Nonlinear Optimization Using the Generalized Reduced Gradient Method

Article · October 1973

DOI: 10.1051/ro/197408V300731

Leon S. Lasdon
University of Texas at Austin
136 PUBLICATIONS **11,246** CITATIONS

ORSA Journal on Computing
Vol. 4, No. 1, Winter 1992

0899-1499/92/0401-0002 $01.25
© 1992 Operations Research Society of America

Solving Large Sparse Nonlinear Programs Using GRG

STUART SMITH *Krannert School of Management, Purdue University, West Lafayette, IN 47907, INTERNET: shsmith@midas.mgmt.purdue.edu*

LEON LASDON *MSIS Department, School of Business Administration, University of Texas, Austin, TX 78712, BITNET: lasdon@utxvm*

(Received: September 1990; final revision received: February 1991; accepted: February 1991)

We describe a feasibility-retaining GRG algorithm for large sparse nonlinear programs of general form. Its FORTRAN implementation, LSGRG, is enhanced by heuristics which aid in basis selection, combatting degeneracy, dynamic tolerance adjustment, and predicting Newton failures. Key roles are also played by efficient procedures for basis inversion and by both pure and limited memory BFGS methods for computing the search direction. The design goal for LSGRG is maximum reliability with at least acceptable speed. Extensive computational tests on both FORTRAN and GAMS models indicate that LSGRG is promising in this regard. Comparisons are presented with GRG2, a sparsity oriented SQP code, and MINOS, indicating that LSGRG is a useful complement to MINOS in a multi-solver NLP environment.

Generalized Reduced Gradient (GRG) algorithms, introduced by Abadie and Carpentier[1] are widely used to solve small to medium size problems, mainly through the FORTRAN codes GRG2[13] and OPT[22] and the interpretive PC-based system GINO[14]. One of the most widely used programs for large sparse NLPs, MINOS,[20] uses a GRG algorithm to solve its linearly constrained subproblems. Drud[9] described a GRG code CONOPT, for directly solving sparse nonlinearly constrained NLPs.

Despite these achievements, many questions remain regarding the tactical and implementation issues involved in a large scale GRG code. In addition, little is known regarding the speed and reliability of careful large scale GRG implementations relative to other methods. We initiated the research described here because of the conviction that a good sparsity oriented GRG code would be very reliable and acceptably fast on a wide range of problems. Results presented here, including comparisons with MINOS on problems from the GAMS[2] library, provide a partial confirmation of these expectations. They indicate that modeling systems like GAMS, AMPL[10] and LINGO[5] can benefit greatly by providing links to both feasible path algorithms like GRG and infeasible path procedures like MINOS and SQP.[17] If one does not work, the user can try another, or even alternate between two or more in various orders. We believe that, by including algorithms which approach the solution of NLPs in very different ways, the chances that at least one will succeed are substantially increased.

The system described here, called LSGRG, draws upon ideas from both GRG2 and CONOPT. The design philosophy was to stay as close to the algorithmic structure of GRG2 as possible, deviating when dictated by size or sparsity, or when tests indicated that increased speed could be achieved without a significant decrease in reliability. LSGRG has the following features:

1. Initial feasibility seeking by an LP-like phase I algorithm, followed by a phase II which retains feasibility at each step of the linesearch.
2. Selection among the many possible bases at each active set change using a procedure which adapts itself to the problem at hand by dynamic adjustment of the pivot threshold.
3. Basis factorization and updating using Schrage's implementation of Suhl and Aittoniemi's sparse matrix inversion procedure.
4. Condition number estimators for the basis and approximate Hessian.
5. Use of a Tabu list[11] and relaxation of bounds to combat degeneracy.
6. Computation of search directions for the superbasic variables using the BFGS method, switching to Liu and Nocedals' limited memory method[15] when the number of superbasic variables exceeds a threshold.
7. A linesearch using quadratic interpolation which allows many superbasics to hit bounds.
8. Retention of feasibility through Newton's method, predicting failure of the Newton iterations by a device due to Drud.[9]
9. An interface with GAMS, with modifications to exploit GAMS' ability to provide any desired subset of constraint values and its rapid evaluation of nonconstant derivatives.

Subject classifications: Programming: nonlinear. Computers/computer science: software.
Other key words: A FORTRAN implementation for large sparse nonlinear programs.

The paper is organized as follows. Section 1 contains a problem statement and the major steps of the algorithm. Section 2 deals with basis selection and factorization, while Section 3 describes anti-degeneracy procedures. Methods for choosing the search direction and adding superbasic variables are in Section 4, followed by a section describing the linesearch. Section 6 provides computational results on problems coded in FORTRAN, and includes comparisons with GRG2 and the SQP code described in [17]. The next section includes a brief description of the GAMS interface, and a comparison with MINOS on the large nonlinear programs in the current GAMS library, plus some large models from the oil industry. Section 8 contains conclusions and suggestions for future work.

We assume throughout that the reader is familiar with the basic concepts of GRG algorithms as described in [1], [13], or by Luenberger in [16].

1. Problem and Algorithm Statement

We consider nonlinear programs of the following general form

minimize $f(x)$

subject to

$$g(x) = b \tag{1}$$

and

$$\ell \leqslant x \leqslant u \tag{2}$$

where x is a vector of n decision variables, g is a vector of m constraint functions, and the objective f and constraints g_i are differentiable at all points x satisfying the bounds (2). The right hand side vector b and bound vectors ℓ, u are specified constants. Slack variables used to convert inequality constraints into the equalities in (1) are contained in x. We are primarily concerned with problems where there are hundreds or thousands of constraints and variables, and where the Jacobian matrix of the constraints, J, is sparse.

For such problems, the major steps of our LSGRG algorithm are:

1. Initialization: the following are user-provided, with default values shown:

   epstop = convergence tolerance (default = 10^{-4})

   nstop = consecutive iteration counter (default = 3)

   epfeas = feasibilty tolerance (default = 10^{-4}).

   Set x to the initial x vector, presumed feasible for ease of exposition.
2. Compute the values and gradients of all problem functions at x.
3. Choose a basis, B, from the columns of the Jacobian matrix, J, so that J and x are partitioned into basic and "not basic" portions as:

$$J = [\partial g/\partial x] = [B, N] \tag{3}$$

$$x = (x_b, x_{nb}) \tag{4}$$

   compute lower and upper triangular factors for B so that $B = LU$.
4. Solve the equations

$$B^T\pi = \partial f/\partial x_b \tag{5}$$

   for the Lagrange multipliers π and compute the reduced gradient of the objective as

$$rg = \partial f/\partial x_{nb} - \pi^T N \tag{6}$$

5. If
   (a) The Kuhn-Tucker conditions are satisfied to within epstop, or
   (b) The last nstop fractional objective changes are less than epstop

   stop
6. Divide the "not basic" variables x_{nb} into superbasic variables, x_s, which are to be changed during the current iteration, and nonbasic variables, x_n, which remain at their bounds. Compute a search direction, d_s, for the superbasic variables using either the BFGS quasi-Newton or Limited Memory algorithms.
7. Perform a line search along d_s, retaining feasibility to within epfeas at each trial point by solving for the basic variables using Newtons method.
8. Set x to the solution resulting from the line search and go to Step 2.

