Mathematical Programming 45 (1989) 373-406
North-Holland

A DIRECT ACTIVE SET ALGORITHM FOR LARGE SPARSE QUADRATIC PROGRAMS WITH SIMPLE BOUNDS

Thomas F. COLEMAN and Laurie A. HULBERT

Computer Science Department & Center for Applied Mathematics, Cornell University, Ithaca, NY 14853, USA

We show how a direct active set method for solving definite and indefinite quadratic programs with simple bounds can be efficiently implemented for large sparse problems. All of the necessary factorizations can be carried out in a static data structure that is set up before the numeric computation begins. The space required for these factorizations is no larger than that required for a single sparse Cholesky factorization of the Hessian of the quadratic. We propose several improvements to this basic algorithm: a new way to find a search direction in the indefinite case that allows us to free more than one variable at a time and a new heuristic method for finding a starting point. These ideas are motivated by the two-norm trust region problem. Additionally, we also show how projection techniques can be used to add several constraints to the active set at each iteration. Our experimental results show that an algorithm with these improvements runs much faster than the basic algorithm for positive definite problems and finds local minima with lower function values for indefinite problems.

AMS/MOS Subject Classifications: 65K05, 90C20, 65K10, 65F30.

Key words: Quadratic programming, large sparse minimization, active set methods, trust region methods, sparse Cholesky factorization updates, simple bounds, box constraints.

1. Introduction

1.1. The problem

In this paper we are interested in minimizing a quadratic subject to simple bounds

$$\min \tfrac{1}{2}x^{\mathrm{T}}Ax + b^{\mathrm{T}}x, \quad l \leqslant x \leqslant u, \tag{1}$$

where A is an $n \times n$ symmetric matrix that is not necessarily positive definite. In particular, we are interested in the case where A is large and sparse. For positive definite A, this problem can be solved in polynomial time by an interior point algorithm [26]; for indefinite A, this problem is NP-hard [22]. In this paper we design algorithms that work well in practice; however, we do not claim that they are polynomial. Our algorithms work for both indefinite and positive definite quadratics, but in the indefinite case, they find only local minima.

Research partially supported by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under grant DE-FG02-86ER25013.A000.

We show how a direct active set method for solving definite and indefinite quadratic programs with simple bounds, typical of the dense algorithms of Fletcher and Jackson [12] and Gill and Murray [15], can be efficiently implemented for large sparse problems. All of the necessary factorizations can be carried out in a static data structure that is set up before the numeric computation begins. The space required for these factorizations is no larger than that required for a single sparse Cholesky factorization of A. We propose several improvements to this basic algorithm: a new way to find a search direction in the indefinite case that allows us to free more than one variable at a time and a new heuristic method for finding a starting point. These ideas are motivated by the two-norm trust region problem. Additionally, we also show how projection, or 'bending', techniques can be used to add several constraints to the active set at each iteration. Our experimental results show that an algorithm with these improvements runs much faster than the basic algorithm for positive definite problems and finds local minima with lower function values for indefinite problems.

We use a direct method to solve our linear systems, namely, Cholesky factorization. This method is numerically stable [17] and there exist fast efficient implementations for large sparse matrices, e.g. Sparspak [13]. We can efficiently update the Cholesky factorizations when we add or drop a row and column from the matrix and use the Cholesky factorization to compute a search direction when the matrix becomes indefinite. Using a direct method requires that we store L, the Cholesky factor of A. By preordering A, we can attempt to reduce the number of nonzeros in L. There exist heuristics for this, such as the minimum degree and nested disection algorithms, that work well in practice [13]. Some of the algorithms that we describe do not require the matrix factorizations to be updated. For these algorithms, using an iterative method to solve the linear system might be competitive.

Many authors have developed active set methods for solving (1). Our basic algorithm is based on the dense algorithms of Fletcher and Jackson [12] for solving (1) and of Gill and Murray [15] for solving the more general problem

$$\min \tfrac{1}{2}x^{\mathrm{T}}Ax + b^{\mathrm{T}}x, \quad Cx \leq d.$$

Calamai and Moré [4] and Moré [19] outlined an active set method that uses projected gradients and can be generalized to the sparse case. They did not fully specify the algorithm; in particular, they did not specify a choice of starting point or search direction for the indefinite case, or details of a sparse implementation. More recently, Moré and Toraldo [21] provided implementation details of an active set algorithm that uses both projected gradient steps and Newton steps. Their algorithm could be implemented for the sparse case using the technology we develop here. Their numerical results for the dense case indicate that their algorithm might be competitive with ours for the sparse case. To solve (1) when A is sparse, Dembo and Tulowitzki [8] proposed several active set methods with inexact subspace minimization, using both projected gradient steps and conjugate gradients steps.

We use exact subspace minimization and solve our linear systems directly. Björck [3] proposed an active set algorithm for the sparse least squares problem

$$\min \tfrac{1}{2}x^{\mathrm{T}}C^{\mathrm{T}}Cx + d^{\mathrm{T}}x, \quad l \leq x \leq u,$$

that is similar to our algorithm of Section 2. His algorithm only considers the case where the matrix $A = C^{\mathrm{T}}C$ is positive definite but it does not form A; the algorithms we consider also allow A to be indefinite. Conn, Gould and Toint [6] use trust region steps in an active set algorithm to solve the general problem

$$\min f(x), \quad l \leq x \leq u.$$

They show that by projecting a scaled gradient direction, the correct active set can be identified in finitely many steps.

The organization of our paper is as follows. The rest of this section introduces notation and provides some background material. Section 2 describes a straightforward active set algorithm and provides a proof of its convergence. Section 3 provides the details of a sparse implementation of this algorithm. Section 4 describes several techniques that can be used to improve the performance of our basic algorithm. Section 5 describes our computer implementation, suggests several algorithms that use the improvements outlined in Section 4, and presents numerical results to compare these algorithms. Section 6 presents our conclusions and ideas for future work.

1.2. *Some notation and background material*

We let c_{ij} denote the ijth entry of a matrix C, c_{*i} the ith column of C, and c_{i*}^{T} the ith row of C. We let $c_{\wedge i}$ denote the top part of the ith column of C, where top part should be obvious from context. Similarly, we let $c_{\vee i}$ denote the bottom part of the ith column of C. The expression $C > 0$ means that C is positive definite; similarly $C \geq 0$ means that C is positive semi-definite. We let I denote the identity matrix, and e_i the ith column of I. If $S = \{s_1, \ldots, s_k\}$ is a subset of the first n integers that is maintained as an ordered list, we define Z_S as the matrix of columns $(e_{s_1}, \ldots, e_{s_k})$. We let $|S|$ denote the size of S.

To simplify our presentation, we note that through a suitable scaling and translation we can rewrite (1) in a simpler form as

$$\min \tfrac{1}{2}x^{\mathrm{T}}Ax + b^{\mathrm{T}}x, \quad -1 \leq x \leq 1. \tag{2}$$

Henceforth we assume the problem is in this form. Note that this transformation may adversely affect the conditioning of A, especially if some of the original bounds are large.

The variables with values of 1 or -1 we call variables *at their bound.* In our active set algorithm, the set B of variables at their bound that we are holding fixed we call *bound* variables. These variables correspond to the *active* constraints, those satisfied with equality that we are forcing to remain satisfied. When we *bind* a

variable we add it to the set of bound variables. The set F of variables which are not bound variables we call *free* variables, i.e., $F = \{1, \ldots, n\} - B$. Note that F can contain variables that are at their bounds. We refer to F as the free set. When we free a variable, we add it to the set of free variables.

We let $g(x)$ denote the gradient at the current point, $g(x) = Ax + b$, or simply g if x is apparent from context. We define the reduced quantities

$$A_F = Z_F^T A Z_F, \quad g_F = Z_F^T g, \quad g_B = Z_B^T g, \quad x_F = Z_F^T x, \quad x_B = Z_B^T x.$$

Note that because of the structure of Z_F and Z_B, the reduced quantities are just submatrices and subvectors of the original quantities. To simplify our discussion, if $F = \emptyset$, although A_F is not really defined, we consider A_F to be positive definite. We call x a *constrained stationary point* if $g_F(x) = 0$. We let $q(x) = \frac{1}{2}x^T Ax + b^T x$ and $q_F(s) = q(x + Z_F s) - q(x) = \frac{1}{2}s^T A_F s + g_F^T s$. Thus the quadratic $q_F(s)$ defined at the point x is a measure of the change in q from the point x to the point $x + Z_F s$.

Let x^* be the current point, let B^* be the set of all variables at a bound and let F^* be the set of all variables not at a bound. Then the following conditions are necessary for x^* to be a local minimum:

$$\begin{aligned} &\text{feasibility:} && -1 \leq x^* \leq 1, \\ &\text{first order:} && g(x^*)_i = 0 \quad \text{if } -1 < x_i^* < 1, \\ &&& g(x^*)_i \leq 0 \quad \text{if } x_i^* = 1, \\ &&& g(x^*)_i \geq 0 \quad \text{if } x_i^* = -1, \\ &\text{second order:} && A_{F^*} \geq 0. \end{aligned} \tag{3}$$

These are not sufficient conditions however. If we assume that x^* is *non-degenerate*, i.e., $g(x^*)_i \neq 0$ for all $i \in B^*$, then the following conditions are sufficient to guarantee that x^* is a local minimum:

$$\begin{aligned} &\text{feasibility:} && -1 \leq x^* \leq 1, \\ &\text{first order:} && g(x^*)_i = 0 \quad \text{if } -1 < x_i^* < 1, \\ &&& g(x^*)_i < 0 \quad \text{if } x_i^* = 1, \\ &&& g(x^*)_i > 0 \quad \text{if } x_i^* = -1, \\ &\text{second order:} && A_{F^*} > 0. \end{aligned} \tag{4}$$

The algorithm we present converges to a point that satisfies (3) with $A_{F^*} > 0$. If x^* is non-degenerate, then we are guaranteed that x^* is a local minimum, since (4) is satisfied. Fletcher [11] showed that if x^* is degenerate then x^* is a local minimum of a nearby problem, one with a slightly perturbed b. Murty and Kabadi [22] showed that if x^* is degenerate then the problem of determining whether or not x^* is a local minimum is NP-hard.

