BIT
2000, Vol. 40, No. 3, pp. 536–558
0006-3835/00/4003-0536 $15.00

AN INCOMPLETE CHOLESKY FACTORIZATION FOR DENSE SYMMETRIC POSITIVE DEFINITE MATRICES *

CHIH-JEN LIN[1] and ROMESH SAIGAL[2] †

[1] *Department of Computer Science and Information Engineering*
National Taiwan University, Taipei 106, Taiwan. email: cjlin@csie.ntu.edu.tw
[2] *Department of Industrial and Operations Engineering, University of Michigan*
Ann Arbor, MI 48109, USA. email: rsaigal@engin.umich.edu

Abstract.

In this paper, we study the use of an incomplete Cholesky factorization (ICF) as a preconditioner for solving dense symmetric positive definite linear systems. This method is suitable for situations where matrices cannot be explicitly stored but each column can be easily computed. Analysis and implementation of this preconditioner are discussed. We test the proposed ICF on randomly generated systems and large matrices from two practical applications: semidefinite programming and support vector machines. Numerical comparison with the diagonal preconditioner is also presented.

AMS subject classification: 46N10, 65F10.

Key words: Incomplete Cholesky factorization, conjugate gradient methods, dense linear systems.

1 Introduction.

Large dense linear systems generally require a prohibitive amount of memory, and thus are very difficult to solve by direct methods. As suggested by Edelman in his survey [7], a modern approach for solving dense linear systems is to use preconditioned iterative methods that access the matrix only by matrix–vector multiplication and employ fast approximate methods for computing the matrix–vector product. In this paper, we focus on the preconditioned conjugate gradient method for dense symmetric positive definite (SPD) systems.

The performance of iterative methods largely depends on the use of good preconditioners which can accelerate the convergence. Several papers [2, 10, 11, 16, 25, 29, 36] have addressed the issue of preconditioning dense matrices. Most of them are for the solution of boundary-integral formulation of partial differential

*Received March 1999. Revised January 2000. Communicated by Robert Schreiber.

†This work was supported in part by the National Science Foundation grant CCR-9321550 and the National Science Council of Taiwan via the grant NSC-88-2213-E-002-097. Part of this work was done while the first author worked for the Mathematics and Computer Science Division, Argonne National Laboratory, supported by U.S. Department of Energy, under Contract W-31-109-Eng-38.

equations with applications in acoustics, elastics, and electromagnetics, etc. For some of these applications, techniques such as multipole methods (for example, [30]) and FFT can be used to speed up the matrix–vector multiplications so efficient iterative methods were implemented. However, many symmetric positive definite matrices from numerical optimization do not have such special structures. Therefore, preconditioning and matrix–vector multiplications are complicated and expensive. In this paper, we will exploit the possibility of using the incomplete Cholesky factorization (ICF) for general dense SPD systems.

For sparse SPD matrices, incomplete Cholesky factorization has been a general way for obtaining preconditioners: Given a symmetric m by m matrix M and a symmetric sparsity pattern S, an incomplete Cholesky factor of M is a lower triangular matrix L such that

$$M = LL^T + R, \quad l_{i,j} = 0 \text{ if } (i,j) \notin S, \quad r_{i,j} = 0 \text{ if } (i,j) \in S.$$

Then we expect the preconditioned matrix $L^{-1}ML^{-T}$ to have a smaller condition number and hence the number of conjugate gradient iterations could be reduced. Note that we do not use the conventional symbol A to represent the matrix but reserve it for future use. In Section 2, we discuss a method for choosing the sparsity pattern S for dense matrices. This method works in situations where matrices cannot be explicitly stored but each column can be easily computed.

We then test the proposed ICF on different problems:

1. Randomly generated dense matrices: M is obtained from MATLAB.
2. Dense matrices from semi-definite programming: $M = (\bar{A}(S^{-1} \otimes X)\bar{A}^T)$, where $\bar{A}, S$, and X can be stored but M becomes fully dense and cannot be explicitly stored.
3. Dense matrices from support vector machines: each component $M_{i,j}$ is a function of two short vectors v_i and v_j. For example, $M_{i,j} = e^{-\|v_i - v_j\|^2}$. All data v_i can be stored but M is fully dense and cannot be explicitly stored.

The experiments are described in Section 3, 4, and 5. For each case, we compare ICF with the diagonal preconditioner. Experiments show that in general ICF helps the conjugate gradient method to take fewer iterations. We also analyze the computational efforts of using the ICF.

Discussions and conclusions are in Section 6. Sections 2 and 4 are based on the preliminary results in Lin [17].

2 Incomplete Cholesky factorization for dense systems.

Based on different orders of loops and indexes, there are several versions of Cholesky factorization (see, for example, the Appendix of [26]). Therefore, even for the same sparsity pattern S, there are different forms of incomplete Cholesky factorization. However, since we do not store the dense matrix but calculate

each element during the factorization, only some special forms are usable. For example, if the popular outer product form [9, Section 10.3.2] is used, in each step, the current sub-matrix

$$\begin{bmatrix} \alpha & v^T \\ v & B \end{bmatrix} \tag{2.1}$$

is considered. Some components of v are made zero to define the vector w, and $w/\sqrt{\alpha}$ is used to generate the factor L. Then the procedure continues to partially factorize the matrix $B - \alpha^{-1}ww^T$. However, since we are not able to store B, $B - \alpha^{-1}ww^T$ is also not stored. This means that we must delay the operation of subtracting $\alpha^{-1}ww^T$ until the respective column is considered.

In other words, if v is the jth column, elements of the jth column of M are not computed until the jth step. The jth column of L is completed by using the 1st to (j–1)st columns of the factor L to update v and obtain w. As it is stored, L must be sparse.

We consider the procedure in Algorithm 2.1 to do the incomplete Cholesky factorization. It is modified from the *jki* version of the Cholesky factorization (see Appendix of [26]). In Algorithm 2.1, entries of M are replaced by entries of L. It can be clearly seen that at the jth step, columns 1 to j–1 of L are used to update the jth column. Note that in this algorithm, the (i, j) component of M is represented as `M(i,j)`.

Algorithm 2.1 (Incomplete Cholesky factorization (ICF)).

```
for j = 1:m
    M(j,j) = sqrt(M(j,j))
    for k = 1:j-1 & M(j,k) ≠ 0
        for i = j+1:m & M(i,k) ≠ 0
            M(i,j) = M(i,j) - M(i,k)*M(j,k)
        end
    end
    for i = j+1:m
        M(i,j) = M(i,j)/M(j,j)
        M(i,i) = M(i,i) - M(i,j)^2
    end
    Find some nonzero elements of the jth column and store.
end
```

A key issue in ICF is to choose the sparsity pattern S. Many methods have been proposed for finding a good S. However, most of them cannot be directly applied to dense matrices. The first ICF proposed by Meijerink and van der Vorst [22] kept the same sparsity pattern, so only elements at nonzero positions of the original matrix are preserved. For the fully dense matrix, this means the whole column v of (2.1) is kept in L so the exact Cholesky factorization is calculated. Since we are able to store neither M nor L, their approach could not be used. Similarly, the level-set approach, first used by Gustafsson [12] and Watts [37], faces the same problem because it essentially generates a denser preconditioner L than the original matrix M.

Another popular ICF method for sparse matrices is based on the drop tolerance approach, (e.g., Munksgaard [24]). In this strategy nonzeroes are included in the incomplete factor when they are larger than some threshold parameter. Therefore, the memory requirements are unpredictable, a situation that is not appropriate for dense matrices.

Kolotilina [16] proposed to use the following rule:

$$(i,j) \in S \quad \text{if} \quad |M_{i,j}| > \epsilon \max_{i,j} |M_{i,j}|,$$

and presented numerical results on some dense systems. As it is difficult to decide the number ϵ, usually a second-stage preconditioning is required, where a second threshold must be decided. In addition, this approach uses only the information of the original matrix but not sub-matrices during the factorization.

