An Incomplete Cholesky Factorization for Dense Matrices

Chih-Jen Lin* **and Romesh Saigal**†

Abstract

In this paper, we study the use of an incomplete Cholesky factorization (ICF) as a preconditioner for solving dense positive definite linear systems. 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.

1 Introduction

Large dense linear systems generally require a prohibitive amount of memory, and thus are very difficult to solve by direct methods. The storage of a 30,000 by 30,000 symmetric linear system could require up to 3GB of memory. Such large memory is usually not available on a workstation. Thus the only hope for the solution of these systems on such platforms, is the use of matrix-free 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 preconditioners. A good preconditioner can accelerate the convergence and thus dramatically reduce the number of iterations as well as computational time. 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 equations with applications in acoustics, elastics, and electromagnetics etc. However, in most cases, the linear system is not symmetric positive definite. Unlike systems from partial differential equations, many symmetric positive definite matrices from numerical optimization do not have any special structures. In addition, the requirement to preserve positive definiteness in the preconditioner complicates the preconditioning process. In this paper, we will exploit the possibility of using the incomplete Cholesky factorization (ICF) for general dense SPD systems.

For symmetric positive definite matrices, incomplete Cholesky factorization has been a general way for obtaining preconditioners: Given a symmetric m by m matrix M and a

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.

*Department of Computer Science and Information Engineering, National Taiwan University, Taipei 106, Taiwan (cjlin@csie.ntu.edu.tw). Part of this work was done while the 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.

†Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor, MI 48109 (rsaigal@engin.umich.edu).

symmetric sparsity pattern S, an incomplete Cholesky factor of M is a lower triangular matrix L such that

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

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. There are several iterative methods for solving symmetric positive definite matrices. Here we restrict our discussion to using the conjugate gradient method. 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. Then L is a sparse matrix and can be used during the preconditioned conjugate gradient method. This method works in situations where it is easy to compute a column of the matrix.

In Section 3, some randomly generated dense positive definite systems are tested. We compare the number of conjugate gradient iterations and the time when using the incomplete Cholesky factorization, presented in Section 2, to the case when the diagonal preconditioner is used. Experiments show that for these problems, ICF takes fewer iterations. In addition, we analyze the computational effort required during each step of the ICF.

Though the proposed ICF is suitable for all dense positive definite matrices, it is important to test its validity on practical applications. Unlike the ICF for sparse matrices, we do not expect an order of reduction on the number of iterations; hence some serious testing is necessary to judge its usefulness. In Section 4, we introduce fully dense linear systems that arise in the interior-point procedure for solving semidefinite programming (SDP) problems. The discussion in Section 2 and 4 is based on the preliminary results presented in Lin [17]. The test matrices are obtained from the solution of the SDP relaxation of quadratic assignment problems. The experiments, presented in Section 5, show that when compared to the diagonal preconditioner, there is a reduction in the number of conjugate gradient iterations for most of the cases. Special numerical issues are also discussed in this section.

In Section 6, we discuss another practical application: support vector machines for pattern recognition. Some approaches for this application may require the solution of large dense linear systems. We conduct preliminary tests in this section and discuss the performance of ICF.

The paper presents some discussions and concluding remarks in the last Section 7.

2 Incomplete Cholesky Factorization for Dense Systems

Based on different orders of loops and indexes, traditionally 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 $B-\frac{ww^T}{\alpha}$. However, since we are not able to store B, $B-\frac{ww^T}{\alpha}$ is also not stored. This means that we must delay the operation of subtracting $\frac{ww^T}{\alpha}$ 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)`.

```
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
```

Algorithm 2.1: Incomplete Cholesky factorization (ICF)

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 can not 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:

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

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 [30, 31] 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, it is easier to use a fixed p for dense matrices. 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 just a heuristic, the following observation shows that better approximations may be obtained by selecting the largest elements.

