The Solution of Large-Scale Optimization Problems
(using MATLAB)

Thomas F. Coleman[1]
Computer Science
Cornell University
Ithaca , New York

February 17, 1999

[1]coleman@tc.cornell.edu http://www.tc.cornell.edu/~ coleman

Contents

Chapter 1

Sparse Linear Algebra Tools

The ability to solve large-scale optimization problems in an efficient manner depends, to a significant degree, on sparse linear algebra technology. The solution of continuous optimization problems typically involves a sequence of linearizations; as illustrated with numerous examples in Chapter 1, the linear subproblems are often sparse, either explicitly or implicitly (due to structure).

There are two basic approaches to solving sparse linear systems, direct and iterative. Both approaches have their place. In this chapter we review direct and iterative techniques for solving several sparse linear algebra problems that arise in the optimization setting. The emphasis in this chapter is on understanding some of the basic problems and approaches, with an eye toward our ultimate goal, the effective solution of large optimization problems. References to more complete treatments of sparse linear algebra techniques are provided below where appropriate.

1.1 Direct Factorization Methods.

We consider four important problems: the solution of symmetric positive definite systems using sparse Cholesky techniques, the solution of general square systems using a sparse version of LU-factorization, the sparse QR-factorization for the least-squares solution of overdetermined systems, and techniques for working with an underdetermined constraint system.

1.1.1 Sparse Cholesky Factorization

Symmetric positive definite (SPD) systems, $Ax = b$, frequently arise in optimization. For example, the solution of the Newton direction in the neighbourhood of a strong (unconstrained) local minimizer involves SPD systems.

The workhorse for solving SPD systems is the Cholesky factorization, $A = L \cdot L^T$, where L is a lower triangular nonsingular matrix. If we ignore sparsity, the cost of computing the Cholesky factorization is $\frac{n^3}{3}$ flops; after computing $A = L \cdot L^T$ a system $Ax = b$ can be solved by simply solving two triangular systems: $Ly = b$, and $L^T x = y$. The Cholesky factorization is "essentially" unique (i.e., unique up to the signs of the columns of L) and can be computed stably without need for row or column permutations. For more background on the Cholesky factorization, see, for example, [18, 15].

A sparse SPD matrix A may enjoy a sparse Cholesky factorization. Figure 1.1 gives an example: by considering the mechanics of the Cholesky factorization algorithm it is easy to see that in this case the nonzeroes of occupy the same locations as in the lower half of matrix A. Since it is easy

$$A = \begin{pmatrix} x & & & & x \\ & x & & & x \\ & & x & & x \\ & & & x & x \\ x & x & x & x & x \end{pmatrix} \longrightarrow L = \begin{pmatrix} x & & & & \\ & x & & & \\ & & x & & \\ & & & x & \\ x & x & x & x & x \end{pmatrix}$$

Figure 1.1: Cholesky With No Fill

to tailor the Cholesky algorithm to compute only with nonzeroes of L, it follows that the Cholesky factorization of matrix A, with structure illustrated in Figure 1.1, can be computed in $O(n)$ flops.

Unfortunately a sparse SPD matrix A does not always yield a sparse Cholesky factor. The matrix structure in Figure 1.2, a simple symmetric permutation of the matrix in Figure 1.1, yields a dense Cholesky factorization. Nonzero locations in L that correspond to zeros of A are called *fill*

$$A = \begin{pmatrix} x & x & x & x & x \\ x & x & & & \\ x & & x & & \\ x & & & x & \\ x & & & & x \end{pmatrix} \longrightarrow L = \begin{pmatrix} x & & & & \\ x & x & & & \\ x & x & x & & \\ x & x & x & x & \\ x & x & x & x & x \end{pmatrix}$$

Fill

Figure 1.2: Cholesky With Complete Fill

elements, and the collection of such elements is denoted *fill*. There is *complete* fill in Figure 1.2. Therefore the cost of computing the Cholesky factor of A in this case is not reduced by the sparsity of the matrix A: sparsity gives no advantage in this case.

Figures 1.1 and 1.2 illustrate the extreme cases; in general, the set of nonzeros of a Cholesky factor of a sparse SPD matrix A is a superset of the nonzero locations of the matrix A. Since the system $Ax = b$ can be replaced with an equivalent system, $(P \cdot A \cdot P^T)(Px) = Pb$, where P is a permutation matrix, and there are no adverse stability effects when computing the Cholesky factorization of $P \cdot A \cdot P^T$ versus A, there is a possibility of preserving sparsity by using permutation matrices. The idea is to find a permutation matrix P such that the Cholesky factor of $P \cdot A \cdot P^T$ has little fill. For example, if we permute the first and last rows and columns of matrix A in Figure 1.2 then we get a matrix with the structure illustrated in Figure 1.1, and no fill! So in this case a simple permuation reduces the cost of solving $Ax = b$ from $O(n^3)$ to $O(n)$.

To choose a good permutation matrix P we must understand how fill occurs during the Cholesky process. To do this a graph model is useful.

Given a symmetric matrix A of order n, define a labelled *adjacency* graph of A, $G_{adj}(A) = (V, E_A)$, where V is a set of vertices (or nodes), labelled $1, 2, ..., n$ and E_A is a set of edges. Figure 1.3 illustrates the idea. The cardinality of E_A is equal to the number of nonzeros of A strictly below (or above) the diagonal: $(i, j) \in E_A$ if $A(i, j) \ = 0$ for $i > j$. The adjacency graph of $L + L^T$, $G_{adj}(L + L^T) = (V, E_L)$, models the nonzero structure of the Cholesky factor. Note that

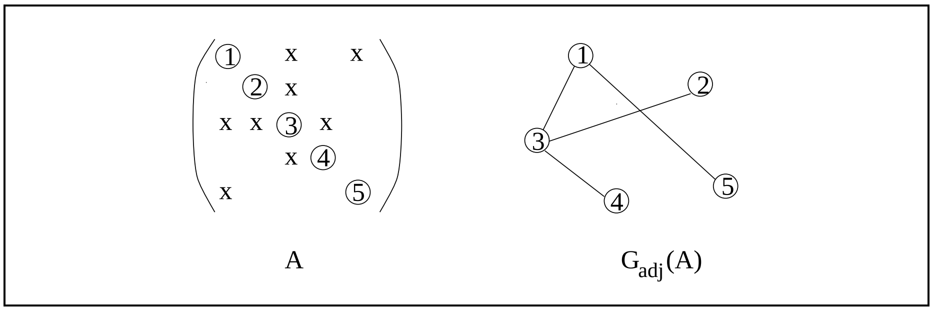

Figure 1.3: The adjacency graph of a symmetric matrix

$E_A \subseteq E_{L+L^T}$.

It is not difficult to see that the graph algorithm given in Figure 1.4 models Cholesky factorization, from a structural point of view, ultimately producing a graph (V, E_{L+L^T}) such that $G_{adj}(L + L^T) = (V, E_{L+L^T})$.

Initialize: $E_L \leftarrow E_A$
For $i = 1 : n$

For each $j > i,\ k > i$ *s.t.* $(i, j) \in E_L,\ (i, k) \in E_L$

$E_L \leftarrow E_L \bigcup \{j, k\}$

Figure 1.4: The Cholesky fill algorithm

Figures 1.5 and 1.6 illustrate the graph model of the Cholesky fill process. In the example in Figure 1.5, the first step of the fill algorithm results in the addition of the fill edge $(3, 5)$; edge $(4, 5)$ is added in the third step, i.e., $i = 3$.