Saad [31, 32] and Jones and Plassmann [14] keep only the largest elements (in magnitude) in the preconditioner L. Lin and Moré [18] used this idea and proposed a preconditioner based on Algorithm 2.1. In Algorithm 2.1, the incomplete factor L is calculated column by column. After the jth column is obtained, some of the largest (in magnitude) elements are stored back to L (see the last line of Algorithm 2.1). For sparse matrices, their methods tend to retain more nonzero elements in L than in M. However, these methods can be extended to dense matrices since a sparse L is generated when fewer largest elements are stored. In order to keep L sparse, we select the p ($p \ll m$) largest elements. Hence, the number of nonzero elements of L is $O(pm)$. The number p could be a variable; for example, in [14], the number of nonzeros of each column is used. However, for fully dense matrices, it is difficult to identify which columns are more important as they all have the same number of nonzeros. Therefore, in our implementation a fixed p is used. The ICF by selecting the p largest elements will be the algorithm implemented and tested in the rest of this paper.

The idea of selecting some of the largest elements during factorization is to keep LL^T as close to M as possible, though this is a heuristic. We consider the sub-matrix (2.1); after one step of Cholesky decomposition,

$$\begin{bmatrix} \alpha & v^T \\ v & B \end{bmatrix} = \begin{bmatrix} \alpha^{1/2} \\ \alpha^{-1/2} w \end{bmatrix} \begin{bmatrix} \alpha^{1/2} \\ \alpha^{-1/2} w \end{bmatrix}^T + \begin{bmatrix} 0 & 0 \\ 0 & B - \frac{1}{\alpha} w w^T \end{bmatrix} + E, \tag{2.2}$$

where w is the vector obtained by setting some entries in v to be zero, and

$$E = \begin{bmatrix} 0 & (v-w)^T \\ (v-w) & 0 \end{bmatrix}$$

is the error matrix.

Note that E is not the true error matrix of the whole incomplete factorization because the factorization of $B - \alpha^{-1} w w^T$ is also incomplete. However, by selecting the largest elements (in magnitude), E has the smallest 1-norm. Another way to look at the error is the difference between $B - \alpha^{-1} v v^T$ and $B - \alpha^{-1} w w^T$. It can be easily proved that by selecting the largest elements, their difference is minimized.

Regarding the implementation, we modify the sparse code of Jones and Plassmann [14], and Lin and Moré [18]. The matrix L is stored in compressed sparse column format but diagonal elements are separated in a different array. Note that when using the column format, a difficulty is on accessing the row `M(j,k), k = 1:j-1`. Following the efficient implementation of Jones and Plassmann, we use two additional working arrays of length m to store additional information so that `M(j,k), j = 1:k-1` can be easily obtained. For this implementation, diagonal elements must be stored separately and updated earlier than other off-diagonal elements. This explains why in Algorithm 2.1, diagonal elements `M(i,i), i = j+1:m` are updated immediately after each column has been processed.

The computational work of the incomplete Cholesky factorization includes three parts: the generation of the jth column of the original matrix, the update of the jth column by using the 1st to the $(j-1)$st columns of L, and the selection of the p largest elements. The cost of generating each column depends on different applications. For the second part, from (2.1), column w of L is used to update the rest of the matrix $B - \alpha^{-1}ww^T$. Therefore, the total number of operations is $O(mp^2)$. Note that the practical number of operations depends on different implementations. Appropriate data structures must be used. For our implementation, all matrix elements can be efficiently accessed so the total number of operations of this part is kept as $O(mp^2)$. For the third part, there are many methods to find the p largest nonzero elements. A natural choice is to employ a heap-sort strategy. The cost is $O(k + p \log_2 k)$ for a column of length k. Since $p \ll k$ for dense matrices, the total cost is about $O(\sum_{j=1}^{m} j) = O(m^2)$. If a method similar to the quick split (see [32, Section 10. 4. 3]) is used, the number of operations is $O(k)$ on the average if k is the length of the input array. Therefore the total number of operations is also $O(m^2)$. In Section 3, 4 and 5, we will discuss the practical computational time for our test matrices. Our implementation also requires that the row indices of each column is in ascending order. This costs $O(p \log_2 p)$ for each column. Then the total cost is $O(mp \log_2 p)$ which is smaller then $O(m^2)$ for selecting the p largest elements.

Scaling is a common way to improve the condition of M. In other words, a simple matrix D, usually a diagonal matrix, is obtained, and the ICF works on the scaled matrix $D^{-1/2}MD^{-1/2}$. For example, diagonal scaling ($D_{i,i} = |M_{i,i}|$, $i = 1, \ldots, m$), l_2 norm scaling ($D_{i,i} = \|M$'s ith column$\|_2$, $i = 1, \ldots, m$), and l_∞ norm scaling ($D_{i,i} = \|M$'s i th column$\|_\infty$, $i = 1, \ldots, m$) are reasonable choices. We note that the norms that involve all columns may not be suitable for dense matrices. Because we do not store the matrix, for those norms all columns must be calculated at least twice: once for the scaling matrix D and once during the ICF.

Another difficulty with the incomplete Cholesky factorization is that it may fail even if the original matrix is positive definite [15]. This problem occurs during ICF when a negative diagonal element is encountered. One solution is to replace the negative diagonal element with a positive number. Another approach is the shifted incomplete factorization of Manteuffel [20, 21]. It adds αI to the

original matrix, where I is an identity matrix. Eventually, when α is big enough, the matrix will be diagonally dominant and ICF will succeed. This follows from the fact that diagonally dominant matrices are H-matrices [3] and ICF always succeeds for them. The advantage of the first approach is that only one ICF is required. However, it is difficult to decide the modification of the negative diagonal elements and is usually *too late* to update them. In [18], these two approaches are discussed, and the shifted incomplete factorization appears more stable. Benzi, Kouhia, and Tuma [4] also have the same observation. In addition, [18] also discusses the relation between the scaling matrix D and the number of α. Algorithm 2.2 thus is the actual procedure we propose for calculating ICF of a matrix M.

ALGORITHM 2.2 (INCOMPLETE CHOLESKY FACTORIZATION FOR GENERAL MATRICES).

Choose $\mu > 0$*;*
Compute $\widehat{B} = D^{-1/2} M D^{-1/2}$ *;*
Set $\alpha_0 = 0$*;*
For $k = 0, 1, \ldots,$
    Use Algorithm 2.1 on $\widehat{B}_k = \widehat{B} + \alpha_k I$*; if successful exit;*
    Set $\alpha_{k+1} = \max(2\alpha_k, \mu)$*;*

It can be seen that two major parameters of the ICF are p, the number of nonzero retained, and μ, the initial shift. From the experiments in [18], p should be as large as possible, if the computer memory is available. Note that $O(pm)$ is the storage required by L. Hence one method is to choose a p such that $O(pm)$ is the same order as storage required by other parts of the application. We use this selection for problems in Section 4. The second parameter, the initial shift μ, affects the number of iterations in Algorithm 2.2 to achieve an acceptable shift α where the incomplete factor exists. Therefore, if μ is too small, many iterations may be required. Fortunately our experience shows that even if μ is too small, ICF fails in a very early stage and exits. Therefore, the real effort is not in proportion to the number of iterations. In Section 3, we experimentally demonstrate this property.

3 Testing randomly generated matrices.

In this section, we test the proposed ICF by comparing it with the diagonal preconditioner when solving randomly generated dense symmetric positive definite systems. The computational experiments for this section were done on a Sun UltraSPARC2-300 workstation with 1024 MB RAM. The code is written in ANSI Fortran 77 and compiled with the option -fast, -xO5, -xdepend, -xchip=ultra, -xarch=v8plus, -xsafe=mem (following the recommendations in [8]).