We consider the sub-matrix (2.1); after one step of Cholesky decomposition,

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

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 $B - \frac{ww^T}{\alpha}$ is also incomplete. However, it is natural to let $(v-w)$ contain the smallest elements. By selecting the largest elements (in magnitude), E has the smallest 1-norm. At the current step, this is a good property. Another way to look at the error is the difference between $B - \frac{vv^T}{\alpha}$ and $B - \frac{ww^T}{\alpha}$. If the exact factorization is calculated,

$B - \frac{vv^T}{\alpha}$ is the sub-matrix for the next iteration. Therefore, we expect that if $B - \frac{ww^T}{\alpha}$ is closer to $B - \frac{vv^T}{\alpha}$, the preconditioner is better. Note that

$$(B - \frac{vv^T}{\alpha}) - (B - \frac{ww^T}{\alpha}) = \frac{1}{\alpha}(vv^T - ww^T). \tag{2.2}$$

We will show that if a fixed number of components of v are assigned to zero, $vv^T - ww^T$ has the smallest 1-norm when the largest elements are preserved. Because permutation of columns and rows does not affect the matrix norm, and w is defined by making some components of v to zero, without loss of generality we can assume that $v_1, \ldots, v_p$ are nonzero elements which will be kept in w. Note that we assume, at the current iteration, v and w are both vectors with length n. Then

$$(vv^T - ww^T)_{i,j} = \begin{cases} v_i v_j & \text{if } i > p \text{ or } j > p, \\ 0 & \text{otherwise.} \end{cases}$$

That is, $vv^T - ww^T$ is an n by n matrix and its first p by p sub-matrix is zero. Therefore,

$$\begin{aligned} \|vv^T - ww^T\|_1 &= \max_{i=p+1,\ldots,n} \sum_{j=1}^{n} |v_i v_j| \\ &= (\sum_{j=1}^{n} |v_j|) \max_{i=p+1,\ldots,n} |v_i|. \end{aligned} \tag{2.3}$$

From (2.3), it can be clearly seen that by selecting the largest elements to be $v_1, \ldots, v_p$, $\max_{i=p+1,\ldots,n} |v_i|$ has the smallest value.

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. From (2.1), column w of L is used to update the rest of the matrix $B - \frac{ww^T}{\alpha}$. Therefore, the total number of operations is about $O(mp^2)$. 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 [31, Section 10.4.3]) is used, the number of operation is $O(k)$ in average if k is the length of the input array. Therefore the total number of operations is also $O(m^2)$. In Section 3, 5 and 6, we will discuss the practical computational time for our test matrices. Based on the implementation, after selecting the p largest elements, there may be a need to sort the row (column) indices of L to the ascending order. For example, in [14] and [18], the matrix L is stored in the compressed sparse column format. Hence there is an array which stores row indices of columns of L. Their implementations require that the row indices of each column is in an ascending order. If that is the case, the cost would be $O(p\log_2 p)$ 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 ($D_{i,i} = \|M\text{'s } i\text{th column}\|_2, i = 1, \ldots, m$), and l_∞ norm scaling ($D_{i,i} = \|M\text{'s } i\text{th column}\|_\infty, i = 1, \ldots, m$) are reasonable choices. We note that for those norms that involve all columns may not be suitable for dense matrices. In other words, because we do not store the matrix, 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.

Choose $\mu > 0$.
Compute $\widehat{B} = D^{-1/2}MD^{-1/2}$
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 $\alpha_{k+1} = \max(2\alpha_k, \mu)$

Algorithm 2.2: Incomplete Cholesky factorization for general matrices

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 UltraSPARC10-300 workstation with 256 MB RAM. The code is written in ANSI Fortran 77 and compiled with the option `-fast, -xO5, -xdepend, -xchip=ultra, -xarch=v8plus, -xsafe=mem` (follow the recommendations in [8]).

We use the following command of MATLAB (version 5.2) [33] to obtain test matrices:

```
M = sprandsym(1500, 1, 1.0d-15, 2) ;
```

The four parameters are the size, density, approximate reciprocal condition number, and the way to generate positive definite matrices. According to the manual of MATLAB, 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: Randomly generated matrices

	Diag.		ICF ($p = 10m^{1/3}$)				ICF ($p = m^{1/3}$)			
	cgiter	cgtime	icftime	icfshift	cgiter	cgtime	icftime	icfshift	cgiter	cgtime
Prob1	47	4.80	1.79	1	12	1.50	0.71	4	30	3.11
Prob2	61	6.23	1.76	1	16	2.02	0.68	4	39	4.03
Prob3	49	4.98	1.69	1	13	1.62	0.98	8	33	3.39
Prob4	52	5.26	1.70	1	12	1.49	0.72	4	32	3.30

The right-hand side is chosen such that the solution of $Mx = b$ is 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 the relative residue $\|D^{-1/2}MD^{-1/2}x - D^{-1/2}b\|/\|D^{-1/2}b\| < 10^{-3}$ as the stopping criteria. 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.