If the initial point violates the bounds (2) it is forced to satisfy them by projecting it onto these bounds. If the resulting point violates any of the general constraints (1), an LP-like phase I is initiated, during which the objective f is replaced by the sum of constraint violations.

To exploit sparsity, the nonzero elements of the Jacobian, J, are stored in packed form by column, just like the coefficient matrix in a sparse LP code. Row indices and start of column pointers are also needed. The subroutines which factorize B in Step 3 produce sparse factors. This Jacobian and basis factor data is utilized in solving linear equations in steps 4 (BTRAN) and 7 (FTRAN at each Newton iteration), and in the pricing calculation of Equation 6.

2. Basis Selection and Factorization

At any feasible point which is not a vertex, there will be more than m variables not at bounds. Hence there are usually many possible choices for the set of basic variables. A good choice strikes a balance between two often conflicting criteria:

1. Basic variables should be far from their bounds, allowing the linesearch to improve the objective as much as

possible without being terminated by basic variables hitting bounds.
2. The basis matrix should be well conditioned. This improves the accuracy of linear equation solving in (5) and in the Newton iteration within Step 7 of the GRG algorithm. A small basis condition number also promotes rapid convergence of the Newton iterations and influences the conditioning of the reduced problem (see [16], section 11.7).

The basis selection procedure used in GRG2 is based on complete pivoting. It requires computing all elements in the unpivoted rows and columns of the updated Jacobian tableau, and is obviously unsuitable for large problems. The logical next step is a procedure based on partial pivoting. This is called "Complete Search" mode. Its steps are:

Complete search mode is used to select an initial basis if one is not provided, and as a recourse procedure when the algorithm stalls or has made too many consecutive degenerate iterations. It is very time-consuming, however, and we use a faster method whenever possible. As a first step, we try to refactorize the current basis (its elements may have changed if any constraints are nonlinear). A new set of LU factors is produced. The refactorization can be done by a fast procedure which uses the previous pivot positions, although this has not yet been implemented. If no basic variables hit bounds in the last linesearch, and the refactorization has been successful, we are finished. If it is not successful, a complete search mode is initiated.

When one or more basic variables have hit bounds, replacements for these columns in B must be selected. If R is the set of rows in which pivots are required, we simply apply Steps 2 thru 6 of complete search mode to

PROCEDURE: SEARCH MODE

INPUT:

J	—The current Jacobian matrix
x	—Current values of all variables
eppiv	—Absolute pivot tolerance
THRESH	—Pivot threshold

OUTPUT:

X_b	—Index set of basic variables
B	—Basis matrix, along with its LU factors

Step 1. The rows of the Jacobian, J, are examined in natural order. Assume that $k - 1$ basic columns have been selected from J.
Step 2. Let B_{k-1} be the matrix having these basic vectors as its first $k - 1$ columns, with the remainder being the last $m - k + 1$ unit vectors. Compute row k of B_{k-1}^{-1}, b_k, by solving the linear system

$$b_k B_{k-1} = e_k$$

where e_k is the $(k)^{\text{th}}$ unit vector.
Step 3. Determine a set of candidate variables C_k. Initially this is the set of unpivoted variables not at bounds.
Step 4. Compute the elements of the updated tableau in row k and in columns i in C_k, $\bar{a}_{ik}$, as the scalar product of b_k and column i of the Jacobian. Let $\max_k$ be the maximum absolute value of these elements. If $\max_k <$ eppiv (pivot tolerance, default value $= 10^{-6}$), set C_k to the set of unpivoted variables at bounds and compute the corresponding $\bar{a}_{ik}$
Step 5. Choose a pivot column from C_k satisfying

$$|\bar{a}_{ik}| > \text{THRESH}*\max_k$$

whose associated variable is farthest from its nearest bound (If C_k contains only columns at bounds, a column with $|\bar{a}_{ik}| = \max_k$ is selected). THRESH is called the pivot threshold (default $= 0.1$).
Step 6. In B_{k-1}, replace e_k by the selected column, yielding B_k, and update the LU factors of B_{k-1} so that they become the factors of B_k.

each row in R, updating the LU factors each time. This is called partial search mode.

Within partial search mode, a heuristic designed to restrict the set R has proven to be quite effective on large problems. Tests on large models revealed that basic variables at bounds often remained basic for several consecutive iterations. This occurs when the tangent vector component for such variables is close to zero, and thus these variables change only slightly as the line search progresses. To take advantage of this observation, we keep track of the number of consecutive times a variable at a bound has remained basic. If any basic variable at a bound has a count greater than 2, this variable will be kept basic by not adding the corresponding row to the set R during partial search mode. Of course, such variables will leave the basis during a degenerate step if they are blocking the move in the superbasics.

Pivot Threshold Adjustment

The pivot threshold parameter THRESH can have a significant impact on the overall performance of the algorithm. It is used to strike a balance between basis conditioning and distance of the basic variables from bounds. Values closer to 0 allow greater flexibility in choosing variables farther from bounds, but the resulting matrix may have a higher condition number. This can mean an increase in Newton failures, causing the algorithm to take small steps. In contrast, values nearer to 1 lead to better conditioned bases, but many basic variables may be close to or at bounds, possibly resulting in degeneracy. This effect is exhibited both between problems and during the solution of individual problems. Performance often can be enhanced by allowing different THRESH settings as the algorithm progresses.

To allow for varying conditions during a run, THRESH is dynamically adjusted by the algorithm as conditioning or degeneracy difficulties arise. Initially a value of 0.1 is used. As the solution proceeds, the parameter is allowed to increase or decrease among the values (0.005, 0.01, 0.05, 0.1, 0.5, 0.8, 0.9, 0.95) in a systematic way. A record is kept of the last 20 Newton calls since the last THRESH increase, and the last 20 bases formed since the last decrease. Before the start of each basis selection phase, an adjustment, if necessary, is determined by applying the following rules:

1. If more than 1/3 of the last 20 Newton calls since the previous increase have resulted in a failure, then THRESH is increased.
2. Otherwise, if more than 1/2 of the last 20 bases since the previous decrease have been degenerate, then THRESH is decreased.