Notice that if x^* solves (2) then it also solves the equality constrained problem

$$\min \tfrac{1}{2}x^T Ax + b^T x, \quad x_{B^*} = x_{B^*}^*.$$

So if we knew the correct active set, we would need only to solve an unconstrained problem. This observation forms the basis for the active set methods. At each iteration, these methods hold a subset of the variables fixed at a bound, and they attempt to minimize $q(x)$ in the space of the remaining variables, subject to the bounds on these variables. They continue adding variables to the active set until they minimize $q(x)$ with some active set. If the necessary conditions (3) show that this point is not a local minimum of (2) they remove some of the active constraints and repeat.

2. A basic active set algorithm

2.1. The algorithm

In this section, we describe an active set algorithm GSA, based on the dense algorithms by Fletcher and Jackson [12] and Gill and Murray [15], and outlined below. The details of the sparse implementation are presented in the next section.

The first step of the algorithm is to find a permutation matrix P and update A via $A = PAP^{\mathrm{T}}$ and b via $b = Pb$. Choose P to reduce the number of nonzeros in the Cholesky factor of a positive definite matrix with the sparsity structure of A. Finding an optimal P is an NP-complete problem [25], so we must rely on heuristics for this step such as the minimum degree or the nested dissection algorithms [13]. For simplicity, we will call this new ordering of A the original ordering.

The next step of the algorithm is to find a free set F on which A_F is positive definite. Although we refer to F as a set, we maintain F as an ordered list so that A_F is well-defined. Initialize $F := \{1, \ldots, n\}$ and begin the Cholesky factorization of the matrix A_F. If, at the ith step of Cholesky, a positive element occurs on the diagonal then set $x_i := 0$ and continue the factorization. If a nonpositive element occurs, remove i from F, exclude row and column i from the Cholesky factorization, and set x_i equal to whichever bound yields the smallest value when substituted into the one dimensional quadratic $\frac{1}{2}a_{ii}x_i^2 + b_i x_i$. At the end of the Cholesky factorization, x is the starting point, F is the desired free set (possibly the empty set), and we have the Cholesky factorization of A_F. Note that in the case where $A = \mathrm{diag}(a_{ii})$ and $a_{ii} \leq 0$, x is the global solution to (2).

The next step is to find a constrained stationary point. If x is not already a constrained stationary point, perform the following sequence of steps. Compute the Newton direction s by solving $A_F s = -g_F$, so s is the global minimum of q_F. Update x via the formula $x := x + \alpha Z_F s$, where $0 \leq \alpha \leq 1$ is chosen as large as possible without violating a constraint. If the Newton step is feasible, i.e., $\alpha = 1$, then x is a constrained stationary point. Otherwise remove from F any variables that have reached a bound and update the Cholesky factorization of A_F, as described in Section 3.2. If x is not a constrained stationary point, repeat the preceding sequence of steps until such a point is reached.

A General Sparse Active set algorithm (GSA).
Find a permutation matrix P and reorder the matrix A via $A := PAP^T$ and the vector b via $b := Pb$.
Using a greedy approach during the computation of the Cholesky factorization of A, find a starting point and a free set F such that A_F is positive definite.
{Find an initial constrained stationary point.}
while not at a constrained stationary point **do**
  compute Newton step.
  if Newton step is feasible take it.
  otherwise follow Newton direction until a constraint becomes active;
    bind corresponding variable and update factorization.
enddo
{Find an optimal point.}
while constrained stationary point is not optimal **do**
  free one of the bound variables corresponding to a component of the gradient that has the incorrect sign according to (3);
  update factorization.
  repeat
    if A_F is positive definite **then**
      compute Newton step.
      if Newton step is feasible take it.
      otherwise follow Newton direction until a constraint becomes active;
        bind corresponding variable and update factorization.
    else
      compute nonascending search direction of negative or zero curvature.
      follow search direction until a constraint becomes active;
        bind corresponding variable and update factorization.
    endif
  until at constrained stationary point where $A_F > 0$.
  if necessary, reorder F and update factorization.
endo
Reorder x via $x := P^T x$.

Now determine if the constrained stationary point x is indeed an optimal point. If all of the components of $g_B(x)$ have the correct sign according to (3), then x is an optimal point. Otherwise, find the bound variable, say t, corresponding to the component of g_B that is largest in magnitude and has the incorrect sign. Update F by adding t to the end of it. Let F' be the value of F before it is updated. Solve $L_{F'}r = Z_{F'}a_{*t}$ for r and set $\tau := a_{tt} - r^T r$. If $\tau > 0$ then A_F is positive definite and its Cholesky factorization is

$$L_F = \begin{pmatrix} L_{F'} & \\ r^T & \sqrt{\tau} \end{pmatrix}.$$

If $\tau \leq 0$ then A_F is not positive definite, so its Cholesky factorization is not defined, but $A_F = L_F D L_F^T$ where

$$L_F = \begin{pmatrix} L_{F'} & \\ r^T & 1 \end{pmatrix}$$

and $D = \text{diag}(1, \ldots, 1, \tau)$.

We now distinguish two cases.

If A_F is positive definite, compute the Newton direction s by solving $A_F s = -g_F$. Set $x := x + \alpha Z_F s$, where $0 \leq \alpha \leq 1$ is chosen as large as possible without violating a constraint. If the Newton step is feasible, i.e., $\alpha = 1$, x is a constrained stationary point and A_F remains positive definite. Otherwise remove from F any variable that has reached a bound and update the Cholesky factorization of A_F.

If A_F is not positive definite, find a nonascending search direction as follows. Solve $L_F^T \tilde{s} = e_m$, where $m = |F|$ and set s to be $\tilde{s}$ or $-\tilde{s}$ so that $g_F^T s \leq 0$. Note that s is a direction of nonpositive curvature since $s^T A_F s = \tau \leq 0$. Set $x := x + \alpha Z_F s$, where $\alpha \geq 0$ is chosen as large as possible without violating a constraint. Remove from F any variable that has reached a bound and update the LDL^T factorization of A_F.

If x is not a constrained stationary point or A_F is not positive definite, repeat the preceding sequence of steps.

Now x is a constrained stationary point and A_F is positive definite. If $t \in F$ reorder the elements of F so that they are in their original order and update the Cholesky factorization of A_F so that it corresponds with this order, as described in Section 3.3. If x is not optimal proceed as above. If x is optimal, reorder x via $x := P^T x$.

2.2. A proof of convergence

In this section we prove that GSA converges. The proofs refer to the algorithm GSA as outlined and described in detail in Section 2.1. They use ideas contained in the convergence analysis of active set methods by Fletcher [11] and Gill and Murray [15]. Recall that we are assuming that $A_F > 0$ when F is empty.

Lemma 2.1. *Execution of the first while loop and each repeat loop in the second while loop must terminate in a finite number of steps.*

Proof. This is immediate, since there are only $|F|$ variables that can be bound and for each search direction, we either bind a variable or terminate the loop by stepping to a constrained stationary point. □

Lemma 2.2. *The function value of each constrained stationary point is lower than that of the previous one.*

Proof. Let $\hat{x}$ be a constrained stationary point and let $\hat{F}$ be the value of the free set F at $\hat{x}$. So $g_{\hat{F}} = 0$. Consider the second while loop in GSA. We free a variable, say t. Then after updating F we have $g_F = \beta e_m$, where $m = |F|$ and $\beta \neq 0$. Suppose we take a step of length 0. This implies that we bind a variable. The variable that we bind cannot be the variable t, since the fact that $-g_F = -\beta e_m$ is a feasible nonascending direction and that $g_F^T s \leq 0$ imply that $s_m e_m$ is a feasible nonascending direction. After binding the variable and updating F, we still have $g_F = \beta e_m$, where $m = |F|$. Thus we have not reached a constrained stationary point so the loop will not terminate. We can take steps of length 0 at most r times, where r is the number of variables at their bound in $\hat{F}$. Then we must take a step of positive length, so $\alpha > 0$. We now show that this step decreases the function value. We have two cases, depending on how we generate s.

Case 1. A_F is positive definite. Then s solves $A_F s = -g_F(x)$ and $\alpha \leq 1$. So

$$q(x + \alpha Z_F s) - q(x) = (\tfrac{1}{2}\alpha^2 - \alpha)(g_F^T A_F^{-1} g_F) < 0.$$

Case 2. A_F is not positive definite. We compute s by solving $L_F^T s = \pm e_m$. Since L_F^T is upper triangular and nonsingular, $s_m \neq 0$. By our choice of the sign of s, $g_F^T s = \beta e_m^T s = \beta s_m < 0$. So

$$q(x + \alpha Z_F s) - q(x) = \alpha g_F^T s + \tfrac{1}{2}\alpha^2 s^T A_F s < 0.$$

Therefore we have decreased the function value. In each subsequent step we are never increasing the function value since either we are following the Newton direction, i.e., we are stepping toward the global minimum of the quadratic q_F, or we are following a nonascending direction of nonpositive curvature. Thus the next constrained stationary point has a strictly lower function value than the previous constrained stationary points. □

Theorem 2.3. *The algorithm* GSA *converges to a point satisfying* (3) *with* $A_F > 0$ *in a finite number of steps.*

Proof. Finding an initial free set is obviously a finite procedure. By Lemma 2.1, execution of the first while loop and each repeat loop in the second while loop terminates in a finite number of steps. Each constrained stationary point is a global, although not necessarily unique, solution to an equality constrained problem. By Lemma 2.2 the function value of each constrained stationary point is lower than the values of all previously found points. Since there are only finitely many equality constrained problems, we cannot cycle and must eventually terminate. When we terminate, $g_F = 0$, $A_F > 0$, and the signs of the components of g_B satisfy (3), so the algorithm GSA converges to a point with the desired properties in a finite number of steps. □

3. A sparse implementation

3.1. The data structure

A major portion of the work in the GSA algorithm involves computing and updating the Cholesky factorization. The following three steps are typically employed in computing the Cholesky factorization of a sparse matrix [13].