In Figure 1.6 the first step of the Cholesky fill algorithm results in the addition of edge $(4, 5)$; the second step yields 3 new edges, $(3, 5)$, $(3, 7)$, and $(5, 7)$; in the third step (i.e., $i = 3$) edges $(6, 7)$ and $(5, 6)$ are added.

The minimum degree algorithm (MDO). Since space and time requirements for sparse Cholesky factorization increase with fill, a worthwhile objective is to find a permutation matrix P such that $P \cdot A \cdot P^T$ produces minimum fill over all possible permutations. Unfortunately this minimum fill problem is known to be NP-hard [27] and so we cannot expect to have an efficient optimal algorithm. However, many effective heuristics exist: the fill algorithm is easily modified to yield the most popular heuristic, the minimum degree algorithm.

The minimum degree algorithm is described in Figure 1.7. The set of unlabelled vertices is denoted by U, the degree of a vertex v, in the graph induced by nodes in U, is written $deg_U(v)$.

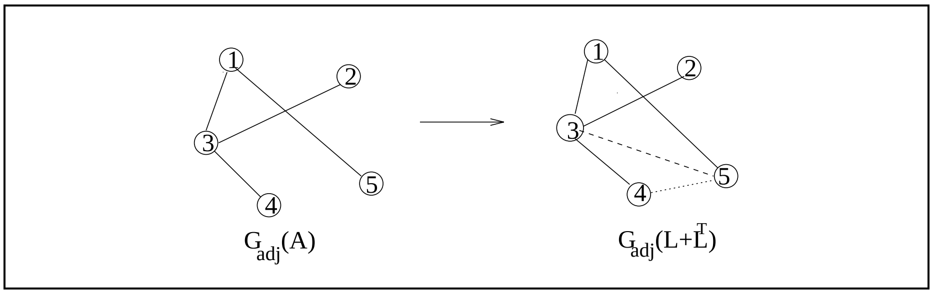

Figure 1.5: Modelling the fill process I

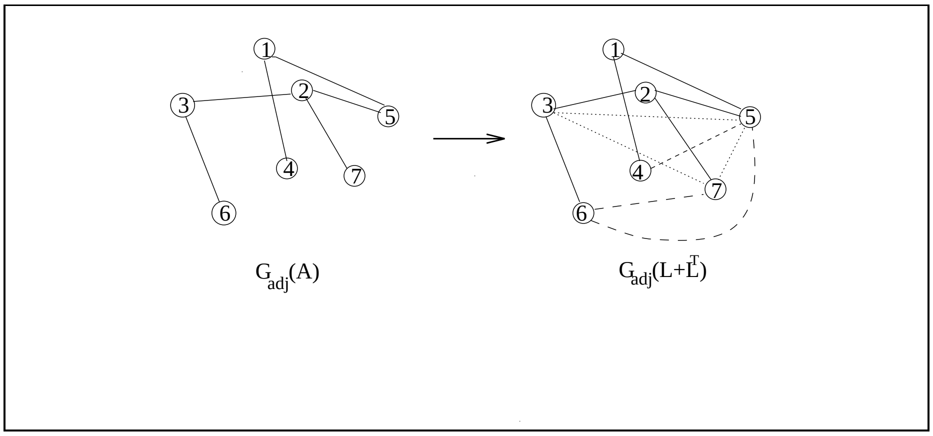

Figure 1.6: Modelling the fill process II

The idea behind MDO is simple: notice that in step i of the Cholesky fill algorithm there will be little fill if the number of higher-numbered neighbours of vertx i is small. Algorithm MDO applies a greedy principle: over all the unlabelled vertices choose the vertex to be labelled i with the fewest neighbours in U.

The minimum degree ordering is the most popular ordering used for general sparse positive definite systems. Efficient implemention depends on data structure questions and practical details beyond the scope of these notes. Indeed, the description of MDO given in Figure 1.7 is not fully deterministic since we have not presented a tie-breaking strategy. See [15, 20, 14, 16] for more details on this and other implementation matters.

Numerous other orderings have been studied and are available. MDO is not always the best choice, and in some contexts can generate dismal results, e.g., [25], and so this is an area where the availability of a collection of different heuristics can be very useful. The goal of some orderings is to find a permutation matrix P such that $P \cdot A \cdot P^T$ has a small bandwidth (or profile), e.g., the Reverse Cuthill-McKee (RCM) ordering [15]. It is easy to see that the Cholesky factor of a matrix A will have the same (lower) band structure as the matrix A. Band and profile methods tend to produce orderings with more fill than MDO and other fill-minimizing orderings, and are therefore more costly in terms of flops. However, band/profile methods are easy to implement and simpler data structures can be used.

Initialize: $E_L \leftarrow E_A, \;\; U \leftarrow V$
For $i = 1 : n$

> *Determine a vtx* $v_{min} \in U$ *where* $deg_U(v_{min}) = \min\{deg_U(v) : v \in U\}$
>
> *Assign index* i *to vtx* v_{min}
>
> *For each* $x \in U, \; y \in U$ *s.t.* $(i, x) \in E_L, \; (i, y) \in E_L$
>
> > $E_L \leftarrow E_L \bigcup \{(x, y)\}$
>
> $U \leftarrow U \; - \; \{v_{min}\}$

Figure 1.7: The minimum degree algorithm(MDO)

MATLAB *solution of SPD systems.* In MATLAB if A is sparse and symmetric with positive diagonal elements then the statement

$$\mathsf{x} = \mathsf{A} \setminus \mathsf{b}$$

triggers sparse Cholesky factorization with minimum degree ordering. If it turns out that A is not positive definite then, when Cholesky fails, a general sparse LU-factorization (with column minimum degree ordering) is automatically invoked to solve $Ax = b$. See §1.1.2 and [17] for more details.

It is often advantageous to compute and save the permutation vector p (representing the permutation matrix P) and Cholesky factor for later use. In MATLAB this can be done by first determining the permutation vector p and then computing

```
R = chol(A(p,p));
```

where R is the transpose of the Cholesky factor of A(p,p). Built-in MATLAB orderings are:

- p=symmmd(A): Symmetric minimum degree ordering – referred to as MDO above.
- p=colperm(A): Column permutation – A(p,p) has columns ordered by increasing number of nonzeros.
- p=symrcm(A): Symmetric Reverse Cuthill-McKee [15] – an ordering designed to minimize the profile of a symmetric sparse matrix.

After determining p and R, the solution to $Ax = b$ can be recovered:

```
y = R' \ b(p);    x(p) = R \ y;
```

Multifrontal methods. Recent work in the area of sparse SPD systems has lead to the development of the multifrontal idea, a complex but relatively fast way to compute the sparse Cholesky factorization [21]. Speed comes from the organization of work so that updates are performed on dense submatrices.

1.1.2 Sparse LU.

The direct solution of general sparse square systems, $Ax = b$, is generally obtained by a sparse variant of LU-factorization. However, sometimes a given sparse system can be subdivided into smaller problems by detecting a block triangular form (BTF). For example, suppose that a system $Ax = b$ can be written,

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ & A_{22} & A_{23} \\ & & A_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}, \tag{1.1}$$

where each diagonal block A_{ii} is square (though not necessarily of the same size). Clearly only the diagonal blocks A_{ii} in (1.1) need be factored in order to solve $Ax = b$. It turns out that the block triangular form of a sparse matrix is "essentially" unique and can be determined by fast graph-theoretic techniques. The usual approach divides into two parts:

(i) Determine a permutation matrix Q such that the matrix $\bar{A} = A \cdot Q$ has a zero-free diagonal.

(ii) Determine a permutation matrix P such that $\tilde{A} = P \cdot \bar{A} \cdot P^T$ is in BTF.