We use the following command of MATLAB (version 5.3) [33] to obtain test matrices:

```
M = sprandsym(3000, 1, 1.0d-6, 2) ;
```

The four parameters are the size, density, approximate reciprocal condition number, and the way to generate positive definite matrices. The matrix is generated by a shift sum of outer products. Unlike in practical large applications where matrix-free methods must be used, here we are able to store the whole matrix. Though we specify the density of matrices to be 1, practically the MATLAB command generates matrices with density around 60%.

Table 3.1: Conjugate gradient iterations for two randomly generated matrices.

	Diag.		ICF ($p = 10m^{1/3}$)				ICF ($p = m^{1/3}$)			
	cgit	cgt	icft	shift	cgit	cgt	icft	shift	cgit	cgt
P1	2236	701.63	5.43	1	727	265.22	3.20	4	1544	493.65
P2	2245	705.35	5.53	1	720	263.58	3.07	4	1521	486.96

The right-hand side of the linear system $Mx = b$ is chosen as the vector of all ones. We then diagonally scale the matrix so the actual system to be solved is $D^{-1/2}MD^{-1/2}x = D^{-1/2}b$. Under the current situation, all diagonal elements are one so diagonal preconditioning is the same as having no preconditioner. Similarly, for the ICF, we do not need to perform the scaling step of Algorithm 2.2.

We use a stopping criterion based on the relative residue

$$\|D^{-1/2}MD^{-1/2}x - D^{-1/2}b\|/\|D^{-1/2}b\| < 10^{-3}.$$

The initial solution of the conjugate gradient method is the zero vector. For the ICF, the parameter μ of Algorithm 2.2 is selected to be 1. In addition, we test two cases by keeping the largest $10m^{1/3}$ and $m^{1/3}$ elements in each column for the ICF. This allows us to investigate the relation between the performance of ICF and the available storage.

Table 3.2: Computational time of ICF using different initial shifts μ.

	$p = m^{1/3}$				$p = 10m^{1/3}$			
	$\mu = 0.001$		$\mu = 1$		$\mu = 0.001$		$\mu = 1$	
	icft	shift	icft	shift	icft	shift	icft	shift
P1	3.35	2.05	3.20	4	15.41	1.02	5.43	1
P2	5.06	4.10	3.07	4	15.89	1.02	5.53	1

In Table 3.1, two randomly generated problems are tested. We report the number of conjugate gradient iterations and the time using the diagonal preconditioner and the ICF in the cgit and cgt columns. For the ICF, the preconditioning time and the final αI added are also presented. They are columns icft and shift. All computational time presented in this paper is in seconds. From Table 3.1 it can be seen that, by using the ICF, the number of iterations is reduced. By comparing the cases of $p = 10m^{1/3}$ and $p = m^{1/3}$, results suggest that more storage is useful in this case. Therefore, we suggest that when using Algorithm 2.2 for dense matrices, as much memory should be allocated as possible. We also notice that when p is reduced, the shifts increase. If more storage

is used, ICF is closer to the real Cholesky factorization; hence less modification of the diagonal elements is required.

In Table 3.2, we compare the computational time of ICF using different initial shifts μ. Though Algorithm 2.2 with $\mu = 1$ takes 10 more iterations than with $\mu = 0.001$, we notice that the time spent by ICF is not 10 times more. If μ is too small, ICF fails in a very early stage and exits.

Because matrices are explicitly stored, the two computationally intensive parts during ICF are incomplete factorization and sorting. "Sorting" includes finding the largest elements (using a heap-sort strategy) and sorting row indices. The percentage of computational time during the ICF for factorization and sorting is as follows:

	factorization	sorting
P1	65%	35%
P2	66%	34%

It shows that factorization is more expensive than finding the largest elements and sorting the row indices. Theoretically, the number of operations for factorization is $O(mp^2) = O(m^{5/3})$ which is less than $O(m^2)$ for finding the largest elements (as shown in Section 2). However, in practice, the factorization takes about $2mp^2 = 200m^{5/3}$ multiplications/divisions and also other operations for maintaining the data structure. Therefore, unless m is extremely large, factorization costs more than sorting. In addition, we observe that compared to finding the largest elements, the time for sorting row indices is minor.

Next we design an experiment to examine the effect of shifts (α_k of Algorithm 2.2). First an $m \times m$ matrix A is generated so that each component is a (uniformly distributed) random number between zero and one. Hence $M_1 = AA^T$ is the first test matrix. Second we obtain a new matrix B with $B_{i,j} = 2A_{i,j} - 1$. Then $M_2 = BB^T$ becomes another test matrix. For the matrix M_1, the diagonal elements are expected to be about $m/3$ and the off-diagonal elements are around $m/4$. This shows that M_1 is far from a diagonally dominant matrix. On the other hand, the diagonal elements of M_2 are still around $m/3$ but the expected values of the off-diagonal elements are zero. However, the expected absolute values of the off-diagonal elements are still $m/4$. Hence by comparing $M_{i,i}$ and $\sum_{j \neq i} |M_{i,j}|$, matrix M_2 is also unlike an diagonally dominant matrix.

Table 3.3 presents results of experiments on two sets of such matrices. The right-hand side b is chosen as the vector of all ones. Therefore, we have made many properties of systems $M_1 x = b_1$ and $M_2 x = b_2$ similar except one possible difference which we would like to investigate: the shifts. There is a rough explanation for why we think the required shift for the matrix M_2 is smaller. If the original matrix is written as the form in (2.1), after one step of Cholesky decomposition, $B - \alpha^{-1} ww^T$ is the sub-matrix which the procedure will continue to work on. As $B - \alpha^{-1} vv^T$ is positive definite, if $B - \alpha^{-1} ww^T$ is closer to it, it is easier to be positive definite. Now for the matrix M_2 the expected values of $\alpha^{-1}(ww^T - vv^T)_{i,j}$ are zeros if $i \neq j$. As $(ww^T/\alpha)_{i,j}, i \neq j$ are also close to zero, both $B - \alpha^{-1} ww^T$ and $B - \alpha^{-1} vv^T$ are not far from B, Thus the procedure may continue more smoothly without facing diagonal elements.

From the information in Table 3.3 we note that by using the diagonal preconditioner, the conjugate gradient method takes about the same number of iterations for solving $M_1 x = b_1$ and $M_2 x = b_2$ though the condition number of M_2 is a little better. However, using ICF, a larger number of CG iterations are needed for solving the system with M_1 than for solving with M_2. We observe that very big shifts, e.g. 64, are taken to generate the incomplete factor of M_2.

This experiment clearly shows that since ICF follows the structure of Cholesky factorization, the problem of negative diagonal elements is unavoidable but sometimes is the key factor contributing to the degradation of the performance. In Section 5, while running a practical application, a similar situation arises. Furthermore, we may suspect that poor performance of ICF is due to the choice of a non-optimal αI. However, our experiments show that even if the best αI for the matrix is used, but it happens to be large, the ICF tends not to perform as well as the preconditioner generated by factorizing $M + \alpha I$. On the other hand, we would like to mention that for this experiment, the CG does not converge if diagonal elements are locally updated (not by adding αI).

Table 3.3: Comparison of the effects of shifts.

		condition	Diag.		ICF ($p = 10m^{1/3}$)			
	m	number	cgit	cgt	icft	shift	cgit	cgt
P1-1	500	1.7371e+10	1142	4.20	0.49	64	1078	9.52
P1-2	500	1.1370e+07	895	3.28	0.27	1	575	5.13
P2-1	500	2.9864e+08	541	2.03	0.49	64	647	5.34
P2-2	500	1.5740e+08	911	3.34	0.27	1	577	5.11

The programs used for numerical tests in this section are available at the authors' homepage.[1]

4 Solving dense linear systems from semidefinite programming.

In this section we study a practical example: semidefinite programming (SDP). Recently SDP has drawn a lot of attention because of its many potential applications. A recent survey is in Vandenberghe and Boyd [34]. In order to solve SDP, a sequence of dense linear systems must be solved.