Empirical testing on several large problems has shown that this heuristic permits a gradual fine tuning of the basis to compensate for changes in problem characteristics without requiring user intervention. It has never prevented the solution of an otherwise solvable problem, and has resulted in noticeable improvements in several instances.

Condition Number Estimate

A novel feature of LSGRG is the computation of a lower bound on the basis condition number. The approach, developed by Cline et al.[3] estimates the condition number of a matrix A by

$$\|A\|_1 \frac{\|y\|_1}{\|x\|_1} \tag{7}$$

where y and x are two vectors determined so that $\|y\|_1 / \|x\|_1 \doteq \|A^{-1}\|_1$. This is accomplished by solving the two linear systems of equations

$$A^T x = b \tag{8}$$

and

$$Ay = x \tag{9}$$

where b is a vector with components ± 1 chosen so that x will be a near maximizer of $\|A^{-1}x\|_1 / \|x\|_1$.

The actual calculations proceed as follows. Using the factorization $A = LU$, the solution to equation (8) is performed in two steps. First, the vector z is determined by solving $U^T z = b$ using forward substitution where the elements of b are chosen during the solution steps to maximize the magnitude of the components $z_1, \ldots, z_m$ of z in succession. Then x is computed by solving $L^T x = z$ using back-substitution. Next y is determined by solving equation (9). The estimate is then obtained by computing $\|A\|_1$, $\|x\|_1$, and $\|y\|_1$, and applying formula (7). The reader is referred to [3] for further details, as well as a theoretical justification for the particular choice of the vector b.

This method has been used in several software packages, the most notable of which is LINPACK (Dongarra et al.[8]), a widely used collection of FORTRAN subroutines for solving linear systems. Since only two sets of linear equations must be solved, this approach produces reliable estimates with minimal computational effort.

This condition umber estimate is used in LSGRG to monitor the conditioning of the basis matrices. After each basis is factorized, the above estimate is calculated, and compared to a user supplied parameter, CONDMX (default value 10^8). If its value exceeds CONDMX, the current basis is deemed ill-conditioned, and depending on the type of basis selection mode which was used, one of two actions is taken. If the ill-conditioned matrix was produced by a partial search mode or reinversion mode, then a complete search mode is initiated to select a new basis. If the suspect matrix was produced by a complete search mode, a warning message is printed, and the algorithm proceeds with the parameter THRESH increased as outlined previously.

3. Degeneracy

Despite efforts to avoid it, (described in the previous section), bases formed at the start of an LSGRG iteration may contain variables at or very near bounds. After the corresponding set of superbasic variables has been determined and the search direction, d_s, is computed, the tangent vector, v, is calculated by solving

$$Bv = -Sd_s$$

where S is the submatrix of J corresponding to the superbasic variables. The components of v are the directional derivatives of the basic variables as the superbasics vary along the ray $x_s + ad_s$, where x_s contains the current values of the superbasics and the scalar a determines the step size. Treating v as the transformed column in LP, a ratio test determines the largest a value such that all basic variable bound violations are less than a (relative) bound tolerance (default value = 10^{-6}), along with a basic variable achieving this violation, whose index is denoted by r. If this step is judged to be too small, procedure DEGEN is called. It determines a column to enter the basis, replacing column r. Otherwise, a line-search occurs.

Preferably, the entering variable will be away from its bounds. If not, then care must be taken to avoid long sequences of degenerate steps, and in the extreme case, cycling. While degeneracy has not proven to be a serious difficulty for small problems, large sparse problems frequently require numerous degenerate iterations. It is important that the time spent executing such iterations be as small as possible and that the logic used in procedure DEGEN minimizes their occurrence. The logic used here is a modification of Steps 2–6 of the search mode algorithm of Section 2. The major differences are that the selection criteria are biased more toward distance from bounds than conditioning, and that the candidate list is restricted by keeping a "Tabu" list (Glover[11]) of variables which are inadmissible to enter the basis. The algorithm is as follows:

PROCEDURE: DEGEN

INPUT:

X_b	—The current set of basic variables, ordered by rows.
B	—The corresponding basis matrix.
r	—Row index of the basic variable to be replaced.
eppiv	—Absolute pivot tolerance.
TABU	—Circular list of (row, var) pairs of previous degenerate moves.

OUTPUT:

X_b	—New basis with the rth variable replaced.
B	—Updated basis matrix, and an updated LU factorization.
TABU	—Updated to contain newest degenerate step.

Step 1. Compute the rth row of B^{-1}. Compute the rth row of B^{-1}, b_r, by solving the linear equations:

$$b_r B = e_r$$

where e_r is the unit row vector with a 1 in the rth position.
Step 2. Construct a candidate list. Construct a set of candidate variables, C_r. Initially, this set contains all superbasic variables, j, such that $(r, j) \notin$ TABU. If no pivot can be chosen using the test in Step 3 below, then C_r is redefined to be the set of nonbasic variables, j, such that $(r, j) \notin$ TABU.
Step 3. Compute the updated pivot row. Calculate the elements $\bar{a}_{r,j}$ of the updated pivot row in row r and column $j \in C_r$ as the dot product of b_r and column j of the Jacobian. Set

$$\max_r = \max_j |\bar{a}_{r,j}|$$

If $\max_r \leqslant$ eppiv then return to STEP 2 and change C_r.
Step 4. Select a pivot column. From all $j \in C_r$ satisfying

$$|\bar{a}_{r,j}| \geqslant \max\{\text{eppiv}*\max_r, \text{eppiv}\}$$

select the new pivot column (basic variable) p, to be the variable which is farthest from its nearest bound. If C_r contains only variables at bounds, select p such that $\bar{a}_{r,p} = \max_r$.
Step 5. Update the LU factors. Let k be the index of the old rth basic variable. Replace k with p. Change B to reflect this change, and update the LU factors so that they represent this new basis matrix.
Step 6. Update the tabu list. Add the pair (r, k) to the circular list TABU.

Two properties of DEGEN should be emphasized. First, in an attempt to find an entering variable which is away from its bound, a smaller threshold tolerance equal to eppiv is used in Step 4. This replaces the parameter THRESH employed in algorithm SEARCH MODE. Since eppiv is the minimum pivot tolerance allowed, this represents the extreme case of disregarding stability in order to overcome degeneracy. In addition, the TABU list has been added in an effort to prevent cycling during consecutive degenerate steps. This list contains pairs of the form (row, var), meaning that variable "var" was previously basic in row "row" and was replaced by a degenerate pivot. When constructing the candidate list in Step 2, for a particular row r, all variables, j, for which the pair (r, j) is currently in TABU are excluded. TABU is stored as a circularly linked list with a maximum of LENTAB (default value 25) entries. It is initially empty, and is reset to the null set whenever a non-degenerate basis is selected. Following every degenerate pivot, a new (row, var) pair is added in Step 6. If TABU is full then the oldest remaining entry is replaced with this new pair.