Step (i) is solved by finding a *maximum matching* in the bipartite graph G_B representing the sparsity structure of matrix A. In particular, $G_B = (V_R, V_C, E_A)$ is a bipartite graph

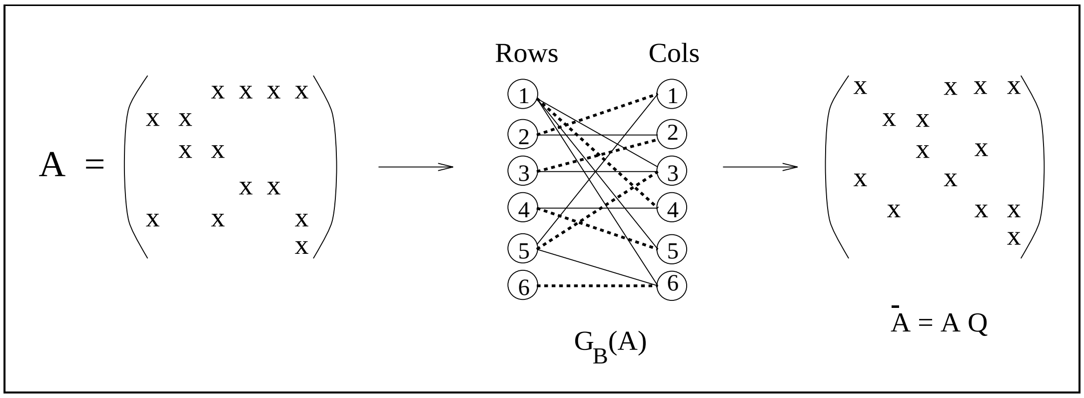

Figure 1.8: A maximum matching in G_B

where V_R is a set of vertices corresponding to the rows of A, numbered $1, 2, ..., n$; V_C is a set of vertices corresponding to the columns of A, numbered $1, 2, ..., n$; index pair $(i, j) \in E_A$ if $A_{i,j} \neq 0$.

A *matching* is a set of independent edges (i.e., edges that do not share a common endpoint), and a maximum matching is a matching of maximum size. Figure 1.8 illustrates the situation. There are efficient algorithms to find maximum matchings, some of which are tailored to this context, e.g., [8]. If the maximum matching has size less than n, where n is the order of the matrix A, then A is singular for all possible values of the designated nonzeros. Moreover, the structurally nonsingular part of A is identified by the maximum matching. If the maximum matching is of cardinality n then the columns of A can be permuted, to reflect the matching, to yield a zero-free diagonal.

Step (ii) of this procedure works with another graph, $G_D(\bar{A})$: $G_D(\bar{A})$ is a directed graph defined by the structure of $\bar{A}$. To define $G_D(\bar{A}) = (V, \vec{E}_{\bar{A}})$ let the n vertices of V represent the diagonal

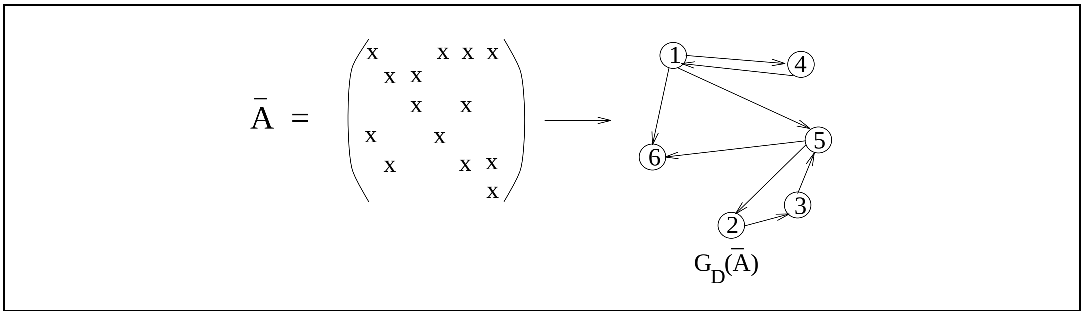

Figure 1.9: The directed graph $G_D(\bar{A})$

elements of $\bar{A}$; $\overrightarrow{(i,j)} \in \vec{E}_{\bar{A}}$ is a directed edge from vertex i to vertex j if $A_{i,j} \neq 0$. Figure 1.9 illustrates the construction of $G_D(\bar{A})$ with an example.

The BTF of $\bar{A}$ can now be constructed by determining the maximum strongly connected components of $G_D(\bar{A})$. Fast algorithms exist for the construction of this component graph which, by definition, is an acyclic graph. As illustrated in Figure 1.4 the topological ordering of the com-

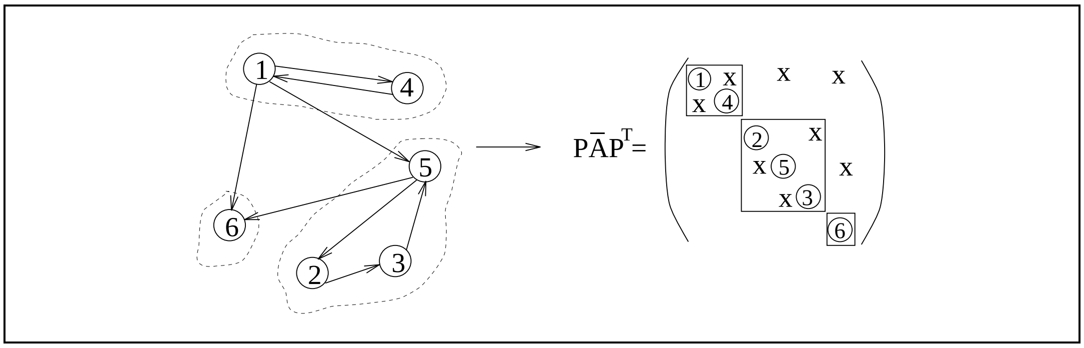

Figure 1.10: Achieving the BTF

ponent graph yields the BTF. For more details and background on this procedure, see [8]. The MATLAB dmperm function can be used to find the BTF of a matrix.

Threshold Pivoting. If the BTF of a matrix A is a single diagonal block consisting of the matrix itself, A is called irreducible. Equivalently, $G_D(\bar{A} = AQ)$ consists of a single strongly connected component. From the discussion above it is clear that we need only consider the LU-factorization of irreducible matrices.

The most popular way to solve a sparse, square, irreducible system $Ax = b$ is to factor $P \cdot A = L \cdot U$ where P is a permutation matrix chosen to *both* preserve sparsity and control numerical stability. The typical strategy is to proceed with the usual LU-factorization method with partial pivoting, with one difference. At the k^{th} iteration, instead of choosing as the next pivot row that row corresponding to the largest pivot element in the k^{th} column, choose the row with the *fewest nonzeros* in the remaining (updated) submatrix of A, i.e., $A(k:n, k:n)$, subject to a (relative) threshold tolerance on the magnitude of the consequent pivot element. For example, the chosen pivot element, $A(i,k)$, should satisfy

$$|A(i,k)| \leq \mu \cdot \max\{|A(j,k)|, j = k:n\}$$

for a positive constant $\mu \leq 1$. Of course the choice $\mu = 1$ reduces to the usual partial pivoting LU-factorization, with no regard for sparsity. A typical choice is $\mu = .1$ or perhaps $\mu = .01$.

Unlike the Cholesky case, the fill locations under sparse LU with threshold pivoting cannot be predetermined because the pivot sequence is determined on the fly. The pivot sequence (i.e., permutation matrix) cannot be determined in advance because it depends on the numerical values created in the process. This is unfortunate for several reasons: e.g., the need for dynamic memory allocation.

