programming technique for the optimization of continuous processing systems*, Management Science, 7, pp. 379–392.

F. G. GUSTAVSON (1972), *Some basic techniques for solving sparse systems of linear equations*, in Sparse Matrices and their Applications, D. J. Rose and R. A. Willoughby, eds., Plenum Press, New York, pp. 41–52.

M. T. HEATH (1982), *Some extensions of an algorithm for sparse linear least squares problems*, this Journal, 3, pp. 223–237.

E. HELLERMAN AND D. RARICK (1971), *Reinversion with the preassigned pivot procedure*, Math. Prog., 1, pp. 195–216.

——— (1972), *The partitioned preassigned pivot procedure* (P^4), in Sparse Matrices and their Applications, D. J. Rose and R. A. Willoughby, eds., Plenum Press, New York, pp. 67–76.

M. R. HESTENES (1969), *Multiplier and gradient methods*, J. Optim. Theory Appl., 4, pp. 303–320.

——— (1980), *Conjugate Direction Methods in Optimization*, Springer-Verlag, New York.

M. R. HESTENES AND E. STIEFEL (1952), *Methods of conjugate gradients for solving linear systems*, J. Res. Nat. Bur. Standards, 49, pp. 409–436.

G. KRON (1956), *Diakoptics*, MacDonald, London.

H. M. MARKOWITZ (1957), *The elimination form of the inverse and its applications to linear programming*, Management Sci., 3, pp. 255–269.

E. S. MARWIL (1978), *Exploiting sparsity in Newton-type methods*, Ph.D. Thesis, Cornell Univ., Ithaca, New York.

S. T. MCCORMICK (1983), *Optimal approximation of sparse Hessians and its equivalence to a graph coloring problem*, Math. Prog., 26, pp. 153–171.

J. J. MORÉ AND D. C. SORENSEN (1982), *Newton's method*, Report ANL-82-8, Argonne National Laboratory, Argonne, IL.

W. MURRAY AND M. H. WRIGHT (1982), *Computation of the search direction in constrained optimization algorithms*, Math. Prog. Study, 16, pp. 62–83.

B. A. MURTAGH AND M. A. SAUNDERS (1977), MINOS User's Guide, Report SOL 77-9, Dept. Operations Research, Stanford University, Stanford, CA.

——— (1978), *Large-scale linearly constrained optimization*, Math. Prog., 14, pp. 41–72.

——— (1980), MINOS/AUGMENTED *User's Manual*, Report SOL 80-14, Dept. Operations Research, Stanford University, Stanford, CA.

——— (1982), *A projected Lagrangian algorithm and its implementation for sparse nonlinear constraints*, Math. Prog. Study, 16, pp. 84–118.

S. G. NASH (1982), *Truncated-Newton methods*, Ph.D. Thesis, Computer Science Department, Stanford Univ., Stanford, CA.

D. P. O'LEARY (1982), *A discrete Newton algorithm for minimizing a function of many variables*, Math. Prog., 23, pp. 20–33.

C. C. PAIGE AND M. A. SAUNDERS (1975), *Solutions of sparse indefinite systems of linear equations*, SIAM J. Numer. Anal., 12, pp. 617–629.

F. PALACIOS-GOMEZ, L. S. LASDON AND M. ENGQUIST (1982), *Nonlinear optimization by successive linear programming*, Management Sci., 28, 10, pp. 1106–1120.

M. J. D. POWELL (1969), *A method for nonlinear constraints in optimization problems*, in Optimization, R. Fletcher, ed., Academic Press, London and New York, pp. 283–297.

M. J. D. POWELL (1976), *A view of unconstrained optimization*, in Optimization in Action, L. C. W. Dixon, ed., Academic Press, London and New York, pp. 117–152.

——— (1981), *A note on quasi-Newton formulae for sparse second derivative matrices*, Math. Prog., 20, pp. 144–151.

——— (1982), *State-of-the-Art Tutorial on "Variable metric methods for constrained optimization"*, Report DAMTP 1982/NA5, Dept. Applied Mathematics and Theoretical Physics, University of Cambridge, England.

M. J. D. POWELL AND P. L. TOINT (1979), *On the estimation of sparse Hessian matrices*, SIAM J. Numer. Anal., 16, pp. 1060–1074.

J. K. REID (1971), *On the method of conjugate gradients for the solution of large sparse systems of linear equations*, in Large Sparse Sets of Linear Equations, J. K. Reid, ed., Academic Press, London and New York, pp. 231–254.

——— (1976), *Fortran subroutines for handling sparse linear programming bases*, Report AERE-R8269, Atomic Energy Research Establishment, Harwell, England.

——— (1982), *A sparsity-exploiting variant of the Bartels–Golub decomposition for linear programming bases*, Math. Prog., 24, pp. 55–69.

S. M. ROBINSON (1972), *A quadratically convergent algorithm for general nonlinear programming problems*, Math. Prog., 3, pp. 145–156.

J. B. ROSEN (1978), *Two-phase algorithm for nonlinear constraint problems*, in Nonlinear Programming 3, O. L. Mangasarian, R. R. Meyer and S. M. Robinson, eds., Academic Press, London and New York, pp. 97–124.

J. B. ROSEN AND J. KREUSER (1972). *A gradient projection algorithm for nonlinear constraints*, in Numerical Methods for Non-Linear Optimization, F. A. Lootsma, ed., Academic Press, London and New York, pp. 297–300.

M. A. SAUNDERS (1976), *A fast, stable implementation of the simplex method using Bartels-Golub updating*, in Sparse Matrix Computations, J. R. Bunch and D. J. Rose, eds., Academic Press, New York, pp. 213–226.

L. K. SCHUBERT (1970), *Modification of a quasi-Newton method for nonlinear equations with a sparse Jacobian*, Math. Comp., 24, pp. 27–30.

D. F. SHANNO (1980), *On variable metric methods for sparse Hessians*, Math. Comp., 34, pp. 499–514.

D. C. SORENSEN (1982), *Collinear scaling and sequential estimation in sparse optimization algorithms*, Math. Prog. Study, 18, pp. 135–159.

T. STEIHAUG (1980), *Quasi-Newton methods for large-scale nonlinear problems*, Working Paper # 49, School of Organization and Management, Yale Univ., New Haven, CT.

——— (1982), *On the sparse and symmetric least-squares secant update*, Report MASC TR 82-4, Dept. Mathematical Sciences, Rice University, Houston, TX.

B. STOTT, O. ALSAC AND J. L. MARINHO (1980), *The optimal power flow problem*, in Electric Power Problems: The Mathematical Challenge, A. M. Erisman, K. W. Neves and M. H. Dwarakanath, eds., Society for Industrial and Applied Mathematics, Philadelphia, pp. 327–351.

M. N. THAPA (1980), *Optimization of unconstrained functions with sparse Hessian matrices*, Ph.D. Thesis, Stanford Univ., Stanford, California.

P. L. TOINT (1977), *On sparse and symmetric matrix updating subject to a linear equation*, Math. Comp., 31, pp. 954–961.

——— (1978), *Some numerical results using a sparse matrix updating formula in unconstrained optimization*, Math. Comp., 32, pp. 839–851.

——— (1979), *On the superlinear convergence of an algorithm for solving a sparse minimization problem*, SIAM J. Numer. Anal., 16, pp. 1036–1045.

J. A. TOMLIN (1975), *An accuracy test for updating triangular factors*, Math. Prog. Study 4, pp. 142–145.

J. H. WILKINSON (1965), *The Algebraic Eigenvalue Problem*, Oxford Univ. Press, London.