While this strategy has proven to be quite successful in deterring degeneracy, it is still possible for long sequences of degenerate steps to occur. If the number of consecutive degenerate steps exceeds the user supplied limit MAXDEG (default of 50), then a complete search mode is initiated and the degenerate count is reset to zero. If the subsequent MAXDEG consecutive steps are still degenerate, the variable bounds are expanded so that a slightly relaxed problem is formed (we thank Bruce Bixby for suggesting this strategy). For each variable, j, the upper and lower bounds u_j and l_j are perturbed outward as follows:

$$u_j \leftarrow u_j + \text{epdeg}(1 + |u_j|)$$

$$l_j \leftarrow l_j - \text{epdeg}(1 + |l_j|)$$

where epdeg is a user parameter with a default value of 10^{-4}. If the algorithm is still degenerate for MAXDEG steps, epdeg is multiplied by 10 and the process repeated. This continues until either a non-degenerate step is made, or until epdeg $\geqslant 10^{-2}$. If the latter happens, a message is printed that the problem is cycling and the algorithm terminates.

If a positive step is made, the algorithm proceeds to solve the relaxed problem until some termination criterion is satisfied. If this final point satisfies the original (unperturbed) bounds, it is accepted as the final solution. Otherwise, all variables which violate a bound are projected onto their nearest bound, and the algorithm is restarted with this point as the new initial point and all variable bounds reset to their original values.

This relaxation is intended to be used only as a last resort when all previous remedies have failed. The goal is to perturb the problem just enough to escape degeneracy, so that the final solution provides a reasonable initial point from which the original problem can be solved quickly. Unfortunately, the relaxed problem can lead to domain errors in the constraints. If some of the variable bounds are used to prevent domain violations such as division by zero or computing square roots of negative numbers, then the perturbed bounds may admit undefined function evaluations. Currently, the success of this strategy is unknown, because it has yet to be invoked on any test model. An expanded database of models which cause the logic to be executed is needed to fully evaluate its reliability and effectiveness.

The "maximal basis" idea proposed by Dembo and Klincewicz[7] may also reduce the effects of degeneracy. This has yet to be explored.

4. Search Direction and Superbasis Management

LSGRG computes search directions for the superbasic variables using the BFGS Quasi-Newton method whenever the number of superbasics is less than a user-defined limit (default value 50), dynamically switching to Liu and Nocedals'[15] Limited Memory procedure when this limit is exceeded. As in GRG2, the BFGS implementation stores and updates the Cholesky factors of the approximate Hessian H, as described in [19], when the set of basic variables is unchanged, resetting H to the identity otherwise. The only modification is that, when a variable is added to the superbasic set, H is extended by adding a new diagonal element whose value is the geometric mean of the other diagonal elements. This idea is due to Drud.[9] The memory length in the Limited Memory procedure has a default value of 4, and the method is restarted whenever the set of basic variables changes or when a variable is deleted from the superbasic set. When a variable is added to the superbasis, the method is extended by appending zeros to all past position and gradient change vectors but the last, and using actual changes during the previous linesearch.

Changes in the Superbasic Set

How variables are added to and subtracted from the superbasic set strongly affects algorithm performance. Too frequent changes can cause jamming,[16] while overly conservative strategies can cause slow convergence. We have chosen a fairly aggressive strategy which has elements from both CONOPT and GRG2. Many superbasics may hit bounds during a linesearch (see Section 6 for details), and several superbasics may be added at each iteration. The logic is as follows:

```
PROCEDURE: SUPERBASICS
INPUT:
  $X_s$ —Current superbasic set.
  $X_n$ —Current nonbasic set.
  $d_s$ —Current search direction.
  $gr_s$ —Reduced gradients of superbasic variables.
  $gr_n$ —Reduced gradients of nonbasic variables.
OUTPUT:
  $X_s$ —Updated superbasic set.
  $X_n$ —Updated nonbasic set.
  $d_s$ —Updated search direction.
Step 1. Delete all variables at bounds from the superbasis and add to the nonbasic
set. If using the BFGS method, modify the approximate Hessian accordingly.
Step 2. Compute $d_s$, the search direction for the superbasis using the BFGS or
Limited Memory algorithm. If using the BFGS update, compute RMEAN as the
geometric mean of the diagonal Hessian elements, otherwise set RMEAN to 1.
Step 3. Compute SLOPE = $d_s^T gr_s$ the slope of the objective along the direction
$d_s$.
Step 4. Find $k_{\max} \epsilon X_n$, the nonbasic variable with the largest favorable reduced
gradient component, $gr_{\max}$.
  Step 5. If $gr_{\max}$ satisfies the condition:
      $\frac{gr_{\max}^2 \text{LKSAME}^2}{\text{RMEAN}} \geqslant \frac{|\text{SLOPE}|}{4}$
then add $k_{\max}$ to $X_s$ with direction component $-gr_{\max}$, and go to Step 2.
```

The test in Step 5 can be justified as follows: The left hand side is the increase in the slope caused by allowing this variable to become superbasic, weighted by LKSAME^2, where LKSAME is the number of consecutive times this nonbasic variable has been a candidate for the superbasis, i.e. had a favorable reduced gradient. This weight is added to force variables which continually have favorable reduced gradients to be included in the superbasis. If this weighted increase is at least a quarter of the current slope, then the variable is allowed to become superbasic. Since $|\text{SLOPE}|$ is increasing, $gr_{\max}$ is decreasing, and X_n is finite, eventually the test is failed and the process terminates.

5. The Line Search

The goal of the line search is to approximate the first local minimum of the reduced objective function along the search direction. Most of the logic is as in GRG2, and detailed descriptions will not be repeated here. Steps are selected by a halving and doubling procedure which attempts to bracket the minimum, followed by a quadratic interpolation. At each step we regain feasibility by a Pseudo-Newton method, which uses the basis factors evaluated at the start of the search. Initial values for Newtons method are found using linear and quadratic extrapolation. Basic variable bound violations are allowed during the Newton iterations. If such bound violations occur after Newton converges, a "backup phase" is initiated which finds a reduced step size at which all basic variable bounds are satisfied. If this point has an improved objective value, the search terminates.