4.1 Dense matrices from semidefinite programming.

A standard form of the semidefinite programming problem is as follows:

$$\begin{array}{lll} \min & C \bullet X & \\ & A_i \bullet X \;=\; b_i & \text{for every } i = 1, \ldots, m, \\ & X \succeq 0 & \end{array} \tag{4.1}$$

where A_i, $i = 1, \ldots, m$, and C are $n \times n$ symmetric matrices, $A \bullet B = \text{trace}(A^T B)$, $X \succeq 0$ means that X is a symmetric and positive semidefinite matrix ($X \succ 0$

[1] The software is available at `http://www.csie.ntu.edu.tw/~cjlin/icfdense.tar.gz`

means that it is a symmetric and positive definite matrix). The dual of problem (4.1) is as follows.

$$\begin{array}{lll} \max & \sum_{i=1}^m b_i y_i, & \\ & \sum_{i=1}^m A_i y_i \ + \quad S \quad = \ C, & \\ & S \succeq 0. & \end{array} \tag{4.2}$$

Currently, the only effective way to solve problem (4.1) or (4.2) is through interior-point methods. One commonly used interior-point method is an infeasible-start primal-dual path-following algorithm (see, for example, [38, 19]). It starts from an infeasible solution, and, at each iterate (X_k, y_k, S_k), a predictor system (4.3) is solved:

$$\begin{array}{rrcl} A_i \bullet \Delta X_k & & = & -(A_i \bullet X_k - b_i), \quad i = 1, \ldots, m, \\ \sum_{i=1}^m A_i \Delta y_i^k \ + \quad \Delta S_k & & = & -(\sum_{i=1}^m A_i y_i^k + S_k - C), \\ \Delta X_k S_k \ + \ X_k \Delta S_k & & = & t_k I - X_k S_k. \end{array} \tag{4.3}$$

A predicted iterate $(\bar{X}_k, \bar{y}_k, \bar{S}_k)$ is obtained based on the solution of the predictor system. Then a corrector system (4.4) is solved:

$$\begin{array}{rrcl} A_i \bullet \Delta \bar{X}_k & & = & 0, \quad i = 1, \ldots, m, \\ \sum_{i=1}^m A_i \Delta \bar{y}_i^k \ + \quad \Delta \bar{S}_k & & = & 0, \\ \Delta \bar{X}_k S_k \ + \ X_k \Delta \bar{S}_k & & = & \bar{t}_k I - \bar{X}_k \bar{S}_k. \end{array} \tag{4.4}$$

Note that t_k and $\bar{t}_k$ are numbers defined by $X_k, S_k, \bar{X}_k$ and $\bar{S}_k$.

The next iteration is updated by using the information of the current iteration, the predictor direction, and the corrector direction.

Solving both systems (4.3) and (4.4) is the main computational burden of interior-point methods. We use the following three derived equations to solve them:

$$(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)\Delta y_k = h_1 - \sum_{i=1}^m A_i \bullet ((h_3 - X_k h_2)S_k^{-1}), \tag{4.5}$$

$$\Delta S_k = -\sum_{i=1}^m \Delta y_i^k A_i + h_2, \tag{4.6}$$

$$\Delta X_k = (-h_3 - X_k \Delta S_k)S_k^{-1}, \tag{4.7}$$

where h_1, h_2 and h_3 are the right-hand sides of (4.3) and (4.4), $\otimes$ is the Kronecker product,

$$\bar{A} = \begin{bmatrix} \mathsf{vec}(A_1)^T \\ \vdots \\ \mathsf{vec}(A_m)^T \end{bmatrix},$$

$(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$ is an m by m matrix, and

$$(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)_{i,j} = A_i \bullet (X_k A_j S_k^{-1}), \quad i,j = 1,\ldots,m. \tag{4.8}$$

Note that $\mathsf{vec}(A_i)$ denotes the vector formed from the columns of A_i. In (4.8) $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$ is a fully dense matrix, as are X_k and S_k. While the calculation of (4.6) and (4.7) is computationally cheap, obtaining $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$ and solving (4.5) are more expensive.

If A_j are dense, forming $X_k A_j S_k^{-1}$ takes $O(n^3)$ operations and hence $O(mn^3 + m^2n^2)$ operations are required to generate the matrix. This may be more expensive than the $O(m^3)$ operations required to solve (4.5) by Cholesky factorizations. This situation is very different from the case of interior-point methods for linear programming and makes SDP problems more difficult to solve. However, for several applications of SDP, mainly in combinatorial optimization, most A_i are extremely sparse, that is, each A_i contains only a constant number of nonzeroes. Using this property, it is easy to see that generating $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$ requires only $O(m^2)$ operations, since for each i,j, $A_i \bullet (X_k A_j S_k^{-1})$ requires only a constant number of multiplications and additions. Therefore, the computational bottleneck is the effective solution of the fully dense system (4.5); this is where iterative methods are used.

In addition, when each A_i has a constant number of nonzeros, the matrix–vector product $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)v$ has the same computational complexity as $(\bar{A}((S_k^{-1} \otimes I)((I \otimes X_k)(\bar{A}^T v))))$. Therefore, it is possible to calculate the matrix–vector product without storing the matrix. Since the matrix $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$ is positive definite, the conjugate gradient method is a natural choice for solving the system (4.5).

4.2 Stopping criteria for the conjugate gradient method.

As in Section 3, we use the relative residue as the stopping condition for the conjugate gradient method. A common shortcoming of iterative methods is, that unless a very strict stopping criterion is used, they do not produce as accurate solutions as direct methods. However, they take large number of iterations when more accurate solutions are required. We show here that for this special application, it is not necessary to generate very accurate solutions.

SDP is a popular method for obtaining lower bounds for difficult combinatorial optimization problems. Good lower bounds improve the effectiveness of procedures used to solve the original problem. The basic idea is to reformulate a combinatorial problem in SDP standard form (4.1) or (4.2). An SDP relaxation is then solved, thereby obtaining a lower bound of the original problem. A high level of accuracy is not necessary for the following reasons: If the algorithm starts at an infeasible point, both predictor and corrector systems help the iterations approach feasibility. That is,

$$A_i \bullet X_{k+1} - b = \alpha(A_i \bullet X_k - b_i), \quad i = 1,\ldots,m, \tag{4.9}$$

$$A^T y^{k+1} + S_{k+1} - C = \alpha(A^T y^k + S_k - C), \tag{4.10}$$

where $0 < \alpha < 1$. However, even if the solution of (4.5) is not very accurate, this does not affect the dual feasibility because ΔS_k and $\Delta \bar{S}_k$ are directly calculated as

$$-\sum_{i=1}^{m} A_i \Delta y_i^k - (\sum_{i=1}^{m} A_i y_i^k + S_k - C) \quad \text{and} \quad -\sum_{i=1}^{m} A_i \Delta \bar{y}_i^k,$$

respectively. Therefore (4.10) is always numerically correct. On the other hand, primal feasibility will be affected. However, since we are trying to get lower bounds for combinatorial applications, it is sufficient that the algorithm returns a dual solution that is almost dual feasible. The dual objective value of that iteration is a lower bound of SDP relaxation and hence a lower bound of the original application. This observation allows a further reduction in the effort of using iterative methods for solving (4.5).