The sparse LU-factorization in MATLAB is not based on the threshold pivoting idea. Instead the conflicting goals of preserving sparsity and maintaining stability are decoupled. Column ordering is performed, a priori: i.e, $\bar{A} = A \cdot Q$, where Q is a permutation matrix chosen to reduce fill. Subsequently, standard LU-factorization with partial pivoting is applied to $\bar{A}$, i.e., $P \cdot \bar{A} = LU$, where P is chosen on stability grounds only. (Note: Of course it is assumed that computations are performed with nonzero elements only.)

Given the column permutation vector q (vector q represents the permutation matrix Q), the sparse LU-factorization can be determined in MATLAB:

```
[L,U,P] = lu(A(:,q));
```

where P is the (sparse!) row permutation matrix determined by the MATLAB lu function. Built-in MATLAB orderings to determine the column permutation vector q are:

q=colmmd(A): Minimum degree orderings based on the structure of $A^T A$.

q=colperm(A): Column permutation – A(:,q) has columns ordered by increasing number of nonzeros.

After determining q and then L,U, and P, the solution to $Ax = b$ can be recovered:

```
y = L \ P * b;    x(q) = U \ y.
```

The MATLAB backslash hierarchy. It is convenient to use the MATLAB backslash command,

```
x = A \ b
```

if there is just one system involving the matrix A. Otherwise it is better to use the lu function so that the factors can be accessed in subsequent computations. MATLAB executes the backslash in the following order:

1. If A is not square compute the least-squares solution (see §1.1.3)
2. If A is a triangular matrix (i.e., permutable to triangular form) compute a (sparse) triangular solve, i.e., no factorization is required.
3. If A is symmetric with positive diagonal elements, *attempt* a sparse Cholesky factorization with the minimum degree ordering.
4. Solve $Ax = b$ using the LU-factorization, as described above, with the columns ordered with a column minimum degree ordering.

1.1.3 Sparse Linear Least-squares.

The solution of large, sparse, overdetermined systems of equations in the least-squares sense, i.e.,

$$\min \|Ax - b\|_2,$$

where A is m-by-n, $m > n$, is a fundamental computation. If matrix A is of full column rank then the least-squares solution is also the solution to the SPD system of normal equations,

$$(A^T A)x = A^T b;$$

however, solving the least-squares problem by forming and solving the normal system is not usually recommended[1], partly because the normal system is often significantly more ill-conditioned than the original problem. For more information regarding the linear least-squares problem consult the comprehensive book by Bjorck [3].

There is a fundamental structure to sparse rectangular matrices that is revealed by the Dulmage-Mendelsohn Decomposition (DMD). The DMD was originally introduced in a series of papers by Dulmage and Mendelsohn [9, 10, 11]; Pothen and Fan [24] provide an illuminating discussion of the DMD (along with efficient algorithms).

The Dulmage-Mendelsohn Decomposition is obtained by finding a maximum matching in the bipartite graph $G_B(A) = (V_R, V_C, E)$, where V_R is a set of vertices representing the rows of A, numbered $1, ..., m$, V_C is the set of vertices representing the columns of A numbered $1, ..., n$, and E is a set of edges: $(i, j) \in E$ if $A(i, j) \neq 0$. The DMD defines permutation matrices P and Q so that $\bar{A} = P \cdot A \cdot Q$ has an upper triangular shape (with rectangular blocks). In particular,

$$\bar{A} = \begin{pmatrix} \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ & \bar{A}_{22} & \bar{A}_{23} \\ & & \bar{A}_{33} \end{pmatrix}, \tag{1.2}$$

where $\bar{A}_{11}$ is a *rectangular* matrix with more columns than rows, $\bar{A}_{22}$ is a structurally non-singular *square* matrix, and $\bar{A}_{33}$ is a *rectangular* matrix with more rows than columns. The *structural rank*[2] of $\bar{A}_{11}$ is equal to the number of rows in $\bar{A}_{11}$; the structural rank of $\bar{A}_{33}$ is equal to the number of columns in $\bar{A}_{33}$.

Figures 1.11 and 1.12, from [24], illustrate a maximum matching in $G_B(A)$ and the corresponding DMD, respectively.

Submatrix $\bar{A}_{11}$ is defined by the set of all vertices reachable by alternating paths[3] beginning at the unmatched vertices in V_C. In the example in Figure 1.11, $\bar{A}_{11}$ is defined by the column index set HC and the row index set HR.

Submatrix $\bar{A}_{33}$ is defined by the set of all vertices reachable by alternating paths beginning at the unmatched vertices in V_R. In the example in Figure 1.11, $\bar{A}_{33}$ is defined by the column index set VC and the row index set VR.

[1] There are exceptions to this rule, e.g., [26]

[2] The *structural rank* of a matrix A is the size of the maximum matching of $G_B(A)$. This is equivalent to the numerical rank of A when the nonzero locations of A are assigned random values.

[3] An *alternating path* in a graph with a set of matched edges M is a path, with no repeated vertices, such that alternate edges are in M.

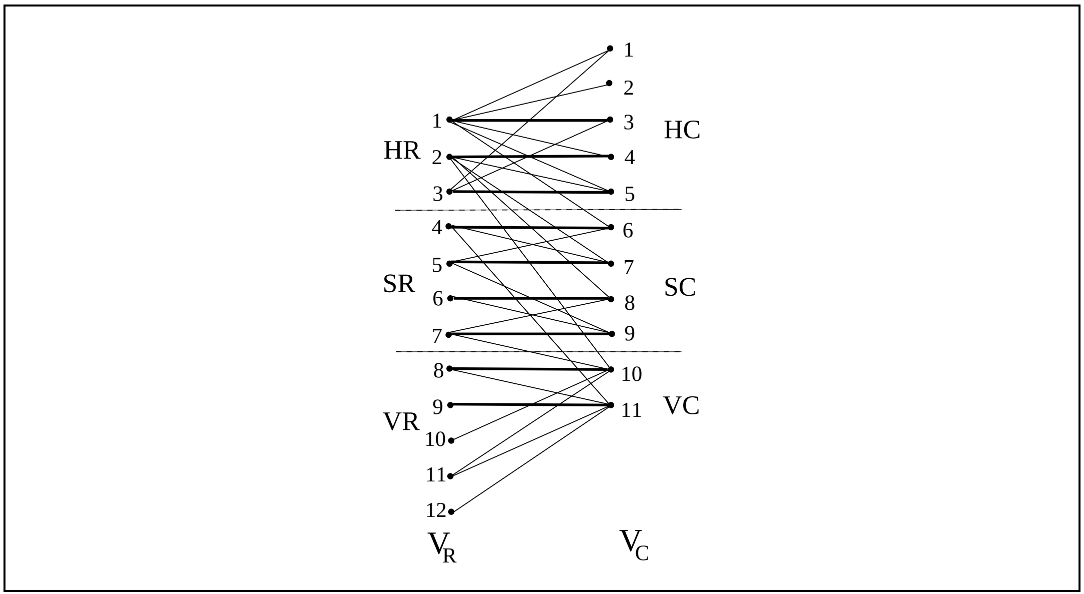

Figure 1.11: A maximum matching in G_B defines the DMD

The remaining vertices, i.e., vertices unreachable by alternating paths beginning at unmatched vertices, define the square submatrix $\bar{A}_{22}$; in the example in Figure 1.11, the column vertices SC and the row vertices SR define submatrix $\bar{A}_{22}$.

This overall structure is independent of the maximum matching that is used.

We note several important properties of the DMD:

1. If $A \in \Re^{m \times n}$ is of full column rank then there is no submatrix $\bar{A}_{11}$. This follows because full column rank implies that the cardinality of a maximum matching is n, the number of columns.