(1) Find a permutation matrix P and form PAP^T. Choose P to reduce the number of nonzeros in L.
(2) Symbolically factor A to determine the positions of the nonzeros in L. Use this information to allocate storage for L.
(3) Numerically factor A.

In this section we show that we can perform the first two steps for the matrix A and use the resulting data structure to store each subsequent L_F.

In designing a data structure for sparse matrices, we want to store only the nonzeros with their row and column indices in a way that allows us to access them easily. Typically, one stores the nonzeros consecutively in an array by columns (rows), with a pointer to the beginning of each column (row) and a corresponding array of row (column) indices. We need to show that such a structure set up for L contains the necessary row and column indices to store L_F. To do this, instead of labeling the rows and columns of the matrices A_F and L_F with consecutive numbers, we label them according to the row and column of A from which they came. For example, if $F = \{2, 4, 6, 9\}$, then the rows of A_F are labeled 2, 4, 6, 9. This convention will be very important in updating the Cholesky factorization.

To exploit the sparsity of a matrix, it is often helpful to view the matrix as a graph. To this end, we provide some useful definitions from graph theory. See George and Liu [13] for a more thorough discussion of this material. We define a *graph* $G = (V, E)$ as a set of vertices V and a set of edges E. An *edge* (v, w) is an unordered pair of distinct vertices. A *path* from v to w in G is a sequence of distinct vertices $v = v_0, v_1, \ldots, v_k = w$ such that $(v_i, v_{i+1}) \in E$ for $i = 0, 1, \ldots, k-1$. We can represent the nonzero structure of an $n \times n$ symmetric matrix A with a graph $G(A) = (V, E)$ where $V = \{v_1, v_2, \ldots, v_n\}$ are the labels of the rows of A and $(v_i, v_j) \in E$ if and only if $a_{ij} \neq 0$ and $i \neq j$.

We can predict the nonzero structure of the Cholesky factor L of a symmetric matrix A using the graph $G(A)$. Assuming no exact numerical cancellation, Rose, Tarjan and Lueker [24] proved the following lemma.

Path Lemma. *Let i and j be integers with $i \geq j$. Then there exists a nonzero in position (i, j) of L if and only if there exists a path in $G(A)$ from i to j through vertices numbered lower than j.* □

We let $\eta(L)$ denote the set of nonzero positions in L that are predicted by the Path Lemma. We call these positions the *structural* nonzeros. We define

$$\eta(l_{*j}) = \{i \mid (i, j) \in \eta(L)\}$$

and

$$\eta_k(l_{*j}) = \{i \mid i \geq k \text{ and } (i,j) \in \eta(L)\},$$

i.e., $\eta(l_{*j})$ is the set of row indices of nonzeros in the jth column of L, and $\eta_k(l_{*j})$ is the set of row indices greater than or equal to k of nonzeros in the jth column of L. For a general vector v, we let $\eta(v)$ denote the set of nonzero positions of v.

We can now prove the first theorem, which says that if entry (i,j) of L_F is nonzero, then entry (i,j) of L is nonzero. This result follows easily from the Path Lemma and has been observed independently by Björck [3].

Theorem 3.1. *If* $(i,j) \in \eta(L_F)$ *then* $(i,j) \in \eta(L)$.

Proof. If $(i,j) \in \eta(L_F)$, then there exists a path in $G(A_F)$ from i to j through vertices numbered lower than j. But since $\eta(A_F) \subset \eta(A)$, there must also exist a path in $G(A)$ from i to j through vertices numbered lower than j. So $(i,j) \in \eta(L)$. □

The next subsections describe the data structure manipulations necessary for the computations we will be performing. Henceforth we will assume that the nonzeros are stored consecutively in an array by columns, with a pointer to the beginning of each column and a corresponding array of row indices. By Theorem 3.1, this data structure is adequate to store L_F for any F.

3.2. Binding a variable

In this section we show how to update the Cholesky factorization when we delete a variable from the free set. Suppose F' is the current free set and F is the free set obtained from F' by deleting the variable t. Let $F' = \{f_1, f_2, \ldots, f_s, \ldots, f_m\}$, where $m = |F'|$ and $f_s = t$. Then the matrix A_F is obtained from $A_{F'}$ by deleting row and column t of $A_{F'}$. We want to see how to obtain L_F from $L_{F'}$. These ideas were also developed by Björck [3].

Let us partition $A_{F'}$ and $L_{F'}$ along row and column t as follows:

$$A_{F'} = \begin{pmatrix} A_{11} & a_{\wedge t} & A_{12} \\ a_{\wedge t}^{\mathrm{T}} & a_{tt} & a_{\vee t}^{\mathrm{T}} \\ A_{21} & a_{\vee t} & A_{22} \end{pmatrix}, \qquad L_{F'} = \begin{pmatrix} L_{11} & & \\ l_{\wedge t}^{\mathrm{T}} & l_{tt} & \\ L_{21} & l_{\vee t} & L_{22} \end{pmatrix}.$$

Then we can write A_F as

$$A_F = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}.$$

By deleting row t from $L_{F'}$ we obtain the lower Hessenberg matrix

$$H = \begin{pmatrix} L_{11} & & \\ L_{21} & l_{\vee t} & L_{22} \end{pmatrix}$$

with the property that $HH^{\mathrm{T}} = A_F$. We can obtain L_F from H by applying Givens rotations to the column pairs (t, f_{s+1}), $(f_{s+1}, f_{s+2}), \ldots, (f_{m-1}, f_m)$, and deleting the last column of the matrix, i.e.,

$$HG = \begin{pmatrix} L_{11} & & \\ L_{21} & L'_{22} & 0 \end{pmatrix},$$

where G is the matrix of Givens rotations, and

$$L_F = \begin{pmatrix} L_{11} & \\ L_{21} & L'_{22} \end{pmatrix}.$$

However, if we apply the Givens rotations in this way, L'_{22} will be stored in columns t through f_{m-1}. Recall that we are labeling each column of L_F according to the corresponding column of A, since we will be storing column j of L_F in that part of the data structure set up for column j of L. Thus L'_{22} should be stored in the same space as L_{22}, i.e. in columns f_{s+1} through f_m. Suppose we apply Givens rotations to the column pairs (t, f_{s+1}), $(t, f_{s+2}), \ldots, (t, f_m)$ of H to zero out column l_{vt}, i.e.,

$$H' = HG = \begin{pmatrix} L_{11} & & \\ L_{21} & 0 & L'_{22} \end{pmatrix},$$

where G is the matrix of Givens rotations. Then $H'H'^{\mathrm{T}} = A_F$ and L'_{22} is where we want it, in columns f_{s+1} through f_m. These are the Givens rotations we will apply in our sparse setting. Since l_{vt} is in general sparse, we do not need to rotate column l_{vt} with every column of L_{22}.

Thus, to obtain L_F from $L_{F'}$ we proceed as follows. Let t be the variable leaving the free set. First, copy column t of $L_{F'}$ into a dense intermediate column vector v. Since we are deleting row t from $L_{F'}$, place a zero in position t of v. Next, zero out the first nonzero position, say position j, of v with column j by applying Givens rotations to the two columns. Now zero out each subsequent nonzero position of v with the appropriate column. The BindVar algorithm below contains the details. Note that here we let l_{vj} refer to column j of $L_{F'}$ below the diagonal.

The BindVar algorithm.

$$v = \begin{pmatrix} 0 \\ l_{vt} \end{pmatrix}$$

while there are nonzeros in v **do**
  j = first nonzero position in v
  compute and apply Givens rotation G such that
  $(v_j \;\; l_{jj})G = (0 \;\; (v_j^2 + l_{jj}^2)^{1/2})$
  for each $i \in \eta(l_{vj})$ **do**
    $(v_i \;\; l_{ij}) = (v_i \;\; l_{ij})G$
  enddo
enddo

Consider the algorithm BindVar. Suppose we are rotating the intermediate column v with column j to zero out position j of v. Since Theorem 3.1 implies that the structure of the jth column of L_F is contained in the structure of the jth column of $L_{F'}$, v must have zeros in positions where column j has structural zeros, otherwise new nonzeros would be created in column j. Therefore we can rotate column j with v by operating only on the structural nonzeros in column j and the corresponding positions in v. Thus we can apply the Givens rotations to $L_{F'}$ to obtain L_F in flops proportional to the number of nonzeros in L_{22}. It is important to note that this update does not require flops proportional to the number of nonzeros *and* zeros in L_{22}. This could be prohibitively large for large n. However, since we are storing $L_{F'}$ in the data structure set up for L, to perform the operations in the *for* loop of the BindVar algorithm, we will need to look through the all nonzeros in column j of L in order to find those in $L_{F'}$.

We will show that the fact that v has zeros in positions where column j has structural zeros is a structural property. This means that these zeros in v are also structural zeros, i.e., they did not occur due to numerical cancellation. The proof is based on the following lemma, due to Parter [23].