4.3 Quadratic assignment problems.

Our test problems are SDP relaxations of some quadratic assignment problems (QAP) from the library QAPLIB by Burkard, Karisch, and Rendl [5]. A QAP problem is to assign r plants to r locations so as to minimize the flow-distance-cost. Its standard form is as follows:

$$\begin{aligned} &\min \sum_{i,j,k,l} x_{i,j} D_{i,k} F_{j,l} x_{k,l} \\ &\sum_{j=1}^{r} x_{i,j} = 1, \quad i = 1, \ldots, r, \qquad \sum_{i=1}^{r} x_{i,j} = 1, \quad j = 1, \ldots, r, \\ &x_{i,j} \in \{0, 1\}. \end{aligned} \tag{4.11}$$

$D_{i,k}$ is the distance between location i and k, and $F_{j,l}$ is the material flow between plant j and l. It is an NP-hard problem, and some instances with $r \leq 20$ are not solved yet. Approaches that solve QAP exactly are limited to small problems and usually involve a branch and bound method. Therefore, a relaxation generating good lower bounds can reduce the computational effort by effectively trimming the search tree of branch and bound. With some minor modifications, our SDP relaxation follows from that of Zhao, Karisch, Rendl and Wolkowicz [39]:

$$\min \begin{bmatrix} D \otimes F & 0 \\ 0 & 0 \end{bmatrix} \bullet X, \tag{4.12}$$

$$\begin{bmatrix} A^T A & 0 \\ 0 & 0 \end{bmatrix} \bullet X = 2r, \tag{4.13}$$

$$X_{i,i} - \frac{1}{2} X_{i,r^2+1} - \frac{1}{2} X_{r^2+1,i} = 0, \quad i = 1, \ldots, r^2, \tag{4.14}$$

$$\begin{aligned} &\frac{1}{2} X_{i,j} + \frac{1}{2} X_{j,i} = 0, \quad i = (r-1)k + l, \quad j = (r-1)k + s, \\ &\qquad l \neq s, \quad 1 \leq k, j, s \leq r, \end{aligned} \tag{4.15}$$

$$\frac{1}{2}X_{i,j} + \frac{1}{2}X_{j,i} = 0, \quad i = (r-1)l + k, \quad j = (r-1)s + k, \tag{4.16}$$
$$l \neq s, \quad 1 \leq k, j, s \leq r,$$
$$X_{r^2+1,r^2+1} = 1, \quad X \succeq 0, \tag{4.17}$$

where A is a $2r \times r^2$ matrix which is the coefficient matrix of (4.11).

Note that (4.14)–(4.16) can be easily represented as SDP constraints. Then the solution of the SDP problem (4.12)–(4.17) is a lower bound of the original QAP. In our SDP formulation, $m = r^3 + 2$ and $n = r^2 + 1$. Therefore, both X_k and S_k take $O(n^2) = O(r^4)$ computer memory.

4.4 Implementation issues.

We use an infeasible start predictor–corrector path-following algorithm to solve the primal and dual pair (4.1)–(4.2). The code is modified from an early implementation in Lin and Saigal [19]. At the start, we select a large number $\rho > 0$, and initial solution $X_1 = \rho I$, $y^1 = 0$, and $S_1 = \rho I$. The barrier parameter t_1 is $n^{-1}X_1 \bullet S_1 = \rho^2$. In each iteration a predictor system (4.3) and a corrector system (4.4) are solved.

For predictor direction $(\Delta X_k, \Delta y^k, \Delta S_k)$, we find the largest $\alpha_k^p < 1, \alpha_k^d < 1$ such that

$$X_k + \frac{\alpha_k^p}{2}(\Delta X_k + \Delta X_k^T) \succeq 0, \qquad S_k + \alpha_k^d \Delta S_k \succeq 0$$

and define

$$\begin{aligned} \bar{X}_k &= X_k + \frac{1}{2}0.99\alpha_k^p(\Delta X_k + \Delta X_k^T), \\ \bar{y}^k &= y^k + 0.99\alpha_k^d \Delta y^k, \qquad \bar{S}_k = S_k + 0.99\alpha_k^d \Delta S_k \\ \bar{t}_k &= \frac{\bar{X}_k \bullet \bar{S}_k}{4n}. \end{aligned} \tag{4.18}$$

Then from $(\bar{X}_k, \bar{y}^k, \bar{S}_k)$, we obtain the next iterate $(\bar{X}_{k+1}, \bar{y}^{k+1}, \bar{S}_{k+1})$ by the same procedure as above except that $(\Delta X_k + \Delta X_k^T, \Delta y^k, \Delta S_k)$ is replaced by $(\Delta X_k + \Delta X_k^T + \Delta \bar{X}_k + \Delta \bar{X}_k^T, \Delta y^k + \Delta \bar{y}^k, \Delta S_k + \Delta \bar{S}_k)$. In addition, we use $t_{k+1} = n^{-1}X_{k+1} \bullet S_{k+1}$. We use the stopping criterion of [39]

$$\frac{b^T y^{k+1} - b^T y^k}{1 + |b^T y^{k+1}|} < 10^{-3}$$

for the interior-point algorithm. When the above inequality is satisfied, the duality gap may still be large. However, since we are obtaining lower bounds for QAP and the dual feasibility of SDP relaxation is good, using dual objective values can provide a lower bound of SDP relaxation and hence a lower bound to QAP.

During conjugate gradient iterations, the matrix–vector product

$$(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)v, \tag{4.19}$$

has to be calculated, where v is a working vector. Note that S_k^{-1} appears in (4.19) and the right-hand sides of (4.5) and (4.7). X_k and S_k are fully dense so it is natural to think about obtaining Cholesky factorization of S_k and doing back substitutions when S_k^{-1} is involved. By doing so, the computational errors may be reduced; in particular, some authors (e.g. [1]) have pointed out that several mathematically equivalent ways of calculating the right-hand sides of (4.5) and (4.6) could cause different stability properties. It is easy to avoid the calculation of S_k^{-1} in Equations (4.5) and (4.6). The best way to implement (4.19) is by using

$$\begin{aligned}(\bar{A}(S_k^{-1}\otimes X_k)\bar{A}^T v)_i &= A_i \bullet (X_k \mathsf{mat}(\bar{A}^T v)S_k^{-1})\\ &= \mathsf{vec}(A_i)^T \mathsf{vec}(X_k \mathsf{mat}(\bar{A}^T v)S_k^{-1}), \quad i=1,\dots,m.\end{aligned} \tag{4.20}$$

The mat operator is the reverse of the vec operator so it changes an n^2 vector to a matrix. Since v is a dense vector, $\mathsf{mat}(\bar{A}^T v)$ is also dense. This means dense matrix multiplications of $X_k\mathsf{mat}(\bar{A}v)S_k^{-1}$ is obtained first and then a sparse matrix–vector multiplication is performed. While doing this, the explicit form of S_k^{-1} is also not necessary. However, for doing the ICF, we have to specifically store S_k^{-1}. In order to use $O(m)$ operations to generate the jth column, $A_i \bullet (X_k A_j S_k^{-1})$, $i=1,\dots,m$, are calculated as

$$\sum_{\substack{a,b,c,d:\\ (A_i)_{a,b}\neq 0,(A_j)_{c,d}\neq 0}} (A_i)_{a,b}(X_k)_{a,c}(A_j)_{c,d}(S_k^{-1})_{d,b}, \tag{4.21}$$

because A_i and A_j contain only very few nonzero elements. Hence direct access to elements of S_k^{-1} is essential. This is a drawback of the ICF. Note that (4.21) is not used for all $j=1,\dots,m$. Because there is a dense A_j from (4.13), for only this column, $\bar{A}\mathsf{vec}(X_k A_j S_k^{-1})$ is implemented.

We use the same implementation of the ICF in Section 3. The parameter μ of Algorithm 2.2 is selected to be 1. For the p nonzero elements kept in each column of the preconditioner, we select $10m^{1/3}$ largest elements in order to keep the same order of storage. In other words, $O(m^{4/3})$ storage is taken for the preconditioner; this is the same as $O(r^4)$ when $m=O(r^3)$. We start the conjugate gradient method with the zero vector and stop the iteration when the relative residue

$$\frac{\|(\bar{A}(S_k^{-1}\otimes X_k)\bar{A}^T)\Delta y_k - h_1 + \sum_{i=1}^m A_i \bullet ((h_3 - X_k h_2)S_k^{-1})\|_2}{\| - h_1 + \sum_{i=1}^m A_i \bullet ((h_3 - X_k h_2)S_k^{-1})\|_2} \le \epsilon, \tag{4.22}$$

where $\epsilon = 10^{-4}$ for the predictor step, $\epsilon = 5\cdot 10^{-4}$ for the corrector step, and h_1, h_2 and h_3 are the right-hand sides of (4.5) and (4.6). The main reason for using a larger tolerance for the corrector step is that h_1 and h_2 are both zero and hence the denominator of (4.22) is smaller. The number of CG iterations is limited to be the maximum of 400 and $m^{2/3}$. We have noticed in Section 2 that the l_2 and l_∞ norm scaling may not be suitable for dense matrices. Therefore, the diagonal scaling is implemented.