As noted by Drud,[9] allowing basic variables to violate bounds may cause domain violations for functions involving logs, fractional powers, etc. Future plans include an option which maintains all variables within their bounds.

Some modifications to the GRG2 logic have been adopted. Let ITLIM be the Newton iteration limit (default value 10). Assume that $k \geqslant 1$ Newton iterations have occurred, let rmax_k be the maximum absolute equation residual at iteration k, and define

$$p = (\text{ITLIM} - k)/2$$

Then, if

$$\text{rmax}_k(\text{rmax}_k/\text{rmax}_{k-1})^p > \text{epnewt}$$

the Newton iterations are terminated and a failure flag is set. If an improved point has been found the linesearch terminates, otherwise the step size is halved. This test, due to Drud,[9] uses the fact that the sequence $\{\text{rmax}_k\}$ converges geometrically, estimating the geometric ratio from the last two elements of the sequence. Experience shows that if a failure is not predicted after the first iteration, the Pseudo-Newton method usually converges.

In GRG2, the step taken is limited to the largest step which maintains all superbasics within their bounds. Hence, usually at most one superbasic will hit a bound during a linesearch. In papers studying NLPs with only simple bound constraints. More[18] and others propose procedures which allow many variables to hit bounds in a single search by projecting variables violating bounds onto those bounds. We have applied this idea to the superbasic variables as our default strategy, although the user may revert to the GRG2 strategy if desired.

Table I compares the GRG2 strategy (FIRST) with the projection strategy (MULT) on the two multi-period models used in [18] to test MINOS. For T time periods, the optimal control model (OPT$\langle T \rangle$ in table 1) has $3T + 2$ variables ($2T + 2$ nonlinear) and $2T$ constraints (T nonlinear). The economic growth model (ECON$\langle T \rangle$) has $3T$ variables ($2T$ nonlinear) and $2T$ constraints (T nonlinear). User coded partial derivatives were used in all runs.

In Table I, the MULT times are always less than those for FIRST, with total time for all problems reduced by 23.2%. The effect increases, somewhat erratically, with problem size. The number of line searches is reduced by MULT in all but two cases, and the total is reduced by 13.5%. However, the average number of Newton iterations per Newton call (Newt Avg column) is higher for MULT in all instances. As the line search step size, a, is increased, more Newton iterations are required, because all partial derivatives are evaluated at $a = 0$. In addition, after projection of one or more superbasics, the initial estimates for the basic variables are usually not as good (if the constraints are linear and MULT is used, these estimates are no longer exact).

6. Results on FORTRAN Models

All computational tests in this paper were performed on an IBM 3081-K mainframe at the University of Texas at Austin, running under the VM/SP CMS operating system. LSGRG routines are written in standard FORTRAN 77, and were compiled with the IBM FORTVS compiler using optimization level 2. All execution times are in virtual CPU seconds, and were obtained by the FORTRAN system timer routine CPUTIME.

Tests on Small to Medium Size Problems

This set of problems contains 10 of the more difficult problems from the widely used Himmelblau[12] and Dembo[6] test sets. Problem sizes are shown in Table II.

Table III gives solution statistics comparing GRG2 and LSGRG on these problems. Both codes used finite difference derivatives, and the same default values for all parameters and options, with one exception—LSGRG did not project superbasics on bounds, for compatibility with GRG2. Also, LSGRG uses relative feasibility tolerances while GRG2 has absolute tolerances. In the table, function calls include those used to approximate first derivatives.

The most notable feature of these results is that LSGRG solved Dembo problem 7 and H6, while GRG2 did not. In D7, GRG2 stopped after a string of 15 consecutive degenerate iterations, while the anti-degeneracy devices in LSGRG allowed it to find the optimal solution (D7 is quite difficult and sensitive to algorithm tolerances—GRG2 solves it starting with epfeas-10^{-4}, then switching to 10^{-6}). In H6, a minimization problem, the final objective values for LSGRG and

Table I
Comparison of Line Search Strategies on Multiperiod Models

	Time (secs)		Line Searches		Newt Avg	
Problem	FIRST	MULT	FIRST	MULT	FIRST	MULT
OPT50	5.75	4.55	151	105	0.33	0.45
OPT100	35.98	30.90	249	296	0.28	0.48
OPT200	138.61	138.44	513	517	0.31	0.41
OPT300	409.20	256.70	830	735	0.18	0.34
ECON50	5.41	4.23	151	104	0.35	0.89
ECON100	27.62	24.71	333	272	0.56	0.74
ECON200	81.10	73.21	347	253	0.63	1.00
ECON300	154.28	126.22	347	243	0.63	0.98
Totals	857.94	658.96	2921	2525		

Table II
Himmelblau and Dembo Test Problems

Problem Name	No. of Variables	No. of Constraints[a]
H4	10	3
H6	45	16
H12	5	21
H18	15	5
H19	16	8
H20	24	20
H23	100	13
D6	13	13
D7	16	19
D8	7	4

[a]Does not include bounds on variables.

GRG2 were −1910.1 and −1837.5 respectively. Otherwise, all final objective values were essentially the same. Solution times are quite comparable, never more than one second apart. LSGRG has fewer function calls in six of the ten problems and a slight edge in total calls. The results suggest that many of the ideas introduced into LSGRG improve its performance on both small and large problems, and that the computational overhead caused by using sparse data structures with small problems is not significant.

Finite Difference versus Analytic Derivatives

Table IV compares finite difference and user-provided FORTRAN formulas for derivatives, using the two multiperiod problems described in Section 5.

For these large problems, where LSGRG requires thousands of function calls, the time spent computing finite difference derivatives dominates total run time, varying from 49.3% to 94.3% in ECON300. Since derivatives of the functions in these models are all simple expressions, the time spent computing analytic partials is never more than 3.9% of total time. The analytic option decreases all run times sharply, with the effect increasing strongly with problem size. The time for OPT300 is reduced by a factor of 3.87, and that for ECON300 by 14.4. Total time for all problems decreases by a factor of 6.3, and total function calls are reduced by a factor of 143.