Lemma 3.2. *Let j and k be integers satisfying $0<j<k\leq n$. Then*

$$k\in\eta(l_{*j}) \Rightarrow \eta_k(l_{*j})\subseteq\eta(l_{*k}). \qquad \square$$

Theorem 3.3. *In the algorithm BindVar, let $j(k)$ denote the value of the variable j during the kth iteration, and define $j(0)=t$. Then at the kth iteration of the while loop of BindVar, before the Givens rotations are applied, we have*

$$\eta(v)=\eta_{j(k)}(l_{*j(k-1)}) \quad \textit{and} \quad \eta(v)\subseteq\eta(l_{*j(k)}).$$

Proof. The proof is by induction on k. For $k=1$, i.e., on the first pass through the loop, before the Givens rotations

$$v=\begin{pmatrix}0\\ l_{vt}\end{pmatrix},$$

so by Lemma 3.2 the result is true. Assume it is true for k passes through the loop. Before applying the Givens rotations in the kth pass, the induction hypothesis tells us that

$$\eta(v)=\eta_{j(k)}(l_{*j(k-1)}) \quad \text{and} \quad \eta(v)\subseteq\eta(l_{*j(k)}).$$

Upon completion of the kth pass, we have

$$\eta(v)=\eta(l_{*j(k)})-\{j(k)\}.$$

Thus, before applying the Givens rotations in the $(k+1)$st pass, we have

$$\eta(v)=\eta_{j(k+1)}(l_{*j(k)}).$$

Furthermore, by our choice of $j(k+1)$ we have $j(k+1) \in \eta(v)$, so we have $j(k+1) \in \eta(l_{*j(k)})$. Thus Lemma 3.2 implies

$$\eta_{j(k+1)}(l_{*j(k)}) \subseteq \eta(l_{*j(k+1)}).$$

Therefore

$$\eta(v) \subseteq \eta(l_{*j(k+1)}).$$

So the result is true for $k+1$ passes through the loop. □

3.3. *Reordering the free set*

Suppose at the end of the second while loop in GSA we need to reorder the free set and update the Cholesky factorization. This occurs when t, the last variable we freed, is still in the free set when the next constrained stationary point is found. Let $F' = \{f_1, f_2, \ldots, f_{s-1}, f_{s+1}, \ldots, f_m, f_s\}$ denote the current free set, where $m = |F'|$ and $f_s = t$, and let F be the free set after F' is reordered. We will show how to obtain L_F from $L_{F'}$. These ideas were also developed by Björck [3].

Let $(r^T \ \tau)$ be the last row of $L_{F'}$, corresponding to row t of the matrix A. Let us partition $L_{F'}$ into blocks,

$$L_{F'} = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ r_\wedge^T & r_\vee^T & \tau \end{pmatrix},$$

by grouping together rows and columns with labels less than t and rows and columns with labels greater than t. If we reorder the rows and columns of $L_{F'}$ to correspond with the ordering of F then we obtain the matrix

$$H = \begin{pmatrix} L_{11} & & \\ r_\wedge^T & \tau & r_\vee^T \\ L_{21} & 0 & L_{22} \end{pmatrix}$$

with the property that $HH^T = A_F$. Here, H is a lower triangular matrix with a horizontal spike in row t. We can obtain L_F from H by zeroing out the entries of $r_\vee^T$ from right to left using column t. That is, we apply Givens rotations to the column pairs $(t, f_m), (t, f_{m-1}), \ldots, (t, f_{s+1})$ to zero out the entries of $r_\vee^T$. However, since r is in general a sparse vector, we only need to rotate those columns whose component of $r_\vee$ is nonzero.

Thus, to obtain L_F from $L_{F'}$ we proceed as follows. Let t be the variable that is out of order in F'. Zero out a dense intermediate vector v that will be used to accumulate column t of L_F. Now use Givens rotations to zero out the nonzero entries in r which have labels greater than t, proceeding from right to left. The Reorder algorithm below contains the details. Note that here we let $l_{\vee j}$ refer to column j of $L_{F'}$ below the diagonal.

The Reorder algorithm.

$v = 0$
while there are nonzeros in r_v **do**
  j = last nonzero position in r_v
  compute and apply Givens rotation G such that
  $(\tau \ r_j)G = ((\tau^2 + r_j^2)^{1/2} \ 0)$
  for each $i \in \eta(l_{vj})$ **do**
    $(v_i \ l_{ij}) = (v_i \ l_{ij})G$
  enddo
enddo
$l_{*t} = v$

Since Theorem 3.1 implies that the structure of the jth column of L_F is contained in the structure of the jth column of $L_{F'}$, when we rotate v with column j, v must have zeros in positions where column j has structural zeros, otherwise, new nonzeros would be created in column j. Therefore we can rotate column j with v by operating only on the structural nonzeros in column j and the corresponding positions in v. Thus we can apply the Givens rotations to H to obtain L_F in flops proportional to the number of nonzeros in L_{22}, rather than proportional to the number of nonzeros *and* zeros in L_{22}. However, since we are storing $L_{F'}$ in the data structure set up for L, to perform the operations in the *for* loop of the Reorder algorithm, we will need to look through the all nonzeros in column j of L in order to find those in $L_{F'}$.

The fact that when we rotate v with column j, v must have zeros in positions where column j has structural zeros is not a structural property, i.e., numerical cancellation occurs to create these zeros. This does not cause any numerical difficulties since the forward error in L_F is no greater when L_F is computed by ignoring the structural zeros than it would be if L_F were computed by operating on them [1, 3].

4. Improving the basic algorithm

In what follows we describe several possible modifications to GSA. The algorithm GSA binds and frees only one variable at a time. Because of this, it runs very slowly on large sparse problems. In Section 4.1, we show that by projecting each direction, we can bind several variables for each search direction. However because of the way GSA computes search directions, it cannot free more than one variable at a time. In Section 4.2, we propose a new method for computing search directions that allows us to free more than one variable at a time. For this technique, A_F need not remain positive definite. In light of this, we present a heuristic method for finding a starting point and an initial active set in Section 4.3. We can combine these techniques in various ways and use them in one or both while loops of GSA. We defer discussion of these possibilities to Section 5, where we present some numerical results to compare them.

4.1. Projecting the descent direction

The algorithm described in Section 2.1 does a lot of work for each variable it binds; in particular, it updates the Cholesky factorization of L_F, solves a linear system, and performs a matrix vector multiply to update g_F. Thus, we consider using each search direction to bind several variables. This idea has been used by many authors [2, 4, 6] and is outlined below.

Follow the search direction until a constraint is violated and remove the corresponding variable from the free set. Now project the search direction onto the space of free variables. If this direction is a descent direction follow it until a local minimum along it is reached or a constraint becomes active. Continue projecting the direction until it is no longer a descent direction or a local minimum along that direction has been reached. Then recompute the factorization and compute a new search direction.

To formalize this, we introduce the notation of a projection operator Π defined componentwise by

$$[\Pi(x)]_i = \begin{cases} -1 & \text{if } x_i \leq -1, \\ x_i & \text{if } -1 < x_i < -1, \\ 1 & \text{if } x_i \geq 1. \end{cases}$$

Given a current point x and a descent direction s, we update x via

$$x := \Pi(x + \alpha s),$$

where $\alpha > 0$ is the first local minimum of $q(\Pi(x+\alpha s))$. Conn, Gould and Toint [7] give a very efficient algorithm for finding such a local minimum, requiring one sparse matrix-vector multiplication and two vector inner-products to begin, and two sparse inner products for each variable bound.

4.2. A new search direction when A_F is indefinite

In the GSA algorithm, we free only one variable at a time. If the matirx A is indefinite, when we free a variable the matrix A_F may become indefinite. If this occurs and we attempt to continue factoring A_F, say with an LDL^T factorization, we may lose numerical stability. We do not want to allow pivoting for numerical stability, since that would entail a more complicated data structure. Here we introduce a new algorithm for computing a descent direction when A_F is indefinite that allows us to free many variables at a time without losing numerical stability.

The idea behind our new step is that we want to step toward an approximate global minimum of the subproblem

$$\min \tfrac{1}{2} s^T A_F s + g_F^T s, \quad \|s\|_2 \leq \Delta, \tag{5}$$

i.e., an approximate global minimum of q_F in a circle of radius Δ centered at our current point. We set Δ equal to $(1+\beta)$ times the diameter of the smallest circle that is centered at our current point and contains the feasible region, where β and

γ are tolerances satisfying $0<\beta<\gamma$ and $\gamma<1$. The larger we choose β and γ, the less accurately we solve (5). We make this choice of Δ to insure that s has the properties we require for convergence.

Problem (5) has been extensively studied, and there exist efficient algorithms and implementations for solving it. We base our algorithm for finding a new search direction on the trust region algorithm by Moré and Sorensen [20], making a few modifications to insure that our new step has the properties that we require for convergence. The next three paragraphs describe their trust region algorithm.

There are two possible forms that the solution of (5) can take. If there exists $\alpha \geq 0$ such that

$$(A_F+\alpha I)>0, \qquad (A_F+\alpha I)s=-g_F \quad \text{and} \quad \|s\|_2=\Delta, \tag{6}$$

then $s(\alpha)=-(A_F+\alpha I)^{-1}g_F$ is the unique solution to (5). We refer to this situation as the *easy* case. The *hard* case occurs when $\|s(\alpha)\|_2<\Delta$ for all α such that $(A_F+\alpha I)>0$. This can happen when g_F is orthogonal to the eigenspace corresponding to the smallest eigenvalue of A_F. Let λ_1 be the smallest eigenvalue of A_F and $\hat{z}^*$ be an associated unit eigenvector. In this case, the solution to (5) is $s=(A_F-\lambda_1 I)^+ g_F+\sigma\hat{z}^*$, where $^+$ denotes pseudo-inverse, and $\alpha=\sigma$ is the root with the smaller norm of the quadratic $\|s(\alpha)\|_2^2-\Delta^2$.

To find the solution to (5), the trust region algorithm applies safeguarded Newton's method to solve

$$f(\alpha)=\frac{1}{\Delta}-\frac{1}{\|s(\alpha)\|_2}=0. \tag{7}$$

Given an initial guess of α, it attempts the Cholesky factorization of $(A_F+\alpha I)$. If this factorization is successful, the algorithm uses it to compute the Newton step for (7) and then takes this step, subject to some safeguarding. However, if the Cholesky factorization breaks down, the algorithm modifies α and tries again. Since there is no convenient way of knowing when to use the hard case, i.e., when there is no α that solves (6), if a particular α yields $\|s(\alpha)\|_2<\Delta$, then the algorithm tries the hard case. Using the Cholesky factorization of $(A_F+\alpha I)$, it computes $\hat{z}$, a unit approximation to an eigenvector associated with λ_1, the smallest eigenvalue of $(A_F+\alpha I)$. Then it computes σ, the root with smaller magnitude of the quadratic $\|s(\alpha)+\sigma\hat{z}\|_2^2-\Delta^2$. This choice of σ guarantees that $\sigma\hat{z}$ is a nonascending direction of q_F at $x+s(\alpha)$. Finally, it determines if $s(\alpha)+\sigma\hat{z}$ satisfies the termination criteria. If not, it repeats the process with a new α.