Table 4.1: Comparisons of running the same linear systems (NUG problems).

iter	$r = 12$		$r = 18$		$r = 20$		$r = 25$	
1	3	3	3	3	3	3	3	3
2	4	4	4	4	4	3	4	3
3	4	4	5	4	5	4	5	3
4	5	5	5	4	5	4	5	4
5	23	62	27	58	34	61	30	55
6	61	119	88	107	83	106	101	130
7	53	33	45	40	48	53	48	76
8	93	36	90	62	100	63	177	99
9	63	34	185	70	59	29	116	59
10	132	55	90	48	82	37	113	158
11	196	186	336	108	401*	161	626*	626*
12	312	169	208	198	283	329	626*	626*
13			240	110	401*	401*	626*	626*
14							278	626*
15							626*	626*
total	949	710	1326	816	1508	1254	3384	3720

*: exceeded the maximal number of CG iterations.

4.5 Numerical results.

Our programs were written in MATLAB. Computationally-intensive parts such as the diagonal preconditioning, the incomplete Cholesky factorization, and the matrix–vector multiplication $\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T v$ were written in Fortran and linked to MATLAB drivers through C subroutines and **mex** scripts.

Before comparing the overall performance, first we compare the number of CG iterations by using the diagonal preconditioner and the ICF on the same linear system. In Tables 4.1 and 4.2, the diagonal preconditioner is used to solve NUG and TAIa problems, both with four sizes; for the same predictor systems, ICF is also used. Then we report the number of CG iterations for solving predictor systems by two preconditioners. For example, when $r = 12$, NUG12 problem is solved and in Table 4.1, two columns show numbers of CG iterations by two preconditioners in each interior-point iteration. Note that the first column iter presents the interior-point iterations. In early iterations, because matrices are well-conditioned and close to diagonal-dominance, the diagonal preconditioner performs better than ICF. In later iterations, matrices are more ill-conditioned, and in general ICF takes fewer CG iterations. For example, fewer CG iterations are taken by using the diagonal preconditioners in the first eight iterations for the problem TAI25a; however, after the ninth iteration, ICF performs better. Hence, the total number of CG iterations is also less. From this experience, we realized it would be better to start the ICF in the middle of the optimization algorithm. The rule implemented is that after the number of CG iterations in one interior-point iteration (either predictor or corrector) reaches more then $0.15r^2$, the incomplete Cholesky factorization is used.

Table 4.2: Comparisons of running the same linear systems (TAIa problems).

iter	$r = 12$		$r = 17$		$r = 20$		$r = 25$	
1	3	3	3	3	3	3	3	3
2	4	4	4	3	5	3	5	4
3	4	4	4	3	5	3	5	4
4	4	4	5	3	5	3	5	4
5	5	4	5	4	5	4	6	5
6	27	51	26	55	26	53	23	35
7	79	98	100	122	121	104	82	164
8	41	32	54	41	57	71	42	125
9	122	54	109	45	123	83	140	68
10	59	26	80	54	401*	217	189	104
11	216	94	81	49	80	60	528	342
12	353	149	145	104	141	126	151	75
13	227	97			151	232	193	161
14								
15								
total	1144	620	616	386	1023	962	1372	1094

*: exceeded the maximal number of CG iterations.

Table 4.3: The incomplete Cholesky factorization and diagonal preconditioner for solving NUG12.

	Diag. Precond.		ICF		
iter	cgit	cgt(sec.)	cgit	icft(sec.)	cgt(sec.)
1-5	88		88		
6	107	23.12	206	6.02	50.83
7	129	27.48	105	6.21	25.50
8	126	26.89	72	9.64	17.84
9	96	20.75	53	9.72	12.79
10	214	45.55	87	9.13	21.16
11	300	64.02	251	8.47	60.66
12	586	125.28	301	11.70	72.82
6–12	1558	333.09	1075	60.89	261.60

Table 4.3 compares the incomplete Cholesky factorization and diagonal preconditioner for solving NUG12. The first column (iter) shows the outer iterations (interior-point iterations). For the diagonal preconditioner, the number of CG iterations and the time of CG are reported in columns cgit and cgt. For the ICF, the time of preconditioning (the icft column) is also reported. For this problem, from the first to the fifth iteration, only the diagonal preconditioner is used. After that, in each outer iteration, the numbers of CG iterations are generally lower for the ICF. However, the last row of the table shows the diagonal preconditioning approach takes about the same time as using the ICF (333.09 seconds vs. 322.49 seconds). For small problems, time spent on calculating the ICF is

Table 4.4: The incomplete Cholesky factorization and diagonal preconditioner for solving TAI25a.

	Diag. Precond.		ICF		
iter	cgit	cgt(sec.)	cgit	icft(sec.)	cgt(sec.)
1-9	616		616		
10	261	5260.79	240	496.20	5041.37
11	787	15850.46	555	844.42	11649.75
12	276	5552.93	146	489.64	3110.13
13	420	8860.04	329	706.07	7221.86
8–13	1744	35524.22	1270	2536.33	27023.11

Table 4.5: Number of CG iterations for the NUG problems.

r	Diag.	ICF	Diag.	ICF	Ratio
12	1646	1163	1558	1075	69.00%
18	2010	1472	1728	1190	68.87%
20	2250	1401	1966	1117	56.82%
25	4922	3929	4606	3613	78.44%

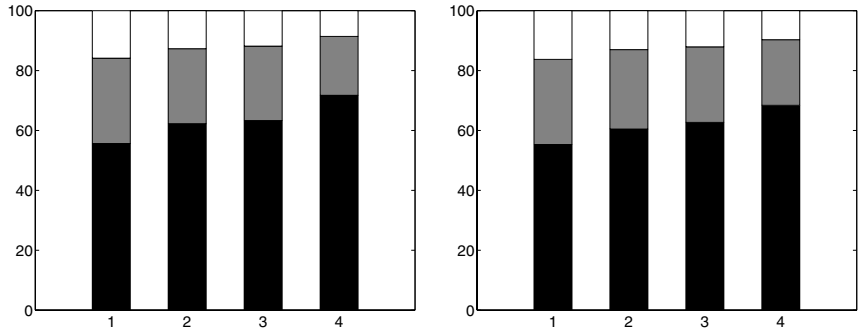
expensive, and hence the benefit of the decrease in CG iterations does not show up.

Table 4.4 presents the same comparison for solving problem TAI25a ($r = 25$). It can be clearly seen that by using the ICF, the total computational time for the preconditioning and CG iterations is less. This result shows us that nontrivial preconditioners such as ICF are useful for large-scale dense linear systems.

We observe consistent results on the decrease in CG iterations when using ICF. Tables 4.5 and 4.6 show the comparison of the number of CG iterations using the two preconditioners. The second and third columns are the total number of CG iterations. In the ICF approach, the diagonal preconditioner is used during the early iterations. Without considering those CG iterations, the comparison is shown in the fourth and fifth columns. Their ratios are shown in the last column. Except NUG25 for which the interior-point algorithm using ICF takes one fewer outer iterations, all other problem instances require the same number of outer iterations by using both preconditioners. For ICF, the total number of CG iterations are less for six of eight problems; in some cases the ratios are very good.