2. If A is of full row rank then there is no submatrix $\bar{A}_{33}$. This follows because full row rank implies that the cardinality of a maximum matching is m, the number of rows.

3. If submatrix $\bar{A}_{11}$ exists then the structural rank of $\bar{A}_{11}$ is equal to the number of rows of $\bar{A}_{11}$.

4. If submatrix $\bar{A}_{33}$ exists then the structural rank of $\bar{A}_{33}$ is equal to the number of columns of $\bar{A}_{33}$.

5. Submatrix $\bar{A}_{22}$ is square and can be permuted to BTF with a zero-free diagonal. See §1.1.2.

6. Both submatrices $\bar{A}_{11}^T$ and $\bar{A}_{33}$, when they exist, have the *Strong Hall Property* (SHP). A p-by-q matrix with $p \leq q$ has the SHP if every submatrix induced by a subset of $r < q$ columns has nonzeroes in at least $r + 1$ rows. As we discuss next, existence of the SHP is important for predicting fill.

Sparse QR. George and Heath [13] suggest the following strategy for solving overdetermined sparse least-squares problems, $\min\{\|Ax - b\|_2\}$:

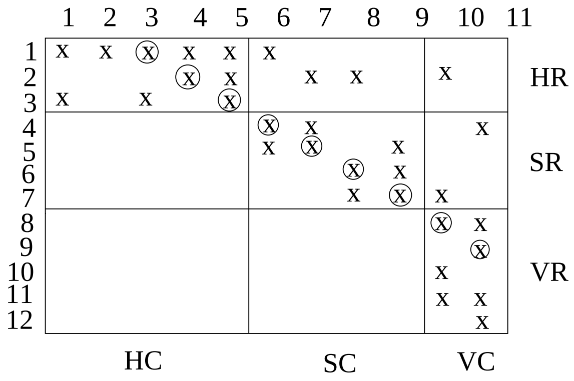

Figure 1.12: The Dulmage-Mendelsohn Decomposition (DMD)

(i) Determine the *structure* of $A^T A$; i.e., determine the adjacency graph $G_{adj}(A^T A)$. This does *not* involve the computation of the values of $A^T A$.

(ii) Apply a Cholesky ordering technique to $G_{adj}(A^T A)$ to determine a permutation matrix P that would be effective for the sparse Cholesky factorization of $A^T A$ (see §1.1.1); set up a storage scheme that would accomodate the sparse Cholesky factor of $P \cdot (A^T A) \cdot P$.

(iii) Permute the columns of A: $\bar{A} \leftarrow A \cdot P$.

(iv) Compute the sparse factor $R \leftarrow Q^T \bar{A}$, where Q is a sequence of Givens rotations applied to the nonzero elements. (*Sparse* Householder transformations could also be used).

The George-Heath idea is clever in that implementation primarily rests on well-developed sparse Cholesky ideas and software. But, does the G-H strategy correctly predict the structure of R? The answer is no, in general. For example, consider

$$A = \begin{pmatrix} x & x & x & x & x \\ & x & & & \\ & & x & & \\ & & & x & \\ & & & & x \end{pmatrix}. \tag{1.3}$$

In this case matrix A is its own upper triangular factor R; however, the G-H scheme applied to A predicts a completely dense structure for R (since $A^T A$ is dense).

Fortunately, it is possible to characterize those matrix structures for which the G-H strategy predicts excessive fill (and G-H never predicts too little fill); moreover, the DMD allows for the removal of the offending structures. In particular, the G-H idea applied to SHP matrices correctly predicts the fill in R [5]; further, by using the DMD it is only really necessary to apply G-H to SHP matrices. To see this recall from §1.1.3 that, by (1.2) and the full column-rank assumption, the DMD of A reduces to a 2-by-2 block matrix,

$$\bar{A} = \begin{pmatrix} \bar{A}_{22} & \bar{A}_{23} \\ & \bar{A}_{33} \end{pmatrix}, \tag{1.4}$$

where $\bar{A}_{22}$ is a (structurally) full rank *square* submatrix and $\bar{A}_{33}$ is a SHP matrix with more rows than columns. (Of course $\bar{A}_{22}$ may not exist in which case A itself is a SHP matrix.) For simplicity let us relabel $\bar{A}$:

$$\bar{A} = \begin{pmatrix} B & C \\ & D \end{pmatrix}. \tag{1.5}$$

So, B is square and nonsingular, D is a SHP matrix with more rows than columns. Clearly if we partition

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

the least-squares problem, $\min \|Ax - b\|_2$, reduces to the sequence

(i) $\min \|Dx_2 - b_2\|_2$,

(ii) Solve $Bx_1 = b_1 - Cx_2$.

Therefore we need only consider the sparse QR-factorization applied to SHP matrices.

Notice that use of the DMD allows for the correct prediction of fill in example (1.3).

Two quick-and-dirty sparse least-squares solvers. It is possible to solve sparse least-squares problems with a sparse *square* system solver. Indeed, until the release of MATLAB 5.0, the MATLAB sparse least-squares solver was based on the sparse LU-factorization of an "augmented" square linear system. In particular, it is easy to see that the least-squares solution to the overdetermined full column-rank problem, $\min \|Ax - b\|_2$, is obtained by solving:

$$\begin{pmatrix} \alpha I & A \\ A^T & 0 \end{pmatrix} \begin{pmatrix} \frac{r}{\alpha} \\ x \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix}, \tag{1.6}$$

where α is a scaling parameter[4]. System (1.6) can be solved by a sparse square solver (MATLAB uses LU-factorization). Symmetry can be used though obviously this system is not positive definite so Cholesky is not an option.

A second idea is due to Peters and Wilkinson [23]. It is based on the observation that if LU-factorization with partial pivoting is applied to a matrix A, $P \cdot A = L \cdot U$, then any ill-conditioning in A tends to show up in matrix U whilst L remains relatively well-conditioned. This appears to hold even when A is rectangular (more rows than columns) in which case L is also rectangular, U is square and nonsingular. It is easy to see that the original least-squares problem can now be replaced with

$$\min \|Ly - \bar{b}\|_2, \tag{1.7}$$

where $\bar{b} = P \cdot b$; the least squares solution x is recovered by solving $Ux = y$. Now since L is expected to be well-conditioned, the least-squares problem (1.7) can be solved by a sparse Cholesky technique applied to the normal system. Of course to be effective it is imperative that L and U be sparse; therefore, sparsity preserving techniques such as those described in §1.1.2 should be used (e.g., threshold pivoting or column ordering).

These two sparse least-squares techniques are easy to implement in MATLAB and can be useful for "small enough" problems. However, they become less competitive for larger problems.

[4]MATLAB 4.2 used the heuristic value for α, $\alpha = max\frac{|A(i,j)|}{1000}$.

Multifrontal methods. Recent research in this area has led to the development of fast "multifrontal" methods to compute the upper triangular factor R, e.g., [7, 22, 6] The idea is to organize the sparse QR using row and column permutations, so that *dense* dense matrix computations can be employed on submatrices. The resulting algorithms and implementations are quite complex; however, due to the use of efficient dense matrix technology, multifrontal methods are relatively fast.

Solving a subsequent system. Frequently arising in optimization is the need to solve subsequent least-squares problems with the same matrix. For large systems it is wasteful to re-compute the upper-triangular factor R. However, the orthogonal transformations that were used to compute R are not usually saved due to large memory requirements. The question then is how to solve subsequent least-squares problems given only R, A, and the permutations vector p. One effective approach is to use the method of semi-normal equations [3, 2], i.e., solve

$$(R^T R)x \;=\; A^T b,$$

followed by one or two iterative refinement steps. Iterative refinement can be effectice in this context even without requiring the use of extra precision [3]. Note that since R is already computed no further factorizations are required to implement either semi-normal equations or iterative refinement. Function SPLLS, described in Appendix A and available in the CS622 Directory, implements this procedure.