The algorithm requires a tolerance γ, and terminates when

$$\alpha=0 \quad \text{and} \quad \|s(\alpha)\|_2\leq\Delta^2, \tag{8}$$

or

$$(1-\gamma)\Delta\leq\|s(\alpha)\|_2\leq(1+\gamma)\Delta, \tag{9}$$

or, in the hard case, when

$$\|s(\alpha)+\sigma\hat{z}\|_2=\Delta$$

and

$$\sigma^2\hat{z}^{\mathrm{T}}(A_F+\alpha I)\hat{z}\leq\gamma(2-\gamma)(s(\alpha)^{\mathrm{T}}(A_F+\alpha I)s(\alpha)+\alpha\Delta^2).$$

These conditions insure that the approximate solution satisfies

$$q_F(s(\alpha)+\sigma\hat{z})-q_F(x^*)\leq\gamma(2-\gamma)|q_F(x^*)|,$$

where $q_F(x^*)$ is the global minimum of (5).

We show how we can adapt Moré and Sorensen's algorithm so it yields a step for which the proof of convergence in Section 2.2 remains true. In particular, in order for Lemma 2.2 to remain true, it suffices to guarantee that the function value is lower at our new point and a new constraint is encountered.

When the trust region algorithm finds a solution using the easy case, we set $s=s(\alpha)$. In this case, (6) shows us that $g_F^{\mathrm{T}}s<0$. If $s^{\mathrm{T}}A_Fs\leq 0$ then we can follow s until we encounter a new constraint and we will decrease the function value. If $s^{\mathrm{T}}A_Fs>0$, the step to the minimum of q_F along s has length

$$\frac{g_F^{\mathrm{T}}s}{s^{\mathrm{T}}A_Fs}=\frac{s^{\mathrm{T}}(A_F+\alpha I)s}{s^{\mathrm{T}}A_Fs}>1.$$

To ensure that a step length less than or equal to 1 is sufficient to bring us to a new constraint, we replace condition (9) with the stronger condition

$$\|s(\alpha)\|_2\leq(1+\gamma)\Delta \quad \text{and} \quad \|s(\alpha)\|_\infty\geq 1. \tag{10}$$

With this choice of s, we are guaranteed that we can follow it until we encounter a new constraint and we will decrease the function value.

We need to be sure that the trust region algorithm can find an α that satisfies (10). Since $\gamma\in(0,1)$ is arbitrary, the trust region algorithm can find α satisfying

$$(1-\beta)\Delta\leq\|s(\alpha)\|_2\leq(1+\beta)\Delta, \tag{11}$$

i.e., condition (9) with γ set equal to β. By our choice of Δ, condition (11) implies condition (9). Thus, the trust region algorithm can find such an α.

When the trust region algorithm finds the solution using the hard case, we use a two part step in our algorithm. First we update $x=x+Z_Fs(\alpha)$, and then we set $s=\hat{z}$. As we showed above, $q(x+Z_Fs(\alpha))<q(x)$. Thus, if $g_F^{\mathrm{T}}\hat{z}\leq 0$ and $\hat{z}$ is a direction of nonpositive curvature, we can follow $\hat{z}$ until we encounter a constraint and be assured that our two part step has produced a function decrease. To accomplish this, we modify the choice of $\hat{z}$ in the trust region algorithm. In order to guarantee convergence, the trust region algorithm requires that the estimate $\hat{z}$ approach a singular vector of $(A_F+\alpha I)$ as $\alpha\to-\lambda_1$. To compute $\hat{z}$, it applies a Linpack [9] type condition estimator to the matrix $(A_F+\alpha I)$. However, for our algorithms, we also need $\hat{z}$ to be a direction of nonpositive curvature. We ensure this as follows.

When we attempt to compute the Cholesky factorization of A_F and it breaks down, we compute a direction of nonpositive curvature as in Section 2.1. We normalize this vector and call it $\hat{y}$. Note that $\hat{y}$ has the property that

$$\lambda_1 \leq \hat{y}^T A_F \hat{y} \leq 0. \tag{12}$$

We then begin the trust region algorithm. Each time we need to compute $\hat{z}$, we compute a vector $\hat{v}$ using the trust region algorithm's condition estimator. We compare $\hat{v}$ to $\hat{y}$, and set $\hat{z}$ to be whichever of the two directions is a direction of greater negative curvature. Equation (12) and the fact that $\hat{v}$ approaches a singular vector of $(A_F + \alpha I)$ as $\alpha \to -\lambda_1$ assure us that $\hat{z}$ will always be a direction of nonpositive curvature while maintaining the condition that $\hat{z}$ approaches a singular vector as $\alpha \to -\lambda_1$.

In summary, we compute our search direction s as follows. Free all the variables whose gradients have the incorrect sign. Attempt the Cholesky factorization of A_F. If it succeeds, set $s := -A_F^{-1} g_F$. Otherwise, perform the following steps. Compute the direction of nonpositive curvature $\hat{y}$ from the breakdown of Cholesky, as described in Section 2.1. Then compute $s(\alpha)$ and, if appropriate, $\hat{z}$, as described above. Finally, if $\|x + Z_F s(\alpha)\|_\infty \geq 1$, then set $s := s(\alpha)$; otherwise, update $x := x + Z_F s(\alpha)$ and set $s := \hat{z}$.

To implement the trust region algorithm for sparse problems, we need to compute the Cholesky factorization of matrices of the form $(A_F + \alpha I)$, where $(A_F + \alpha I) > 0$. Since the sparsity structures of $(A_F + \alpha I)$ and A_F are the same, we can use the same data structure to store their Cholesky factors. Notice that as α changes, the Cholesky factor of $(A_F + \alpha I)$ changes significantly, and our updating scheme cannot be used to obtain the new factor. Thus we must recompute the entire Cholesky factor of $(A_F + \alpha I)$ each time α changes.

4.3. Finding a starting point

If we use the new descent direction algorithm, we do not need to start with a free set F such that A_F is positive definite. This allows us to use a preprocessing heuristic to find a starting point that quickly binds a subset of the variables, hopefully with as much accuracy as possible. Our heuristic uses global information to select which variables to bind in much the same way as our new search direction algorithm did.

We find an initial active set and a starting point as follows. Let $F := \{1, \ldots, n\}$ and $x := 0$. Repeat the following steps a finite number of times. First, find a point s which is an approximate global minimizer of q_F on a simpler region, for example a sphere centered at the current point. Then find the point p in the feasible region closest to $x + Z_F s$. Let p be the starting point and compute $g(p)$. Finally, bind any variable that is at a bound and whose corresponding component of $g(p)$ has the correct sign, and update F. To determine how many times to repeat this loop, we introduce three parameters m_1, m_2 and m_3. We terminate after we have executed the loop m_1 times, or we have bound less than m_2 variables in the past m_3 iterations.

When A_F is positive definite, we solve the unconstrained problem

$$\min \tfrac{1}{2}s^T A_F s + g_F^T s.$$

This amounts to setting s to be the Newton direction. However when A_F is not positive definite, this quadratic may be unbounded below. Instead we find an approximate solution to a problem

$$\min \tfrac{1}{2}s^T A_F s + g_F^T s, \quad \|s\|_2 \leq \Delta,$$

for some appropriate Δ as described in the previous section. Using our projection operator notation, we are setting $p := \Pi(x + Z_F s)$.

5. Numerical results

In what follows, we describe the implementation details of GSA and various modifications to GSA that incorporate our heuristic improvements of Section 4. We provide numerical results for a battery of test problems and use these results to suggest the algorithm which we feel to be the most useful in practice. We also briefly discuss some less successful approaches.

5.1. *Implementation details*

We implemented our algorithms in Fortran in double precision and made extensive use of Sparspak [13] in our implementation. We use Sparspak's GENMMD routine to find an initial minimum degree ordering of A and SMBFCT to perform the symbolic factorization and set up the data structure for L. We modified Sparspak's Cholesky factorization routine GSFCT and triangular solve routine GSSLV to allow us to ignore any specified set of rows and columns.

To compute the search directions of Section 4.2, we modified the Fortran code GQTPAR of Moré and Sorensen [20] to handle sparse matrices. To compute the Cholesky factorization and perform forward and backward triangular solves we use the modified Sparspak routines GSFCT and GSSLV. We slightly modified Sparspak's condition estimator routines COND51 and COND52 to produce the eigenvector approximations we need.

We generated problems by using Coleman's routine [5] that, given the structure of a matrix A, generates a lower triangular matrix L with the property that $\eta(LL^T) \subseteq \eta(A)$. We then set $A = LDL^T$, where D is a diagonal matrix of plus and minus ones. We control the condition number of A by choosing the values of the diagonal of L, and determine the number of negative eigenvalues of A by selecting the number of -1's on the diagonal of D.

For positive definite A, we generate the b vectors as follows. Given parameters θ and μ, we generate a feasible vector x by setting θ-percent of the x_i to a bound, and assigning the rest randomly in the open interval $(-1, 1)$; we set μ-percent of

the bound x_i to negative bounds. We then set $g = Ax$ and

$$b_i = \begin{cases} -g_i & \text{if } -1 < x_i < 1, \\ -g_i - v_i & \text{if } x_i = 1, \\ -g_i + v_i & \text{if } x_i = -1, \end{cases}$$

where the v_i's are random numbers in (0, 10). Note that since A is positive definite, x is the solution of our problem. This allows us to control the percentage of variables bound at the solution, a critical factor in the running times of these problems. For indefinite problems, we generate the b vectors by setting $b = Ax$, where the components of x are randomly chosen in $(-1, 1)$. We chose this interval in an attempt to generate problems with many local minima.

The heuristics of Sections 4.2–4.3 contained the parameters γ, β, m_1, m_2 and m_3 that we left unspecified. Based on some limited testing, we feel that reasonable choices of these parameters are $\gamma = 0.9$, $\beta = 0.1$, $m_1 = 10$, $m_2 = 2$ and $m_3 = 8$.