Table 4.6: Number of CG iterations for the TAI problems (series A).

r	Diag.	ICF	Diag.	ICF	Ratio
12	1738	1057	1621	940	57.99%
17	1168	1382	952	1066	111.97%
20	1945	2070	1567	1692	107.98%
25	2360	1886	1744	1270	72.82%



(a) NUG ($r = 12, 18, 20, 25$) (b) TAIa ($r = 12, 17, 20, 25$)

Figure 4.1: Percentage of ICF time (lower: generating matrices, middle: factorization, upper: sorting).

We investigated problems TAI17a and TAI20a, for which the ICF approach takes more CG iterations. Though there is no conclusive explanation yet, we think the problem is the switch to the ICF. Tables 4.1 and 4.2 show that the diagonal preconditioner is better in the early stage of the optimization algorithm. Hence, it is possible that at an iteration, though the diagonal preconditioner is in fact better, the algorithm switches to ICF. Furthermore, once the worse preconditioner is used, we observe that usually the optimization algorithm is also affected. In other words, if the next interior-point iterate is not good because of using the worse preconditioner, the later iterates tend to generate more ill-conditioned matrices. We guess these iterates are further away from the central path. For problem TAI17a, if ICF is used one iteration late, the total number of CG iterations becomes 976 which is smaller than 1168 of using only the diagonal preconditioner. On the other hand, if the ICF is used too late, the improvement of using it becomes minor.

We further analyze the computational time for the ICF. Unlike the results shown in Section 3, we calculate all matrix elements inside the ICF. Figure 4.1 shows the percentage of doing three computational intensive parts: generation of the matrix, factorization, and sorting. Generation of the matrix takes most of the time. The percentage of this part increases as the problem size becomes larger. This is easy to explain. The gap between the generation of the matrix ($O(m^2)$) and the factorization ($O(mp^2) = O(m^{5/3})$) is $O(m^{1/3})$, an increasing function of m.

Another issue which may affect the performance of the ICF is the shift αI added to the original matrix. Table 4.7 shows shifts and CG iterations of predictor (column cgit1) and corrector (column cgit2) steps in the final stage of solving TAI20a and TAI25a. It can be clearly seen that from one iteration to the next iteration, if the shift becomes larger, the number of CG iterations increases and vice versa. Our experience here reconfirms our conclusion that

sometimes the shift itself and not the condition number or other properties of the matrix is the major reason for the large number of CG iterations.

Table 4.7: Shifts of ICF and CG iterations.

	TAI20a			TAI25a		
iter	shift	cgit1	cgit2	shift	cgit1	cgit2
8	8.0	71	96			
9	4.0	105	31			
10	8.0	401	75	4.0	190	50
11	4.0	278	38	16.0	389	166
12	8.0	348	101	4.0	90	56
13	4.0	79	69	8.0	186	143

5 Dense matrices from support vector machines for pattern recognition.

The support vector machine (SVM) is a promising technique of pattern recognition. For surveys of this subject see, for example, Cortes and Vapnik [6], and Vapnik [35]. Given training vectors v_i, $i = 1, \ldots, m$, of length k in two classes, and a vector $a \in R^n$ such that $a_i \in \{1, -1\}$, the support vector technique requires the solution of a quadratic programming problem

$$\begin{aligned} &\min \frac{1}{2} x^T Q x - e^T x, \\ &0 \le x_i \le C, \quad i = 1, \ldots, m, \\ &a^T x = 0, \end{aligned} \tag{5.1}$$

where e is the vector of all ones, C is the upper bound of all variables, Q is a positive semidefinite matrix, and $Q_{i,j}$ is a function of v_i and v_j.

Note that v_i is considered as a support vector if x is the solution of (5.1) and $x_i > 0$. For different problems, there are several choices of $Q_{i,j}$. Here we use

$$Q_{i,j} = a_i a_j e^{-\|v_i - v_j\|_2^2/k} \quad \text{and} \quad a_i a_j \Big(\frac{v_i^T v_j}{k}\Big)^5.$$

It has been shown in [35] that under these definitions of $Q_{i,j}$, Q is positive semidefinite. Solving the quadratic programming problem is the major computational bottleneck of the support vector technique. However, an even more serious problem is that Q is fully dense, so we are not able to store large instances. For many typical pattern recognition applications, the size of training samples are bigger than $10,000$ or even more than $100,000$. Due to this situation, traditional optimization algorithms such as Newton's, Quasi Newton, etc., cannot be directly applied. Several authors (e.g. [27, 13, 28]) have proposed decomposition methods to conquer this difficulty. To be more precise, the training set is separated to two sets B and N, where B is the working set. During each iteration, an optimization subproblem on B is solved and then B and N are

updated. If the size of B is fixed and restricted to be small, data of the whole subproblem could be fully stored.

However, traditional methods like Newton's method have advantages such as rapid local convergence. We are interested in modifying Newton's method for this application, especially combining the decomposition methods and Newton-like methods. This implies that in each iteration we have to solve a dense symmetric positive semidefinite sub-system (bounded variables are not considered). Hence we would like to exploit the possibility of using the preconditioned conjugate gradient method for such systems.

Though for practical optimization procedures, sub-matrices of Q usually are used, we consider for preliminary tests in this section the following linear system:

$$(Q + \alpha I)x = e,$$

where e is the vector of all ones, $\alpha = 0.1$, and I is the identity matrix. We add αI into Q as Q may be only positive semidefinite. We keep the largest $10m^{1/3}$ elements in each column of the incomplete factor. All settings and the experimental environment are the same as in Section 3. The test problems are from the project Statlog [23]. Some of them contain data in more than two classes. Here we consider all elements not in class 1 as in class 2. Remember that a training vector contains k attributes and each of them may be in different ranges. If training data are directly used, it may lead to ill-conditioned quadratic programming problems. Thus scaling the original data is in general necessary. Here we scale each attribute of training vectors to the interval $[-1, 1]$.

In Table 5.1 we report the computational results of using

$$Q_{i,j} = a_i a_j e^{-\|v_i - v_j\|_2^2/k}.$$

The first column shows the size of four problems, ranging from $1,000$ to $15,000$. Other columns have the same meaning as columns in Table 3.1. Table 5.2 contains the results of using

$$Q_{i,j} = a_i a_j \Big(\frac{v_i^T v_j}{k}\Big)^5.$$

We can see that in most cases, ICF requires fewer terations.

Table 5.1: Support vector machine: $Q_{i,j} = a_i a_j e^{-\|v_i - v_j\|_2^2/k}$.

		Diag.		ICF			
Problem	m	cgit	cgt	icft	shift	cgit	cgt
german	1000	41	56.4	4.81	32	36	50.32
dna	2000	19	292	44.4	16	14	215
satimage	4435	31	1030	108	256	35	1170
letter	15000	72	24700	1570	512	58	20100

Comparing the three computationally intensive parts of the ICF, the calculation of $Q_{i,j}$ is very expensive so its percentage is higher (more than 80%). The ratio of the other two parts, factorization and finding the largest elements is about 3 to 1, similar to the observation in Figure 4.1.

Table 5.2: Support vector machine: $Q_{i,j} = a_i a_j \left(\frac{v_i^T v_j + 1}{k}\right)^5$.