Underdetermined systems. A common request in optimization is to determine a feasible point to an underdetermined system of linear equations. That is, determine x such that $Ax \;=\; b$, where A is m-by-n with $m < n$.

The least-squares solution x_{LS} satisfies $x_{LS} \;=\; A^T y$ where $A \cdot A^T y \;=\; b$. To compute x_{LS} suppose

$$Q \cdot A^T P \;=\; \begin{pmatrix} R \\ 0 \end{pmatrix}.$$

Then, $x_{LS} \;=\; P^T (R^T R)^{-1} P b$. The following MATLAB program computes the least-squares solution using SPLLS:

```
[R,p] = splls(A'); w = R'\ b(p); y(p) = R\ w; x = A'* y;
```

This MATLAB script, along with iterative refinement, is encoded in function feas in the CS622 Directory:

```
x = feas(A,b)
```

A *basic solution* has at least $n - m$ components of x at zero. Basic solutions can be computed by "inverting" an m-by-m submatrix of A. For example, if

$$A \;=\; (A_1, A_2)$$

where A_1 is invertible, then in theory

$$x \;=\; \begin{pmatrix} -A_1^{-1} b \\ 0 \end{pmatrix} \tag{1.8}$$

is a basic solution.

It is important to choose a well-conditioned matrix A_1; otherwise, the computed basic solution may not satisfy the linear system! One way to proceed is to begin with a sparse LU-factorization of A^T,

$$P \cdot A^T \;=\; L \cdot U \;=\; \begin{pmatrix} L_1 \\ L_2 \end{pmatrix} \cdot U$$

A basic solution is

$$x \;=\; P^T \begin{pmatrix} L_1^{-T} U^{-T} b \\ 0 \end{pmatrix}. \tag{1.9}$$

To implement in MATLAB, using column ordering of A^T for sparsity:

```
[m,n] = size(A); AT = A';
pcol = colmmd(AT); [L,U,P] = lu(AT(:,p));
y = (L(1:m,:)') \ (U' \ b(pcol));
yx = [y;zeros(n-m),1)]; x = P'*yx;
```

M-file x = basic(A,b), in the CS622 Directory, encodes the MATLAB script above.

1.2 Exercises

1. Devise a *modified* Cholesky factorization algorithm

$$[R, v] = mchol(H)$$

which accepts as input a symmetric matrix H and returns R, the transpose of the Cholesky factor, if H is positive definite. If H is not positive definite then *mchol* returns a direction of non-positive curvature, v. So, $v \neq 0$ and $v^T H v \leq 0$. In this case R should contain the partially computed Cholesky factor. Implement your method using the MATLAB chol function, and try it on matrices $H = Q^T D Q$, where Q is a random orthogonal matrix of order 10 and D is a diagonal matrix with the magnitude of all elements randomly chosen in $[0, 1]$. Try 4 cases:

(i) The diagonal of D is strictly positive.

(ii) Exactly one of the diagonal elements of D is zero, i.e., $D_{7,7} = 0$; the remaining diagonal elements are positive.

(iii) Exactly one of the diagonal elements of D is negative, i.e., $D_{4,4} < 0$; the remaining diagonal element s are positive.

(iv) The first 5 diagonal elements of D are positive; the remaining diagonal elements are negative.

2. Write an efficient MATLAB Cholesky code for matrices A with an an arrowhead structure as indicated in Figure 1.1. You may assume a convenient data structure of your choice for A. Keep in mind that MATLAB loves to work on vectors, not scalars. Compare your observed efficiency on problems of various dimension, with respect to both the dense and sparse MATLAB Cholesky functions. Comment.

3. Consider a *block* arrowhead generalization of of the arrowhead structure in Figure 1.1: Each diagonal element d_{ii} becomes a dense block D_{ii} of order p_i-by-p_i, $i = 1, ..., N$. The dimension of A is $n = \sum_{i=1}^{N} p_i$. Assume the last block-row is a dense p_N-by-n matrix. Develop a specialized block Cholesky, in analogy to the scalar version in question 2., to solve block arrowhead systems in MATLAB. Test and compare your efficiency with respect to both general sparse and dense Cholesky solvers. Comment on your results.

4. Cholesky factorization can be written in a recursive way (with respect to the problem dimension). Illustrate this and then implement a recursive (dense) Cholesky factorization in MATLAB. Compare your running time to the MATLAB Cholesky solver on problems of various dimensions. Comment.

5. Show that a sparse envelope scheme for Cholesky factorization will generate fill only inside the envelope. Sketch an implementation, indicating a potential data structure of your problem. How much work is an envelope (or skyline) Cholesky factorization as a function of β_i, $i = 1, ..., n$, where β_i is the bandwidth of row i?

6. Let A be a sparse rectangular matrix with bipartite graph $G_B(A)$. Show that the rank of A is never greater than the size of the maximum matching in $G_B(A)$.

7. Let A be a m-by-n matrix with $m > n$, rank $A = n$. Show that row permutations of A will not lessen the fill in R when using orthogonal transformations to compute an upper triangular form R. Illustrate that the amount of work required to obtain the upper triangular form can depend on row ordering.

8. Write an M-file to implement the Peters-Wilkinson method to solve sparse linear least-squares problems, $Ax \overset{\text{l.s.}}{=} b$. Use MATLAB functions chol and lu (both with permutation matrices). Compare running times and accuracy (i.e., consider $\|A^T(Ax - b)\|_2$) to the MATLAB "backslash" (or the SPLLS function described in Appendix A). To obtain two test problems type load lls1 and load lls2. The matrix in lls1.mat, is 2000-by-400 with density about .009; the matrix in lls2.mat, is 1600-by-900 with density about .013

9. Solve the problem $\bar{A}x \overset{\text{l.s.}}{=} \bar{b}$, where

$$\bar{A} = \begin{pmatrix} A \\ e^T \end{pmatrix} \quad \bar{b} = \begin{pmatrix} b \\ 1 \end{pmatrix}, \tag{1.10}$$

and $e^T = (1, 1, ..., 1)$; A, b are given in lls1.mat. Aim for a running time less than double the best time obtained in #8.

1.3 The Conjugate Gradient Method

A large square system of linear equations, $Ax = b$, can be solved by an iterative method, thus avoiding the need for a (sparse) factorization of the coefficient matrix A. The use of iterative methods is particularly popular when A is symmetric positive definite, in which case the method of *conjugate gradients* (CG) is generally the most successful approach. The conjugate gradient approach to SPD systems is discussed in many texts, e.g., [1, 19, 18, 4]; therefore, we will treat this subject briefly here with emphasis on pragmatic and optimization-related matters.

1.3.1 Pure Conjugate Gradients

In theory, the method of conjugate gradients is a finite algorithm, requiring at most n iterations to produce the solution to $Ax = b$ where A is a SPD matrix of order n. The proof of finite convergence is based on the expanding subspace property: in the k^{th} iteration CG produces a minimizer of

$$q(x) \overset{def}{=} -b^T x \ + \ \frac{1}{2} x^T A x \tag{1.11}$$

on the k-dimensional subspace spanned by the k conjugate directions[5] generated by CG.

In practise finite convergence generally will not occur: roundoff error causes the conjugacy property to be lost and with loss of conjugacy goes the expanding subspace property. Therefore, the CG method is usually implemented as an infinite process using a stopping tolerance (is the residual sufficiently small?) for termination.

The notation v_-, where v is a vector, refers to the value of v in the previous CG iteration.