5.2. The algorithms

We began by implementing GSA and found it to be slow on positive definite problems. In particular, it spends a great deal of time for each variable it binds, especially in the beginning when the free set is large. To remedy this we incorporated the projection techniques into GSA. When we modified GSA by adding projection techniques in both the main loop of the program and in the initial stationary point loop, we found that it ran slower because we could bind many variables at a time but only free them one at a time. Thus we included projection techniques only in the initial stationary point loop and we refer to this algorithm as PSA (Projected Sparse Active set algorithm). PSA tended to run much faster than GSA, especially on problems that were moderately or heavily bound at the solution. This is because GSA is more accurate in selecting which variables to bind; PSA is less accurate, but it binds variables more quickly. On the problems that were moderately or heavily bound at the solution, the amount of time spent binding was more important, since a large proportion of the variables needed to be bound. On the problems that were lightly bound at the solution, binding the wrong variables took considerable time to remedy, since we are freeing them one at a time. The algorithm PSA shows the value of the projection techniques.

For indefinite and negative definite problems, GSA and PSA found local minima with approximately the same function value. They ran very quickly on these problems, since many of the variables were bound during the initial Cholesky factorization.

Next we modified GSA by using the new search direction for the indefinite case. Each time we reach a constrained stationary point, we free all of the variables whose gradients have incorrect sign, something we could not do when we used the Cholesky breakdown direction for the indefinite case. Because computation of our new direction does not update the Cholesky factorizations, computation of each direction

is expensive. Thus we project each search direction to allow us to bind several variables for each direction we compute. Since the direction is an approximation to the minimum of q_F on an ellipse centered at the current point, we call the algorithms that use the new search direction Ellipse Active set algorithms. The algorithm with these improvements PEA (Projected Ellipse Active set algorithm) ran much faster than PSA on positive definite problems. On indefinite problems, PEA found local minima that were lower than those found by GSA or PSA. The running time of PEA on indefinite problems was considerably longer than PSA, since PEA uses global information. However, the running time of PEA on indefinite problems was comparable to its running time on positive definite problems with the same structure and size. The algorithm PEA highlights the value of our new search direction and of freeing many variables at a time.

Next we incorporated the starting point heuristic into PEA. We call the resulting algorithm PEAS. (Projected Ellipse Active set algorithm with Starting point heuristic) PEAS ran much faster than PEA on positive definite problems. The running time of PEAS on indefinite problems was faster than PEA, and PEAS found local minima that were lower than those found by PEA. The algorithm PEAS shows the value of our starting point heuristic.

Finally, we implemented an algorithm PEGAS (Projected Ellipse General Active set algorithm with Starting point heuristic) that was identical to PEAS until it found an initial stationary point and was identical to GSA after that. PEGAS ran faster than PEAS on positive definite problems, ran marginally slower on indefinite problems, and found local minima with almost identical function values as those found by PEAS. This seems to indicate that once a good stationary point is found, switching back to the GSA algorithm with its efficient Cholesky factor updating and its adding and dropping of one constraint at each iteration works best.

In Table 1 in the Appendix, we tabulate the features of the algorithms we have just described. We refer to the direction used by GSA for nonpositive definite problems as *breakdown*, since we compute it from the Cholesky factorization when it breaks down.

We need to consider the convergence of these algorithms. In Sections 4.1–4.2 we saw that for each direction we follow, we bind at least one variable, and if we take a step of nonzero length, we decrease the function value. These facts, together with the arguments used to show convergence of GSA, ensure the convergence of the algorithms in Table 1 to a point satisfying (3) with $A_F > 0$ in a finite number of iterations.

5.3. Results

We ran our double precision Fortran algorithms on a Sun 3/50 under UNIX 4.2 Release 3.5 with the parameters described in Section 5.1. We tested our algorithms on positive definite and indefinite problems. We chose three approximate condition number ranges: 10^3–10^5, 10^8–10^{10} and 10^{13}–10^{15}. (Note that machine epsilon is about

10^{-19}.) To approximate the condition number κ of each problem, we used Higham's Condest routine [18]. To solve the systems required by Condest, we used Gilbert and Peierls LU factorization code [14]. We used three sparsity structures from Everstine [10]: the problem of size 503 and 6027 nonzeros, the problem of size 1005 with 8621 nonzeros, and the problem of size 2680 with 25026 nonzeros. The tables referred to in this section are in the Appendix, along with Table 1, which describes the algorithms.

For positive definite problems, we chose three values of θ, the percentage of variables bound at the solution: 10, 50 and 90. For each of the 9 different combinations of κ and θ, we generated four different quadratics, yielding a total of 36 problems for each sparsity pattern. For each condition number range, Table 2 lists the sum of the running times in seconds of the 12 problems with that range; for each percentage of variables bound at the solution, Table 3 lists the sum of the running times in seconds of the 12 problems with that percentage of variables bound. As we stated in Section 5.2, PEGAS runs the fastest on these problems, with PEAS slightly slower. The other algorithms are several times slower.

For indefinite problems, we chose three values for the percentage of eigenvalues of A that were negative: 10, 50 and 90. For each of the 9 different combinations of κ and the percentage of negative eigenvalues, we generated four different quadratics, yielding a total of 36 problems for each sparsity pattern. To compare running times, for each percentage of negative eigenvalues, Table 4 shows the sum of the running times in seconds of the 12 problems with that percentage. The ellipse algorithms display much less sensitivity to the percentage of negative eigenvalues than GSA or PSA, although the latter are considerably faster on problems with a high percentage of negative eigenvalues. To compare the local minima found, for each percentage of negative eigenvalues, Table 5 displays the normalized sum of the function values at the local minimum found in the 12 problems with that percentage. Also, for each condition number range, Table 6 displays the normalized sum of the function values at the local minimum found in the 12 problems with that range. We normalize the sum of the function values with the formula

$$\text{normalized total value} = \frac{\text{lowest total value}}{\text{total value}},$$

so that the lowest function value is normalized to 1, and the others are normalized to values greater than 1. Of the five algorithms, PEGAS and PEAS find the lowest minima, with PEA slightly higher.

Tables 7–11 show the approximate percentage of time spent performing the major tasks of each algorithm for two problem sizes, 503 and 1005, on 6 representative problems. All six problems have condition numbers in the range 10^8–10^{10}. Three are positive definite, with 10, 50 and 90 percent of the variables bound at the solution; three are indefinite, with 10, 50 and 90 percent of the eigenvalues negative. The tables divide the algorithm up into two or three main parts: the starting point heuristic (if appropriate), the initial constrained stationary point loop, and the main

loop of the program. They tabulate separately the tasks performed in each major part. Since we also set up the data structure, permute the matrix, and compute an initial factorization (if appropriate), the percentage of time spent in these three tasks does not add up to 100. Most of the tasks listed are self explanatory, but a few merit further clarification. In Table 8, the task *project* includes computing α and updating g and L_F. In Tables 9–11, the task *solve* consists of solving for the Newton direction or the ellipse direction, as appropriate, and the task *project* includes updating g.

In general, GSA spends most of its time in the initial stationary point loop. Solving for the search direction and updating the gradient within this loop are its most time consuming tasks. Compared to GSA, PSA spends more of its time in the main loop. Its most time consuming tasks are projecting the search direction in the initial stationary point loop and solving for the search direction and updating the gradient in the main loop. Except for lightly bound positive definite problems, PEA spends most of its time computing the initial stationary point. For positive definite problems, factoring the matrix and projecting the search direction in both loops are its most time consuming tasks; for indefinite problems, solving for and projecting the search direction in the initial stationary point loop are its most time consuming tasks. The algorithms PEAS and PEGAS spend most of their time computing the starting point. Notice, however, that the starting point heuristic sometimes finds an initial stationary point and even an optimal point. (This may be due to some special feature present in our test problems because of the way we generate them, such as the orientation of the eigenvectors of A. We also found this behavior on randomly generated indefinite problems. It seems likely that there are classes of problems for which this is not frequently true, but that our sparse problem generator did not tend to generate such problems.) For these algorithms, the starting point heuristic spends most of its time factoring in positive definite problems, and solving for the search direction in indefinite problems.

Tables 12–13 shows the average number of search directions computed for the problems of size 503. The ellipse algorithms tend to compute fewer search direction than either GSA or PSA. In general, the new search direction algorithm for indefinite A_F only requires one additional Cholesky factorization to solve the trust region problem after indefiniteness is discovered.

Table 14 shows the average percentage of variables bound at the solution for the indefinite problems of size 503. For each group of problems, all of the algorithms tend to find local minima with approximately the same percentage of variables bound at the solution.

Tables 15–16 display the average number of variables freed for the problems of size 503. In the algorithms that use the starting point heuristic, all variables that the heuristic sets to a bound are counted as bound; those whose corresponding component of the gradient have the incorrect sign are then counted as freed. We do not include the number of variables bound, since for positive definite problems, it is just a constant plus the number of variables freed, and for indefinite problems,

Table 14 shows that it is almost a constant plus the number of variables freed. Thus the number of variables freed is a good measure of how well the algorithms choose which variables to bind and free. For positive definite problems, notice that GSA frees many fewer variables than the other algorithms, and that in almost all cases, the problems with higher condition numbers have more variables freed. For indefinite problems, the ellipse algorithms tend to free fewer variables, especially those algorithms with the starting point heuristic.

5.4. Other possibilities

There are obviously many ways to combine the improvements of Section 4 into GSA. Some behaved similarly to the algorithms we presented so we did not tabulate results for them. In particular, when we used our starting point heuristic, projecting the search directions did not substantially affect the running times or local minima found. Other combinations performed very poorly, e.g., using our new search direction without projecting the search directions and without the starting point heuristic. Some other ideas we tried briefly that did not seem to work well include starting GSA at the vertex nearest (in the infinity norm sense) to the global minimum of the unconstrained quadratic and freeing variables after every iteration for the first few iterations of PEA.

We also tried other ways of using trust region ideas in our algorithm that were less successful in practice. Above we solved a trust region problem to find the search direction when A_F was not positive definite. We also tried solving a trust region problem to find the search direction when A_F was positive definite and the Newton step was not feasible. We implemented two different versions: one where we used this technique for finding a search direction only when we were finding a starting point and initial stationary point, and another where we used this technique for finding all search directions. The former was more successful but neither method worked as well as the ones that used the Newton step.