In the majority of FORTRAN model applications, users do not wish to code and debug subroutines which compute derivatives. This limitation is removed by algebraic modeling systems like GAMS, which compute derivatives symbolically. Of course, the time required to evaluate interpreted expressions is longer than for com-

Table III
Comparison of LSGRG and GRG2

	Fcn Calls		Grad Calls		Line Sears		Time (secs)	
Problem	GRG2	LSGRG	GRG2	LSGRG	GRG2	LSGRG	GRG2	LSGRG
H4	227	334	21	24	18	20	0.22	0.31
H6	3753	2719	58	54	57	51	13.01	12.04
H12	78	120	6	10	5	8	0.15	0.23
H18	1005	851	35	32	33	29	0.83	0.86
H19	745	698	36	34	35	31	1.93	1.87
H20	1329	1072	36	34	38	32	2.13	1.64
H23	8540	7147	83	68	79	64	17.89	18.40
D6	771	2087	32	96	31	93	0.75	1.48
D7	860	1595	35	68	34	64	0.69[a]	1.51
D8	1229	725	61	46	60	43	0.90	0.76
Totals	18537	17348	403	466	390	435	38.50	39.11

[a]GRG2 stopped after 15 degenerate steps.

Table IV
Analytical vs. Numerical Derivatives on Multiperiod Models

	Time (secs)		Fcn Calls		Parsh Time (%)	
Problem	FIN	ANAL	FIN	ANAL	FIN	ANAL
OPT50	8.83	4.55	16594	330	49.3	1.9
OPT100	75.45	30.90	93489	1050	59.8	1.5
OPT200	434.00	138.44	316042	1755	68.6	1.2
OPT300	993.61	256.70	667481	2281	76.6	1.0
ECON50	19.18	4.23	16310	410	78.7	3.9
ECON100	179.96	24.71	83400	1200	86.6	3.4
ECON200	611.24	73.21	148609	2019	90.0	3.3
ECON300	1822.71	126.22	221450	1850	94.3	2.8
Totals	4144.98	658.96	1563375	10895		

Table V
Comparison of LSGRG with SQP

	Time (secs)		Fcn Calls		Grad Calls	
Problem	SQP	LSGRG	SQP	LSGRG	SQP	LSGRG
OPT100	8.9	30.90	8	1050	6	299
OPT200	70.4	138.44	7	1755	6	520
OPT300	256.2	256.70	7	2281	6	739
ECON100	19.0	24.71	37	1200	32	273
ECON200	140.1	73.21	38	2019	36	254
ECON300	191.9	126.22	47	1850	46	243
Totals	683.5	650.18	144	10155	132	2328

piled code. Nonetheless, these results motivated us to develop an interface to GAMS. Results using this interface are described shortly.

If the problem is sparse, groups of non-overlapping variables often exist, and these can be differenced simultaneously, Coleman et al.[4] This can sharply reduce computation time for finite difference derivatives. However, such procedures provide little benefit if one or more of the problem functions involve most of the variables. This is the case for the objective function in the OPT and ECON series of problems.

Comparisons with SQP

Here, LSGRG is compared with the sparsity-exploiting SQP code described in [17], which uses Liu and Nocedals' Limited Memory BFGS method (with memory length 3) to represent the approximate Hessian. Results for the three largest cases of the two multiperiod models used earlier are shown in Table V.

For these problems, the two codes require similar run times, despite the fact that SQP requires far fewer function and gradient evaluations. SQP is faster on the OPT series, but the effect decreases with problem size, and the times are almost equal to OPT300. LSGRG is significantly faster on the two largest instances of the more difficult ECON problems and it has a slight edge in total time. The time required by SQP to solve its quadratic subproblem dominates its execution time for these large problems, and this reduces or eliminates the advantage that it gains by requiring few function and gradient evaluations.

7. Comparisons with MINOS Using GAMS

Construction of the LSGRG/GAMS interface was motivated by two factors. One is the ease of use and increasing popularity of GAMS, plus the generous donation of support by GAMS Development Co. The second is the availability of a sizeable database of large NLPs written in GAMS. The difficulty of obtaining such problems has been perhaps the largest obstacle to testing and comparing NLP codes. GAMS provides a convenient and widely accepted format for creating and disseminating such problems.

Space does not permit a detailed description of the GAMS interface here. Interested readers are referred to [23]. However, to understand the computational results the following is important. Function values and derivatives are computed in GAMS using a list of "instructions," which are decoded and executed by the GAMS interpreter. These are arranged by row, and they include instructions for evaluating both the nonlinear part of each row and all nonconstant partial derivatives of the row. The constant Jacobian elements are stored separately. Hence, each nonlinear problem function is individually accessible, but all nonconstant derivatives *must* be computed when the function is evaluated. Since LSGRG requires only values of the active constraints during its Newton iterations (which is where most function evaluations occur), we replaced the usual function subroutine (which returned all constraint values), with one which computed only the constraints specified by an index set. During the Newton process, this index set was the set of active constraint indices. However, this advantages was offset by the simultaneous computation of all nonconstant derivatives of these constraints, which are not needed. The GAMS authors estimated that 80% of function evaluation time was spent evaluating derivatives. When these are actually needed, a separate subroutine is called. Hence, for LSGRG, we report both actual and an "adjusted" run time, which is actual time minus 80% of function evaluation time. It is planned that future versions of GAMS will separate function and gradient evaluations, and then "adjusted" time should closely approximate actual time.

Description of GAMS Models

Two sets of nonlinear models were solved using the GAMS interface to LSGRG. The first group consists of the nonlinear problems from the standard GAMS library —a collection of GAMS models which are included with every version of GAMS. A description of these models is given in [2]. A total of over thirty nonlinear models are provided, although many are of small dimension. For this research 11 of the largest were selected for computational testing. Table VI lists the problem sizes for these models. They range in size from 43 to 357 variables, and from 9 to 274 constraints. Since they are intended to show the usefulness of GAMS and since MINOS is the default nonlinear solver supplied with GAMS, these problems represent instances for which MINOS performs well.

In order to provide more challenging test problems, the second set consists of models gathered from three different sources. The problem names and sizes are given

Table VI
GAMS Library Problems Statistics

Problem Name	Total Vars	Nonlin Vars	Total Fcns	Nonlin Fcns	Total Nzeros	Nonlin Nzeros
CAMCGE	280	242	244	147	1357	850
GANGES	357	220	274	185	1450	817
KORCGE	96	71	79	44	347	200
CHENERY	44	35	39	23	133	56
OTPOP	106	83	79	35	268	83
PINDYCK	117	64	97	32	337	80
GTM	64	20	25	1	162	20
ETAMAC	98	35	71	10	226	35
WEAPONS	66	65	13	1	156	65
POLLUT	43	40	9	1	167	40
CHAKRA	63	41	43	22	144	41

Table VII
Large GAMS Models Statistics

Problem Name	Total Vars	Nonlin Vars	Total Fcns	Nonlin Fcns	Total Nzeros	Nonlin Nzeros
PEST	61	49	49	38	155	76
6X6	76	75	44	37	436	360
KOR2TB	309	252	220	154	929	587
STNFRD	1014	808	507	202	3637	808
BLEND	363	253	348	132	3620	649
REFNL1	537	57	456	97	2885	194
KTDA2	267	144	192	73	831	180
ECON100	301	200	201	101	700	200
ECON200	601	400	401	201	1400	400

in Table VII. The first four were provided by Tony Brooke of the GAMS Development Company and are problems for which MINOS performed poorly. The next two are taken from the oil industry. BLEND deals with multiperiod gasoline blending, and REFNL1 represents the operation of an oil refinery. The last two models are the GAMS versions of the Economic Growth model discussed in Chapter 8 with 100 and 200 time periods. Note that these models are generally much larger, ranging from 61 to 1014 variables and from 49 to 507 constraints. In fact, STNFRD with 1014 variables and 507 constraints is the largest model solved by LSGRG to date.