Initialize: $r = b,\ x = d_- \leftarrow 0,\ \beta \leftarrow 0.$

While $\|r\| > tol$

1. $d \leftarrow r + \beta d_-\ ,\ \gamma \leftarrow d^T A d$
2. *if* $\gamma \leq 0$, *return*(x, d)
   else $\alpha \leftarrow \frac{r^T r}{\gamma}$, $x \leftarrow x + \alpha d$, $r \leftarrow r - \alpha A d$
3. $\beta \leftarrow \frac{r^T r}{r_-^T r_-}$

return(x)

Figure 1.13: Algorithm CG

A few remarks on Algorithm CG:

(i) If A is SPD then, in principle, γ should be positive. However, the conditional in Step 2. can be useful when there is some uncertainty regarding the inertia of A. If it is determined that A is not positive definite, i.e., $\gamma \leq 0$, then the offending direction d is useful in some optimization settings – see §**??**.

(ii) The positive scalar α is the optimal step length from x, along the direction d, with respect to $q(x)$.

(iii) The vector r is the residual vector, $r \ = \ b - Ax$.

(iv) Access to matrix A is through multiplication with vector d; therefore, in theory an explicit representation of A is not required. A "subroutine" to compute the products Ad is all that is necessary.

[5] Vectors v and w are *conjugate* with respect to A if $v^T A w \ = \ 0$.

(v) The cost of each iteration of CG is the computation of the matrix-vector product Ad plus a couple of inner products and vector additions.

Unfortunately, the number of CG iterations required to satisfy the stopping tolerance can sometimes be unacceptable. This occurs when the eigenvalues of A are well-separated, spanning a wide spectrum. Indeed, CG may be unable to achieve the requested accuracy for for a given problem.

1.3.2 PCG: Preconditioned Conjugate Gradients

CG works best on problems with clustered eigenvalues. Therefore, it is advisable to "precondition" linear systems to yield an equivalent mathematical problem, but with reduced spread of eigenvalues. In principle this preconditioning transformation can be done in the following way.

Let C be a SPD approximation to A such that $C = L \cdot L^T$ is economical to compute. If C is a sufficiently good approximation to A, the matrix

$$\bar{A} \stackrel{def}{=} L^{-1}AL^{-T} \tag{1.12}$$

will be better conditioned than A, i.e., $\bar{A}$ will exhibit better clustering of eigenvalues. Therefore, applying the transformation $y = L^T x$, the original problem (1.11) becomes

$$\min_y \{-\bar{b}^T y + \frac{1}{2} y^T \bar{A} y\} \tag{1.13}$$

where $\bar{A}$ is defined in (1.12) and $\bar{b} \stackrel{def}{=} L^{-1}b$. We can now proceed by applying the CG method to problem (1.13). This idea yields the Preconditioned Conjugate Gradient (PCG) method given in Figure 1.14.

Initialize: $x = d_- \leftarrow 0,\ r \leftarrow b,\ \beta \leftarrow 0,\ z \leftarrow \mathcal{P}(r).$

While $\|z\| > tol$

1. $d \leftarrow z + \beta d_-$, $\gamma \leftarrow d^T A d$
2. *if* $\gamma \leq 0$, *return*(x, d)
   else $\alpha \leftarrow \frac{r^T z}{\gamma}$, $x \leftarrow x + \alpha d$, $r \leftarrow r - \alpha A d$ *endif*
3. $z \leftarrow \mathcal{P}(r)$
4. $\beta \leftarrow \frac{r^T z}{r_-^T z_-}$

return(x)

Figure 1.14: Algorithm PCG

The function $\mathcal{P}$ is the preconditioning operation; typically, this involves a (sparse) symmetric positive definite matrix C approximating A, and the solution of the system: $Cz = r$

There is no magic formula for choosing an effective, and economical, preconditioner C. It is a particularily challenging task when the matrix A is not explicit, perhaps known only through a matrix-vector multiplication subroutine.

When matrix A is known explicitly, there are several standard choices available, e.g., diagonal, banded, and incomplete Cholesky factorizations, e.g., [1, 12]. In some cases effective problem-specific preconditioners can be found.

The method of preconditioned conjugate gradients is available through the MATLAB function PCG.

A few remarks on PCG in Figure 1.14

(i) Different stopping criteria for PCG are possible. A popular (relative) measure has the form

$$\|z\| \ \leq \ \eta \cdot \|b\| \tag{1.14}$$

where η reflects the desired (relative) accuracy. Different norms can be used in 1.14; however, the C-norm[6] has an attractive monotonicity property: $\|x\|_C$ increases with each PCG iteration (See Question 3, in §1.4).

(ii) If PCG is applied to a problem where A is symmetric but not positive definite, then there are two possible outcomes:

(a) The stopping tolerance, e.g., (1.14), is satisfied before negative curvature is discovered. This can happen in the proximity of a saddle point of (1.11).

(b) PCG exits with a direction of non-positive curvature, i.e., $d^TAd \ \leq \ 0$. If this inequality is strict then it follows that d (or -d) is a direction of infinite descent for $q(x)$. (The MATLAB PCG function does *not* return the negative curvature direction d - just the current solution x is returned and a flag indicating A is not positive definite.)

(iii) In the context where b is the (negative) gradient of a nonlinear function $f : \Re^n \to \Re$ and A is the Hessian matrix, $A \ = \ \nabla^2 f$, it is worth noting that the Hessian matrix A need not be formed explicitly to implement PCG. In particular, given a preconditioner, matrix A is accessed only through matrix-vector products. That is, each PCG iteration requires the product $v = Ad = \nabla^2 f \cdot d$ for a different direction d. Clearly v can be approximated directly by finite-differencing the gradient of f, ∇f, along direction d. Alternatively, v can be computed by automatic differentiation (exactly) in time proportional to the work required to evaluate the gradient. In either case this option is attractive when the number of PCG iterations is small but loses its edge (compared to obtaining $A \ = \ \nabla^2 f$ directly) as the number of PCG iteration increases. This is particularly true when $\nabla^2 f$ is sparse and can be computed/approximated at relatively low cost.

1.4 Exercises

1. Prove, by induction, that Algorithm CG generates conjugate directions. Subsequently, use induction to establish the expanding subspace property: in the k^{th} iteration CG produces a minimizer of

$$q(x) \stackrel{def}{=} -b^Tx \ + \ \frac{1}{2}x^TAx$$

[6] If v is an n-vector then $\|v\|_C \ = \ (v^TCv)^{\frac{1}{2}}$.

on the k-dimensional subspace spanned by the k conjugate directions generated by CG.

2. Show that Algorithm PCG (Figure 1.14) is equivalent to algorithm CG applied to (1.13).

3. Prove that $\|x\|_C$ increases with each iteration of Algorithm PCG.

4. There is a large sparse SPD system, $Ax = b$, in file ps3.mat. Construct a table of timings for 4 sparse SPD solvers on this problem. The 4 solvers are PCG with default preconditioner, PCG with tridiagonal preconditioner L, PCG with a 6-band preconditioner L and MATLAB "backslash". For the PCG solves, try 3 stopping tolerances: $\mathsf{tol} = 1e-1, 1e-5, 1e-8$. Comments?