6. Conclusion

We showed that direct active set methods that solve both definite and indefinite quadratic programs with simple bounds can be efficiently implemented for large sparse problems. All of the necessary factorizations can be carried out in a static data structure requiring space equal to that needed for a single Cholesky factorization of A. We can efficiently update the Cholesky factorization of A_F if we are only binding or freeing one variable at a time.

We showed that our new descent direction for the indefinite case and our new heuristic method for finding a starting point, combined with projection techniques, produced an algorithm that runs much faster than the basic algorithm for positive

definite problems and finds local minima with lower function values for indefinite and negative definite problems.

There may be ways we can improve our algorithm. For example, we could use a line search procedure along the piecewise quadratic $q(\pi(x+\alpha Z_F s))$ instead of exactly computing the first local minimum, or incorporate projected gradients into an algorithm using our new descent direction algorithm and/or our new heuristic starting point algorithm.

Our results may have some applications in solving the least squares problem

$$\min \tfrac{1}{2}x^T C^T C x + d^T x, \quad l \leq x \leq u. \tag{13}$$

As we mentioned in Section 1, Björck [3] proposed an algorithm for solving (13) that is very similar to GSA. Our improvements could be added to such an algorithm. However, computing and updating the Cholesky factorization is more complicated in the least squares case; in particular, since we do not wish to compute $A^T A$, adding a row and column to L_F cannot be accomplished stably with a single triangular solve. All of the algorithms we propose vary the active set more than GSA does. Thus the time saved by our improvements may not be as great in the least squares case, and there may be tradeoffs in stability.

Our results show that minimizing a quadratic with simple bounds may be very expensive compared to minimizing the quadratic subject to a two-norm bound on the solution, since we solve several two-norm problems in our fastest algorithm. This suggests that for large sparse problems, minimizing a quadratic with simple bounds as a subproblem of a more difficult problem, e.g., minimizing a nonlinear function subject to simple bounds, may not be very practical.

Acknowledgements

Computational experiments were performed in the Cornell Computational Optimization Project laboratory, partially supported by the Computational Mathematics Program of the National Science Foundation under grant DMS-8706133 and the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under grant DE-FG02-86ER25013.A000.

We thank Nick Higham for giving us his condition estimation code and John Gilbert and Tim Peierls for giving us their sparse LU factorization code. We also thank Earl Zmijewski for a careful and critical reading of the manuscript. We acknowledge a referee for comments that improved the understandability of the paper.

Appendix

Table 1

The features of the algorithms

Method	Heuristic starting pt.	Initial stationary pt.		Main loop	
		neg. curv. dirn.	projecting	neg. curv. dirn.	projecting
GSA	no	breakdown	no	breakdown	no
PSA	no	breakdown	yes	breakdown	no
PEA	no	new	yes	new	yes
PEAS	yes	new	yes	new	yes
PEGAS	yes	new	yes	breakdown	no

Table 2

Running times (secs.) vs. condition number for positive definite problems

n	κ	GSA	PSA	PEA	PEAS	PEGAS
503	10^3–10^5	1522.54	688.02	312.06	158.34	138.88
	10^8–10^{10}	1666.50	739.24	440.54	192.66	161.54
	10^{13}–10^{15}	1699.78	760.36	528.92	218.76	181.90
	total	4888.82	2187.62	1281.52	569.76	482.32
1005	10^3–10^5	*	1999.68	1085.96	559.64	473.66
	10^8–10^{10}	*	2049.50	1480.64	560.86	491.40
	10^{13}–10^{15}	*	2159.48	1956.78	623.48	525.54
	total	*	6208.66	4523.38	1743.98	1490.60
2680	10^3–10^5	*	*	3690.46	912.86	769.70
	10^8–10^{10}	*	*	4649.44	979.24	854.26
	10^{13}–10^{15}	*	*	5478.60	1213.66	944.40
	total	*	*	13818.50	3105.76	2568.36

* Requires excessive CPU time.

Table 3

Running times (secs.) vs. percent of variables bound for positive definite problems

n	% bnd.	GSA	PSA	PEA	PEAS	PEGAS
503	10	1530.12	1222.38	701.38	371.22	288.44
	50	1675.80	605.34	320.66	118.56	115.10
	90	1682.90	359.90	259.48	79.98	78.78
	total	4888.82	2187.62	1281.52	569.76	482.32
1005	10	*	3409.14	2290.84	1058.52	832.48
	50	*	1795.04	1159.60	410.50	384.26
	90	*	1004.48	1072.94	271.20	273.86
	total	*	6208.66	4523.38	1743.98	1490.60
2680	10	*	*	5747.06	1976.96	1520.40
	50	*	*	4349.32	677.74	611.02
	90	*	*	3722.12	451.06	436.94
	total	*	*	13818.50	3105.76	2568.36

* Requires excessive CPU time.

Table 4

Running times (secs.) vs. percentage of negative eigenvalues for indefinite problems

n	% neg. ev.	GSA	PSA	PEA	PEAS	PEGAS
503	10	630.76	193.76	194.54	178.94	180.14
	50	128.68	62.86	192.52	166.76	166.42
	90	33.46	34.38	179.80	154.78	155.56
	total	792.90	291.00	566.86	500.48	502.12
1005	10	2652.46	529.66	613.12	523.36	544.98
	50	469.96	167.22	643.90	503.34	532.56
	90	91.88	89.38	650.50	461.04	475.60
	total	3214.30	786.26	1907.52	1487.74	1553.14
2680	10	*	3004.82	2711.46	1099.54	1294.56
	50	2978.92	734.04	3054.84	1092.96	1164.40
	90	271.78	242.22	3007.52	873.78	850.88
	total	*	3981.08	8773.82	3066.28	3309.84

* Requires excessive CPU time.

Table 5

Normalized function value vs. condition number for indefinite problems

n	κ	GSA	PSA	PEA	PEAS	PEGAS
503	10^3-10^5	1.039	1.039	1.016	1.000	1.000
	10^8-10^{10}	1.030	1.030	1.015	1.000	1.000
	10^{13}-10^{15}	1.030	1.030	1.013	1.000	1.000
1005	10^3-10^5	1.022	1.022	1.008	1.000	1.000
	10^8-10^{10}	1.017	1.017	1.005	1.000	1.000
	10^{13}-10^{15}	1.017	1.016	1.005	1.000	1.000
2680	10^3-10^5	*	1.025	1.011	1.000	1.000
	10^8-10^{10}	*	1.013	1.007	1.000	1.000
	10^{13}-10^{15}	*	1.013	1.007	1.000	1.000

* Requires excessive CPU time.

Table 6

Normalized function value vs. percentage of negative eigenvalues for indefinite problems

n	% neg. ev.	GSA	PSA	PEA	PEAS	PEGAS
503	10	1.002	1.002	1.000	1.000	1.000
	50	1.030	1.030	1.010	1.000	1.000
	90	1.052	1.052	1.026	1.000	1.000
1005	10	1.002	1.002	1.000	1.000	1.000
	50	1.012	1.011	1.004	1.000	1.000
	90	1.035	1.035	1.011	1.000	1.000
2680	10	*	1.001	1.000	1.000	1.000
	50	1.004	1.004	1.003	1.000	1.000
	90	1.037	1.037	1.018	1.000	1.000

* Requires excessive CPU time.

Table 7

Percentage of time spent performing subtasks in GSA

		Positive definite						Indefinite					
		503 % bnd.			1005 % bnd.			503 % neg. ev.			1005 % neg. ev.		
		10	50	90	10	50	90	10	50	90	10	50	90
init. stat. pt.	solve	34	45	51	47	58	62	36	22	2	54	34	3
	update g	15	23	26	13	19	21	35	26	1	27	30	5
	update L_F	8	11	12	4	6	6	3	3	1	3	1	2
	total	61	85	96	69	89	96	84	61	8	93	77	14
main loop	solve	13	3	0	13	3	0	2	2	1	0	1	0
	update g	6	3	0	4	2	0	3	4	1	0	1	0
	update L_F	13	3	0	8	2	0	1	1	0	0	0	0
	total	35	11	1	27	8	1	8	13	10	2	4	3

Table 8

Percentage of time spent performing subtasks in PSA

		Positive definite						Indefinite					
		503 % bnd.			1005 % bnd.			503 % neg. ev.			1005 % neg. ev.		
		10	50	90	10	50	90	10	50	90	10	50	90
init. stat. pt.	solve	2	2	4	1	1	4	2	1	0	2	1	0
	project	20	41	68	17	33	57	25	18	6	36	21	5
	total	21	43	72	18	34	60	27	19	6	37	21	5
main loop	solve	23	11	1	33	16	1	8	6	1	6	2	0
	update g	15	15	2	16	16	2	19	8	1	8	5	1
	update L_F	26	9	1	18	7	1	7	3	0	3	2	0
	total	75	49	14	76	55	21	50	32	11	38	25	5

Table 9

Percentage of time spent performing subtasks in PEA

		Positive definite						Indefinite					
		503 % bnd.			1005 % bnd.			503 % neg. ev.			1005 % neg. ev.		
		10	50	90	10	50	90	10	50	90	10	50	90
init. stat. pt.	solve	3	4	6	1	2	4	30	24	43	31	28	27
	factor	21	30	46	15	32	51	2	1	0	1	0	0
	project	10	29	32	12	32	31	52	52	42	57	60	64
	total	34	64	84	29	67	86	84	76	86	89	88	91
main loop	solve	8	4	0	5	3	0	1	2	0	1	0	0
	factor	49	16	1	58	16	2	3	1	0	0	0	0
	project	7	10	7	5	10	7	2	13	6	1	3	1
	total	63	30	9	69	28	9	6	15	7	2	4	1