Computational Results Using GAMS Models

The results from solving the GAMS library set are given in Table VIII. **Adj Time** is the adjusted time described previously. As a rule, LSGRG compared very well to MINOS on these models. Although it was slower than MINOS on every model, and overall the total time was about 2 times higher (1.5 times adjusted), most of this difference is due to the model GANGES, for which LSGRG required nearly 27 seconds more than MINOS. In addition, LSGRG was able to solve the model OTPOP while MINOS terminated at an infeasible point. Perhaps one factor contributing to the slower run times is the choice of starting points. Note that a majority of the line searches occur in phase I, so that LSGRG is spending a large portion of its time getting feasible. As an infeasible point algorithm, MINOS requires no phase I procedure. Hence, these results suggest that LSGRG compares well with MINOS in both execution time and reliability.

The second set of GAMS models offers a greater challenge. These are somewhat larger than the Library models and many have proven difficult for MINOS to solve. Results from solving these are given in Table IX. Note that MINOS was unable to find a feasible point for four problems, while LSGRG solved the entire set. For the five problems that both optimizers solved, LSGRG required over 5 times the run time of MINOS (3.5 times adjusted). However, most of this difference is due to one model, BLEND, for which LSGRG used 330 more seconds than MINOS. Also, on the problems which LSGRG solved but MINOS did not, LSGRG used less time to find the actual solution for all but STNFRD. Again, note that for all but four instances, LSGRG is forced to spend much of its time in phase I.

Overall, these results are consistent with our underlying philosophy of stressing reliability without compromising execution times. The conservative approaches employed allowed LSGRG to solve every problem, although the resulting run times were generally slower than MINOS. However, by concentrating on the parts of the code which require the most time, it is expected that future improvements can be made which will decrease run time without sacrificing reliability.

Experiments with Scaling

Scaling the rows and/or columns of a mathematical program often speeds its solution. The scaling option in LSGRG scales all rows and columns so that the geometric means of the elements in each row and each column of the scaled Jacobian matrix are close to unity. The multipass procedure is a modification of that in MINOS, and is described in [23]. It is implemented in LSGRG as a user-invoked option whose default value is FALSE. When TRUE, scaling is applied only to the Jacobian evaluated at the initial point.

Here, we consider the effects of scaling on all of the GAMS library models, and four of the models from Table VII. Of the five models from Table VII omitted, LSGRG could not solve STNFRD, BLEND, and REFNL1 using scaling, and the two Economic Growth models showed

Table VIII
GAMS Library Results

	MINOS	LSGRG						
Model Name	Time (sec)	Time (sec)	Adj Time	Fcn Call	Grad Call	Newt Avg	Tot LS	Ph-1 LS
CAMCGE	6.00	13.57	10.25	229	30	0.91	28	28
GANGES	5.55	32.43	22.08	573	79	0.77	77	77
KORCGE	0.89	1.93	1.37	167	23	1.23	21	21
CHENERY	0.57	1.83	1.25	508	59	1.34	56	28
OTPOP	10.80[a]	12.43	7.78	1951	171	1.82	168	115
PINDYCK	2.26	7.51	6.12	764	110	1.34	106	58
GTM	0.56	1.16	10.90	284	61	0.27	57	6
ETAMAC	2.13	5.07	3.78	639	118	0.66	115	84
WEAPONS	1.85	5.94	3.40	968	167	0.07	164	8
POLLUT	0.64	0.92	0.50	246	32	0.25	30	5
CHAKRA	1.09	1.41	0.77	259	34	0.77	31	19
Totals[b]	34.55	71.77	50.42					

[a]No feasible point found.
[b]Does not include OTPOP.

Table IX
Large GAMS Models Results

	MINOS	LSGRG						
Model Name	Time (sec)	Time (sec)	Adj Time	Fcn Call	Grad Call	Newt Avg	Tot LS	Ph-1 LS
PEST	3.71[a]	2.16	1.34	352	55	1.12	52	40
6×6	7.93[a]	6.00	3.63	651	91	1.29	87	0
BLEND	34.91	364.62	198.06	3747	454	1.05	450	377
REFNL1	5.72	42.36	41.04	355	45	0.71	43	31
KOR2TB	5.55	54.17	32.11	1765	227	2.04	225	225
KTDA2	48.05	71.16	49.76	2063	391	1.18	386	187
STNFRD	254.79[a]	404.78	242.54	3431	451	0.93	449	74
ECON100	17.04	30.73	23.28	1152	269	0.69	267	1
ECON200	114.08[a]	83.07	61.14	1711	255	0.77	253	1
Totals[b]	111.27	563.04	344.25					

[a]No feasible point found with MINOS.
[b]Does not include models marked with an *a*.

little difference. Results are shown in Table X. On 7 out of the 15 problems, scaling was slightly or moderately helpful. On two models, GANGES and KORCGE, performance improved significantly, reducing run times from 100.45 and 5.92 seconds to 32.43 and 1.93 seconds respectively. On the other 6 models, scaling degraded performance. The worst result was for KTDA2 for which the execution time jumped from 71.16 seconds to 334.35 seconds, a nearly fivefold increase. These experiments indicate that the effects of scaling are highly problem dependent, but that it can be useful on many problems.