5. Consider solving $Ax = b$ by Algorithm PCG where the matrix A is defined by

$$A \;=\; B^T M^{-1} B$$

where B is sparse, square, and full rank, M is SPD and sparse. In this case matrix A should not be formed explicitly (since it will be a dense matrix and expensive to compute if n is large). In Algorithm PCG define $z = \mathcal{P}(r)$:

$$\begin{pmatrix} B & I & 0 \\ & -I & \tilde{M} \\ & & -B^T \end{pmatrix} \begin{pmatrix} z \\ u \\ v \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ r \end{pmatrix}, \tag{1.15}$$

where $\tilde{M}$ is a (very) sparse SPD approximation to M. Explain why (1.15) might define a successful preconditioning transformation $\mathcal{P}$ (be sure to comment on the positive definiteness question). Can we further replace B (B^T) with an approximation $\tilde{B}$ ($\tilde{B}^T$) in (1.15)? Any restrictions on $\tilde{B}$ ($\tilde{B}^T$)?

Chapter 2

Appendices

2.1 SPLLS: A Sparse Linear Least Squares Solver

SPLLS : A sparse least-squares solver employing the MATLAB sparse QR factorization, $A = QR$.

Purpose

Solve sparse linear least squares problems.

Synopsis

```
X = splls(A, B)
[X, R] = splls(A, B)
[X, R, p] = splls(A, B)

R = splls(A)
[R, p] = splls(A)

X = splls(A, B, R)
```

Parameters

`A`	an $m \times n$ real matrix
`B`	an $m \times r$ real matrix
`R`	an $m \times n$ real upper triangular matrix
`X`	an $n \times r$ real matrix
`p`	a $1 \times n$ real matrix

Description

`X = splls(A, B)` This function returns the least squares solution to `A*X = B` if `A` has full column rank, or a basic solution to `A*X = B` if `A` does not have full column rank. The rank of `A` is determined without numeric column pivoting. If the rank of `A` is k, then the solution `X` has at most k nonzeros per column. In solving `A*X = B`, the sparse QR factorization `A(:,p) = Q*R` is computed, where `Q` is an $m \times m$ orthogonal matrix, `R` is an $m \times n$ upper triangular matrix and `p` contains a column minimum degree ordering of `A` used for preserving the sparsity of `R`. The matrix `Q` is not explicitly computed. Matrices `R` and `p` are not returned.

`[X, R] = splls(A, B)` This function is the same as the function `X = splls(A, B)` except that (1) sparse QR factorization `A = Q*R` is computed, i.e., no column permutation on `A` is performed, and (2) the upper triangular matrix `R`, which satisfies `A = Q*R`, is returned.

`[X, R, p] = splls(A, B)` This function is exactly the same as the function `X = splls(A, B)` except that matrices `R` and `p` are returned.

`R = splls(A)` This function returns the sparse upper triangular matrix `R` that satisfies `A = Q*R`, where `Q` is an $m \times m$ orthogonal matrix which is not explicitly computed.

`[R, p] = splls(A)` This function returns the sparse upper triangular matrix `R` that satisfies `A(:,p) = Q*R`, where `p` contains a column minimum degree ordering of `A` used for preserving the sparsity of `R` and `Q` is an $m \times m$ orthogonal matrix which is not explicitly computed.

`X = splls(A, B, R)` This function returns the least squares solution to `A*X = B` if `A` has full column rank. Let `A = Q*R` be the QR factorization of `A`, where `Q` is an $m \times m$ orthogonal matrix and `R` is an $m \times n$ upper triangular matrix. If the factor `R` is available, this function obtains the least squares solution to `A*X = B` by solving the semi-normal equations `R'*R*X = A'*B`, and then performing two steps of iterative refinement. For example, the following segment of MATLAB code solves a sparse linear least squares problem.

```
[R, p] = splls(A);
Y = splls(A(:,p), B, R);
X(p) = Y;
```

Bibliography

[1] O. Axelsson, *Iterative Solution Methods*, Cambridge University Press, 1994.

[2] Å. Bjorck, *Stability analysis of the method of semi-normal equations for linear least squares problems*, Tech. Rep. CMA-R26-85, Centre for Mathematical Analysis, Australian National University, 1985.

[3] ——, *Numerical Methods for Least Squares Problems*, SIAM, 1996.

[4] T. F. Coleman, *Large Sparse Numerical Optimization*, Lecture Notes in Computer Science, Volume 165, Springer-Verlag, 1984.

[5] T. F. Coleman, A. Edenbrandt, and J. R. Gilbert, *Predicting fill for sparse orthogonal factorization*, J. Assoc. Comput. Mach., 33 (1986), pp. 517–532.

[6] T. F. Coleman and P. E. Plassmann, *Solution of nonlinear least-squares problems on a multiprocessor*, in Parallel Computing 1988, Shell Conference Proceedings, G. van Zee and J. van de Vorst, eds., Springer-Verlag, Lecture Notes in Computer Science, 384, 1989.

[7] T. F. Coleman and C. Sun, *Parallel orthogonal factorizations of large sparse matrices on distributed-memory multiprocessors*, in Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, SIAM, Philadelphia, 1993, pp. 457–461.

[8] I. S. Duff, A. Erisman, and J. K. Reid, *Direct methods for sparse matrices*, Clarendon Press, Oxford UK, 1986.

[9] A. L. Dulmage and N. Mendelsohn, *Coverings of bipartite graphs*, Canadian Journal of Mathematics, 10 (1958), pp. 517–534.

[10] ——, *A structure theory of bipartite graphs of finite exterior dimension*, Trans. Royal Society Can. Sec. III, 53 (1959), pp. 1–13.

[11] ——, *Two algorithms for bipartite graphs*, J. Soc. Ind. Appl. Math., 11 (1963), pp. 183–194.

[12] D. J. Evans, *Preconditioning Methods, Theory and Applications*, Gordon and breach Science Publishers, 1983.

[13] J. A. George and M. T. Heath, *Solution of sparse least squares problems using Givens rotations*, SIAM J. Numer. Anal., 34 (1980), pp. 69–83.

[14] J. A. George and J. W. H. Liu, *A fast implementation of the minimum degree algorithm using quotient graphs*, ACM Trans. on Math. Software, 3 (1980), pp. 337–358.

[15] J. A. George and J. W.-H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[16] J. A. George and J. W. H. Liu, *The evolution of the minimum degree ordering algorithm*, SIAM Review, 31 (1989), pp. 1–19.

[17] J. R. Gilbert, C. Moler, and R. Schreiber, *Sparse matrices in Matlab: Design and implementation*, SIAM Journal on Matrix Analysis and Applications, 13 (1992), pp. 333–356.

[18] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, 1989.

[19] L. A. Hageman and D. M. Young, *Applied Iterative Methods*, Academic Press, Computer Science and Applied Mathematics, 1981.

[20] J. W. H. Liu, *Modification of the minimum degree algorithm by multiple elimination*, ACM Trans. on Math. Software, 11 (1985), pp. 141–153.

[21] ——, *The multifrontal method for sparse matrix solution: Theory and practice*, SIAM Review, (1992), pp. 82–109.

[22] P. Matstoms, *Sparse QR factorization in MATLAB*, ACM Trans. on Math. Software, 20 (1994), pp. 136–159.

[23] G. Peters and J. Wilkinson, *The least-squares problem and pseudo-inverses*, Computer Journal, 13 (1970), pp. 309–316.

[24] A. Pothen and C.-J. Fan, *Computing the block triangular form of a sparse matrix*, ACM Transactions on Mathematical Software, 16 (1990), pp. 303–324.

[25] E. Rothberg and B. Hendrickson, *Sparse matrix ordering methods for parallel sparse Cholesky factorization*, Tech. Rep. TR 96-0475, Sandia National Laboratories, 1996.

[26] S. J. Wright, *Modified Cholesky factorizations in interior-point algorithms for linear programming*, Tech. Rep. MCS-P600-0596, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[27] M. Yannakakis, *Computing minimum fill-in is NP-complete*, SIAM J. Alg. & Disc. Meth., 1 (1981), pp. 77–79.