		Diag.		ICF			
Problem	m	cgit	cgt	icft	shift	cgit	cgt
german	1000	23	37.2	3.08	1	7	11.5
dna	2000	7	121	19.0	0	3	51.7
satimage	4435	7	273	48.7	0	4	156
letter	15000	11	4410	1260	8	12	4440

6 Discussions and concluding remarks.

In this paper, we study the use of an incomplete Cholesky factorization preconditioner for general dense SPD systems. Numerical results show that, when the size of the system is large, the proposed ICF has the potential to decrease the number of CG iterations as well as the computational time. On the other hand, our experience suggests that preconditioning dense symmetric positive definite matrices is a complicated and difficult issue. Although ICF is suitable for all general matrices, without using any special knowledge of the application, the reduction in iterations and time is not large compared to using the diagonal preconditioner. Naturally, it is arguable whether a 30% reduction in iterations (or time) is useful or not. In addition, except for the matrix itself, many factors affect the applicability of the ICF, e.g., the efficiency of calculating the matrix–vector multiplication in CG and matrix elements in ICF, the required accuracy of the CG, and the number of right-hand sides for the same linear system. In general we think ICF could at least save some iterations. However, the computational time will benefit more if the matrix calculation inside the ICF is cheap. For example, if the generation of one column of the matrix takes about the same time as doing one matrix–vector product, ICF will not be a good choice. On the other hand, if the calculation of the whole matrix takes about the same time as doing one matrix–vector multiplication, the advantage of ICF may show up more. In conclusion, many unsolved issues still need investigations.

Acknowledgments.

The first author thanks Dr. Jorge Moré for introducing him to the subject of incomplete Cholesky factorization. Section 2 is heavily dependent on the joint work [18]. The authors also thank Dr. Michele Benzi for his constructive comments and Stephanie Lindemann for her careful reading of the manuscript.

REFERENCES

1. F. Alizadeh, J.-P. A. Haeberly, and M. L. Overton, *Primal–dual interior-point methods for semidefinite programming: Convergence rates, stability and numerical results*, SIAM J. Optim., 8 (1998), pp. 746–768.
2. G. Alléon, M. Benzi, and L. Giraud, *Sparse approximate inverse preconditioning for dense linear systems arising in computational electromagnetics*, Numer. Algorithms, 16 (1997), pp. 1–15.

3. O. Axelsson, *Iterative Solution Methods*, Cambridge University Press, 1994.
4. M. Benzi, R. Kouhia, and M. Tuma, *An assessment of some preconditioning techniques in shell problems*, Comm. Numer. Methods Engrg., 14 (1998), pp. 897–906.
5. R. Burkard, S. Karisch, and F. Rendl, *QAPLIB—A quadratic assignment problem library*, Tech. Report, Department of Mathematics, Graz University of Technology, Graz, Austria, 1996.
6. C. Cortes and V. Vapnik, *Support-vector network*, Machine Learning, 20 (1995), pp. 273–297.
7. A. Edelman, *Large dense numerical linear algebra in 1993: The parallel computing influence*, International J. Supercomputer Appl., 7 (1993), pp. 113–128.
8. K. Goebel, *Getting more out of your new UltraSPARC^{TM} machine*, Sun Developer News, 1 (1996).
9. G. H. Golub and C. F. Van Loan, *Matrix Computations*, 2nd ed., The Johns Hopkins University Press, Baltimore, MD1989.
10. S. A. Goreinov, E. E. Tyrtyshnikov, and A. Y. Yeremin, *Matrix-free iterative solution strategies for large dense linear systems*, Numer. Linear Algebra Appl., 4 (1997), pp. 273–294.
11. A. Greenbaum, L. Greengard, and G. B. McFadden, *Laplace's equation and the Dirichlet–Neumann map in multiply connected domains*, J. Comp. Phys., 105 (1993), pp. 267–278.
12. I. Gustafsson, *A class of first order factorization methods*, BIT, 18 (1978), pp. 142–156.
13. T. Joachims, *Making large-scale svm learning practical*, in Advances in Kernel Methods—Support Vector Learning, B. Schölkopf, C. J. C. Burges, and A. J. Smola, eds., MIT Press, Cambridge, MA, 1998.
14. M. T. Jones and P. E. Plassmann, *An improved incomplete Cholesky factorization*, ACM Trans. Math. Software, 21 (1995), pp. 5–17.
15. D. S. Kershaw, *The incomplete Cholesky-conjugate gradient method for the iterative solution of system of linear equations*, J. Comp. Phys., 26 (1978), pp. 43–65.
16. L. Y. Kolotilina, *Explicit preconditioning of systems of linear algebraic equations with dense matrices*, J. Sov. Math., 43 (1988), pp. 2566–2573. English translation of a paper first published in *Zapiski Nauchnykh Seminarov Leningradskogo Otdeleniya Matematicheskogo Instituta im. V. A. Steklova AN SSSR*, 154:90–100, 1986.
17. C.-J. Lin, *Preconditioning dense linear systems from large-scale semidefinite programming problems*, in Proceeding of the Fifth Copper Mountain Conference on Iterative Methods, 1998.
18. C.-J. Lin and J. J. Moré, *Incomplete Cholesky factorizations with limited memory*, SIAM J. Sci. Comput., 21 (1999), pp. 24–45.
19. C.-J. Lin and R. Saigal, *An infeasible start predictor corrector method for semi–definite linear programming*, Tech. Report, Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor, MI 48109-2117, 1995.
20. T. A. Manteuffel, *Shifted incomplete Cholesky factorization*, in Sparse Matrix Proceedings, SIAM, Philadelphia, 1979, pp. 41–61.
21. T. A. Manteuffel, *An incomplete factorization technique for positive definite linear systems*, Math. Comp., 34 (1980), pp. 307–327.

22. J. A. Meijerink and H. A. van der Vorst, *An iterative solution method for linear equations systems of which the coefficient matrix is a symmetric M-matrix*, Math. Comp., 31 (1977), pp. 148–162.
23. D. Michie, D. J. Spiegelhalter, and C. C. Taylor, *Machine Learning, Neural and Statistical Classification*, Prentice-Hall, Englewood Cliffs, NJ, 1994. Data available at anonymous ftp: `ftp.ncc.up.pt/pub/statlog/`.
24. N. Munksgaard, *Solving sparse symmetric sets of linear equations by preconditioned conjugate gradients*, ACM Trans. Math. Software, 6 (1980), pp. 206–219.
25. R. Natarajan, *An iterative scheme for dense, complex-symmetric, linear systems in acoustics boundary-element computations*, SIAM J. Sci. Comput., 19 (1998), pp. 1450–1470.
26. J. M. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum Press, New York, 1988.
27. E. Osuna, R. Freund, and F. Girosi, *Training support vector machines: An application to face detection*, in Proceedings of CVPR'97, 1997.
28. J. C. Platt, *Fast training of support vector machines using sequential minimal optimization*, in Advances in Kernel Methods—Support Vector Learning, B. Schölkopf, C. J. C. Burges, and A. J. Smola, eds., MIT Press, Cambridge, MA, 1998.
29. J. Rahola, *Solution of dense systems of linear equations in the discrete-dipole approximation*, SIAM J. Sci. Comput., 17 (1994), pp. 78–89.
30. V. Rokhlin, *Rapid solution of integral equations of classical potential theory*, J. Comput. Phys., 60 (1985), pp. 187–207.
31. Y. Saad, *SPARSKIT: A basic tool kit for sparse matrix computations*, Tech. Report, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, 1994.
32. Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, Boston, MA, 1996.
33. The MathWorks, Inc., *MATLAB User Guide*, Natick, MA 01760-1500, 1998.
34. L. Vandenberghe and S. Boyd, *Semidefinite programming*, SIAM Rev., 38 (1996), pp. 49–95.
35. V. Vapnik, *The Nature of Statistical Learning Theory*, Springer-Verlag, New York, NY, 1995.
36. S. A. Vavasis, *Preconditioning for boundary integral equations*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 905–925.
37. J. W. Watts III, *A conjugate graduent-truncated direct method for the iterative solution of the reservoir simulation pressure equation*, Soc. Pet. Eng. J., 21 (1981), pp. 345–353.
38. S. J. Wright, *Primal–Dual Interior-Point Methods*, SIAM, Philadelphia, PA, 1996.
39. Q. Zhao, S. E. Karisch, F. Rendl, and H. Wolkowicz, *Semidefinite programming relaxations for the quadratic assignment problem*, Tech. Report, Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Ontario, N2L 3G1 Canada, 1996.