8. Conclusion

We have presented details of a large scale GRG algorithm and its implementation, LSGRG. Like its predecessor GRG2, LSGRG is Simplex-like, achieving and then retaining feasibility using a two phase framework. Retaining feasibility is computationally expensive but, by careful

Table X
Effects of Scaling on GAMS Models

	Unscaled				Scaled			
Model	Line Sear	Fcn Call	Grad Call	Time (sec)	Line Sear	Fcn Call	Grad Call	Time (sec)
CAMCGE	33	235	35	14.02	28	229	30	13.57
GANGES	196	1892	202	100.45	77	573	79	32.43
KORCGE	81	481	85	5.92	21	1.67	23	1.93
CHENERY	81	872	84	2.74	56	508	59	1.83
OTPOP	187	2036	190	16.24	168	1951	171	12.43
PINDYCK	122	1001	127	9.13	106	764	110	7.51
GTM	49	230	52	0.95	57	284	61	1.16
ETAMAC	92	447	95	3.83	115	639	118	5.07
WEAPONS	159	966	164	5.60	164	968	167	5.94
POLLUT	39	286	41	1.05	30	246	32	0.92
CHAKRA	58	502	62	2.42	31	259	34	1.41
PEST	52	352	55	2.16	156	1337	161	7.46
6X6	87	651	91	6.00	95	867	99	7.47
KOR2TB	225	1765	227	54.17	186	1672	188	51.06
KTDA2	386	2063	391	71.16	421	3435	424	334.35
Totals	1847	13779	1901	295.84	1711	13899	1756	484.54

design and judicious use of heuristics, this cost can be made reasonably small. The average number of Newton iterations per call in the two series of GAMS models in Section 7 is almost always near unity. The resulting algorithm still requires many more function and gradient evaluations than its chief competitors, MINOS and SQP. However its overhead, the algorithmic effort per function evaluation, is much smaller than for these others. The result is a code which is acceptably fast and seems to be quite reliable, solving several GAMS models where MINOS failed.

Of course, LSGRG needs much more testing before its value can be assessed with any confidence. We plan to perform some of these experiments in an "optimum-following" mode, where the object is to repeatedly find optimal solutions as model parameters change, starting from the previous optimum. This situation is encountered in real time process unit optimization, where operations are reoptimized whenever product specifications, process parameters, or economic factors change significantly. Further tests are also planned through the GAMS and LINGO interfaces. A key question is whether a large scale GRG algorithm can quickly and reliably solve realistic nonlinear problems with 100 to 5000 rows. In this regard, we wish to solicit challenging nonlinear models of any size from the GAMS user community. Addition of such problems to the GAMS library will further enhance its value.

As many experienced modelers can attest, building and solving nonlinear programs can still be a frustrating exercise, especially for the uninitiated. While, at its best, this process is difficult, solver failures often render it much more so in the NLP realm. We believe that, by including two or three very different NLP solvers, the authors of modeling languages can ease this problem considerably. The results presented here indicate that a good GRG implementation should be included in such systems.

REFERENCES

1. J. Abadie and J. Carpentier, 1969. Generalization of the Wolfe Reduced Gradient Method to the Case of Nonlinear Constraints, in R. Fletcher, (ed), *Optimization*, Academic Press, New York, pp. 37–47.
2. A. Brooke, D. Kendrick and A. Meeraus, 1988. *GAMS —A Users Guide*, Scientific Press, Redwood City, CA.
3. A. K. Cline, C. B. Moler, G. W. Stewart and J. H. Wilkinson, 1979. An Estimate of the Condition Number of a Matrix, *SIAM Journal on Numerical Analysis*, *16:2*, 368–375.
4. T.F. Coleman, B. Garbow and J. Moré, 1984. Software for Estimating Sparse Jacobian Matrices, *ACM Transactions on Mathematical Software 10:3*, 329–345.
5. K. Cunningham and Schrage, L.1989. *The LINGO Modeling Language*, Lindo Systems, Inc., Chicago, IL.
6. R.S. Dembo, 1976. A Set of Geometric Programming Test Problems and Their Solutions, *Mathematical Programming 10*, 192–213.
7. R.S. Dembo and J.G. Klincewicz, 1985. Dealing with Degeneracy in Reduced Gradient Algorithms, *Mathematical Programming 31*, 357–363.

8. J.J. Dongarra, C.B. Moler, J.R. Bunch and G.W. Stewart 1979, *LINPACK Users Guide*, SIAM Press, Philadelphia, PA.
9. A. Drud, 1985. CONOPT—A GRG-Code for large sparse dynamic nonlinear optimization problems, *Mathematical Programming 31*, 153-191.
10. R. Fourer, D. Gay and B. Kernighan, 1990. A Modeling Language for Mathematical Programming, *Management Science 36:5*, 519-554.
11. F. Glover, 1989. Tabu Search—Part 1, *ORSA Journal on Computing 1:3*, 190-206.
12. D.M. Himmelblau, 1972. *Applied Nonlinear Programming*, McGraw-Hill, New York.
13. L.S. Lasdon, A. Waren, A. Jain and M. Ratner, 1978. Design and Testing of a Generalized Reduced Gradient Code for Nonlinear Programming, *ACM Transactions on Mathematical Software 4:1*, 34-50.
14. J.L. Liebman, et al., 1986. *Modeling and Optimization with GINO*, Scientific Press, Redwood City, CA.
15. D.C. Liu and Nocedal, J., 1988. On the limited memory BFGS method for large scale optimization, Report NAM 03, Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL.
16. D.G. Luenberger, 1984. *Introduction to Linear and Nonlinear Programming*, Ed. 2, Addison-Wesley, Reading, MA.
17. D. Mahidhara and L.S. Lasdon, 1990. An SQP algorithm for large sparse nonlinear programs, Working Paper, MSIS Department, School of Business, The University of Texas, Austin, TX.
18. J. More, 1990. Numerical Solution of Bound Constrained Problems, Technical Memo ANL/MCS 96, Argonne National Laboratories, 9700 S. Cass Ave., Argonne, IL.
19. B.A. Murtagh and M.A. Saunders, 1978, Large Scale Linearly Constrained Optimization, *Mathematical Programming 14*, 41-72.
20. B.A. Murtagh and M.A. Saunders, 1982. A Projected Lagrangian Algorithm and Its Implementation for Sparse Nonlinear Constraints, *Mathematical Programming Study 16*, 84-117.
21. M.J.D. Powell, 1978, Algorithms for Nonlinear Constraints that Use Lagrangian Functions, *Mathematical Programming 14*, 224-248.
22. G.V. Reklaitis, A. Ravindran, and K.M. Ragsdell, 1983. *Engineering Optimization—Methods and Applications*, John Wiley & Sons, New York, Section 12.6.
23. S. Smith, 1990. The Design and Testing of a Large Scale GRG Algorithm and GRASP Approaches for Independent Set and Graph Coloring, Doctoral Dissertation, Mechanical Engineering Dept., University of Texas at Austin, Austin, TX.
24. U.H. Suhl and L. Aittoniemi, 1987. Computing Sparse LU-Factorizations for Large-Scale Linear Programming Bases, Working Paper, Freie Universitat, Berlin.