Table 10

Percentage of time spent performing subtasks in PEAS

		Positive definite						Indefinite					
		503 % bnd.			1005 % bnd.			503 % neg. ev.			1005 % neg. ev.		
		10	50	90	10	50	90	10	50	90	10	50	90
start pt.	solve	5	8	9	4	7	6	75	74	77	83	84	80
	factor	37	53	56	52	64	70	4	2	1	2	1	1
	project	3	8	12	2	4	6	10	12	10	4	5	7
	total	46	71	79	58	75	82	89	89	89	90	91	88
init. stat. pt.	solve	2	0	0	1	0	0	0	0	0	0	0	0
	factor	13	0	0	11	0	0	0	0	0	0	0	0
	project	1	0	0	0	0	0	0	0	2	0	0	1
	total	16	0	0	12	0	0	0	0	2	0	0	1
main loop	solve	4	3	0	2	1	0	0	0	0	0	0	0
	factor	28	11	0	22	10	0	0	0	0	0	0	0
	project	2	3	0	1	1	0	0	0	0	0	0	0
	total	34	16	0	24	13	0	0	0	0	0	0	0

Table 11
Percentage of time spent performing subtasks in PEGAS

		Positive definite						Indefinite					
		503 % bnd.			1005 % bnd.			503 % neg. ev.			1005 % neg. ev.		
		10	50	90	10	50	90	10	50	90	10	50	90
start pt.	solve	7	9	8	6	7	6	75	74	77	82	84	80
	factor	51	54	56	67	72	70	4	2	1	2	1	1
	project	5	9	12	2	5	5	10	12	10	4	5	6
	total	63	73	78	75	85	82	89	89	89	89	91	88
init. stat. pt.	solve	3	0	0	1	0	0	0	0	0	0	0	0
	factor	18	0	0	14	0	0	0	0	0	0	0	0
	project	2	0	0	0	0	0	0	0	2	0	0	1
	total	22	0	0	15	0	0	0	0	2	0	0	1
main loop	solve	34	4	0	2	1	0	0	0	0	0	0	0
	update g	1	3	0	0	0	0	0	0	0	0	0	0
	update L_F	5	4	0	1	0	0	0	0	0	0	0	0
	total	10	13	0	3	2	0	0	0	0	0	0	0

Table 12
Average number of search directions computed for positive definite problems

		κ		
	% bnd.	10^3-10^5	10^8-10^{10}	10^{13}-10^{15}
GSA	10	211.25	308.00	333.00
	50	359.75	428.00	436.75
	90	475.75	496.25	496.25
PSA	10	330.75	353.25	372.75
	50	280.00	326.75	302.00
	90	181.75	191.50	178.75
PEA	10	16.75	15.25	27.50
	50	14.00	13.25	18.50
	90	25.00	27.50	14.75
PEAS	10	8.50	11.25	14.00
	50	6.00	7.25	6.50
	90	4.75	5.25	5.00
PEGAS	10	14.50	16.75	20.75
	50	6.50	10.25	8.25
	90	4.75	5.00	5.00

Table 13

Average number of search directions computed for indefinite problems

		κ		
	% bnd.	10^3–10^5	10^8–10^{10}	10^{13}–10^{15}
GSA	10	307.50	317.50	320.50
	50	185.75	174.75	168.00
	90	43.00	37.75	36.50
PSA	10	171.50	158.00	148.50
	50	81.75	72.25	67.25
	90	22.00	20.00	18.25
PEA	10	10.25	9.25	9.25
	50	12.75	11.75	11.25
	90	7.25	8.00	6.25
PEAS	10	10.25	9.00	10.25
	50	14.50	13.00	11.00
	90	12.50	12.50	11.25
PEGAS	10	10.25	9.00	34.00
	50	39.25	24.25	11.00
	90	13.00	12.75	11.25

Table 14

Average percent of variables bound at the solution for indefinite problems

		κ		
	% neg. ev.	10^3–10^5	10^8–10^{10}	10^{13}–10^{15}
GSA	10	73.43	79.75	81.45
	50	92.65	94.85	95.00
	90	99.45	99.65	99.65
PSA	10	73.43	79.75	81.45
	50	92.70	94.85	95.10
	90	99.45	99.65	99.65
PEA	10	73.53	79.80	81.55
	50	92.25	94.85	95.00
	90	99.50	99.70	99.55
PEAS	10	73.73	79.80	81.40
	50	92.50	94.85	95.25
	90	99.50	99.75	99.60
PEGAS	10	73.73	79.80	81.40
	50	92.50	94.85	95.25
	90	99.50	99.75	99.60

Table 15
Average number of variables freed for positive definite problems

		κ		
	% bnd.	10^3-10^5	10^8-10^{10}	10^{13}-10^{15}
GSA	10	89.00	141.75	156.00
	50	62.50	101.00	105.25
	90	17.50	33.00	33.75
PSA	10	325.00	345.25	363.25
	50	275.50	320.75	292.75
	90	177.25	185.00	170.25
PEA	10	329.50	363.75	390.00
	50	319.25	399.75	354.00
	90	213.50	227.25	202.50
PEAS	10	487.25	637.50	708.50
	50	433.00	544.00	553.75
	90	264.75	324.50	337.00
PEGAS	10	486.75	637.25	708.50
	50	433.00	544.00	553.75
	90	264.75	324.50	337.00

Table 16
Average number of variables freed for indefinite problems

		κ		
	% neg. ev.	10^3-10^5	10^8-10^{10}	10^{13}-10^{15}
GSA	10	49.25	48.75	39.50
	50	44.25	39.00	34.00
	90	18.00	17.75	15.75
PSA	10	164.25	150.25	142.25
	50	70.50	60.75	57.25
	90	18.50	18.00	16.00
PEA	10	27.75	20.75	20.50
	50	52.25	34.75	34.25
	90	13.00	11.75	8.75
PEAS	10	6.25	6.75	38.50
	50	33.50	14.75	1.50
	90	0.75	1.25	0.25
PEGAS	10	6.25	6.75	32.75
	50	29.50	13.25	1.50
	90	0.75	1.25	0.25

References

[1] J.L. Barlow, "The use of three different rank detection methods in the solution of sparse weighted and equality constrained least squares problems," Technical Report CS-88-18, The Pennsylvania State University (University Park, PA, 1988).

[2] D.P. Bertsekas, "Projected Newton methods for optimization problems with simple constraints," *SIAM Journal on Control and Optimization* 20 (1982) 221-246.

[3] A. Björck, "A direct method for sparse least squares problems with lower and upper bounds," *Numerische Mathematik* 54 (1988) 19-32.

[4] P.H. Calamai and J.J. Moré, "Projected gradient methods for linearly constrained problems," *Mathematical Programming* 39 (1987) 93-116.

[5] T.F. Coleman, "A chordal preconditioner for large scale optimization," *Mathematical Programming* 40 (1988) 265-287.

[6] A.R. Conn, N.I.M. Gould and Ph.L. Toint, "Global convergence of a class of trust region algorithms for optimization with simple bounds," *SIAM Journal on Numerical Analysis* 25 (1988) 433-460.

[7] A.R. Conn, N.I.M. Gould and Ph.L. Toint, "Testing a class of methods for solving minimization problems with simple bounds on the variables," *Mathematics of Computation* 50 (1988) 399-430.

[8] R.S. Dembo and U. Tulowitzki, "On the minimization of quadratic functions subject to box constraints," Technical Report B 71, Yale University (New Haven, CT, 1983).

[9] J.J. Dongarra, J.R. Bunch, C.B. Moler and G.W. Stewart, *LINPACK Users' Guide* (SIAM, Philadelphia, PA, 1979).

[10] G.C. Everstine, "A comparison of three resequencing algorithms for the reduction of matrix profile and wave front", *International Journal for Numerical Methods in Engineering* 14 (1979) 837-853.

[11] R. Fletcher, "A general quadratic programming algorithm," *Journal of the Institute of Mathematics and its Applications* 7 (1971) 76-91.

[12] R. Fletcher and M.P. Jackson, "Minimization of a quadratic function of many variables subject only to lower and upper bounds," *Journal of the Institute of Mathematics and Its Applications* 14 (1974) 159-174.

[13] A. George and J.W.H. Liu, *Computer Solution of Large Sparse Positive Definite Systems* (Prentice-Hall, Englewood Cliffs, NY, 1981).

[14] J.R. Gilbert and T. Peierls, "Sparse partial pivoting in time proportional to arithmetic operations," *SIAM Journal on Scientific and Statistical Computing* 9 (1988) 862-874.

[15] P.E. Gill and W. Murray, "Numerically stable methods for quadratic programming, *Mathematical Programming* 14 (1978) 349-372.

[16] P.E. Gill, W. Murray and M.H. Wright, *Practical Optimization* (Academic Press, New York, 1981).

[17] G.H. Golub and C.F. Van Loan, *Matrix Computations* (The Johns Hopkins University Press, Baltimore, MD, 1983).

[18] N.J. Higham, "Fortran codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation," Technical Report Numerical Analysis Report No. 135, University of Manchester (Manchester, UK, 1987).

[19] J.J. Moré, "Numerical solution of bound constrained problems," Technical Report 96, Argonne National Laboratory (Argonne, IL, 1987).

[20] J.J. Moré and D.C. Sorensen, "Computing a trust region step," *SIAM Journal on Scientific and Statistical Computing* (1983) 553-572.

[21] J.J. Moré and G. Toraldo, "Algorithms for bound constrained quadratic programming problems," Technical Report 117, Argonne National Laboratory (Argonne, IL, 1988).

[22] K.G. Murty and S.N. Kabadi, "Some NP-complete problems in quadratic and nonlinear programming," *Mathematical Programming* 39 (1987) 117-129.

[23] S. Parter, "The use of linear graphs in Gauss elimination," *SIAM Review* 3 (1961) 119-130.

[24] D.J. Rose, R.E. Tarjan and G.S. Leuker, "Algorithmic aspects of vertex elimination on graphs," *SIAM Journal on Computing* 5 (1976) 266-283.

[25] M. Yannakakis, "Computing the minimum fill-in is NP-complete," *SIAM Journal on algebraic and Discrete Methods* 2 (1981) 77-79.

[26] Y. Ye and E. Tse, "A polynomial-time algorithm for convex quadratic programming," Technical Report, Stanford University (Stanford, CA, 1986).