In Table 3.1, four problems are tested. We report the conjugate gradient iterations and time by using the diagonal preconditioner and the ICF in the cgiter and cgtime columns. For the ICF, the preconditioning time and the final αI added are also presented. They are columns icftime and icfshift. All computational time presented in this paper is in seconds. From Table 3.1, it can be clearly seen that, by using the ICF, the number of iterations is reduced to less than one third. By comparing the cases of $p = 10m^{1/3}$ and $p = m^{1/3}$, results confirm that more storage leads to better ICF. Therefore, when using the ICF 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. About the computational time, from Table 3.1, ICF is also better. However, unlike ICF which involves all elements of the matrix, the diagonal preconditioner needs only the diagonal part. Therefore, when matrices are not stored and have to be calculated during the preconditioning process, the gap between using the ICF and the diagonal preconditioner may be smaller.

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. Table 3.2 presents the percentage of computational time during the ICF for factorization and sorting. It shows that factorization is more expensive than finding the largest elements and sorting the row indices. Although 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), practically, the factorization takes about $2mp^2 = 200m^{5/3}$ multiplications/divisions which is much larger than m^2. This explains our results in Table 3.2. In addition, we observe that compared to finding the largest elements, the time for sorting row indices is minor.

Table 3.2: Computational time of ICF

	factorization	sorting
Problem1	70.1%	29.9%
Problem2	70.1%	29.9%
Problem3	70.0%	30.0%
Problem4	69.6%	30.4%

We tested ill-conditioned matrices by selecting condition numbers around 10^{15}. We also checked that the condition number of the scaled matrix $D^{-1/2}MD^{-1/2}$ is still around 10^{15} in this case. Hence we know that the new ICF is suitable for preconditioning such matrices. We also tested several well-conditioned matrices with condition numbers 10^1. For these matrices, many fewer CG iterations are required. The difference between the ICF and the diagonal preconditioner becomes smaller, but the ICF is still a little better.

MATLAB can generate a symmetric positive definite matrix by random Jacobi rotation of a positive definite diagonal matrix. The same command is used but the last parameter is different: `M = sprandsym(1500, 1, 1.0d-15, 1)`. Results are similar to what we have shown in Tables 3.1 and 3.2 for experiments on matrices generated by this method.

Next we design an experiment to demonstrate the effect of shifts (α_k of Algorithm 2.2) of the ICF. First an $m \times m$ matrix A is generated so that each component is a (uniformly distributed) random number between zero and one. Hence $M_1 = AA^T$ could be a test matrix because it is symmetric positive definite. 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 so that the solution of the linear system is $[1, -1, 1, -1, \ldots]^T$.

If we select the vector of all ones as the solution, $(M_1x)_i$ is around $m/3 + m(m-1)/4$ but $(M_2x)_i$ is only $m/3$. This will cause a difference when using $\|Mx - b\|/\|b\|$ as the stopping condition of both systems. Therefore, we make many properties of systems $M_1x = b_1$ and $M_2x = b_2$ similar except one possible difference which we would like to investigate: the shifts. There is a rough explanation about 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, $B - ww^T/\alpha$ is the sub-matrix which the procedure will continue to work on. As $B - vv^T/\alpha$ is positive definite, if $B - ww^T/\alpha$ is closer to it, it is easier to be positive definite. Now for the matrix M_2 the expected values of $\frac{1}{\alpha}(ww^T - vv^T)_{i,j}$ are zeros if $i \neq j$. As $(ww^T/\alpha)_{i,j}, i \neq j$ are also close to zero, both $B - ww^T/\alpha$ and $B - vv^T/\alpha$ are not far from B, Thus the procedure may continue more smoothly without facing diagonal elements.

In Table 3.3, columns present the same information as columns in Table 3.1. We note that by using the diagonal preconditioner, the conjugate gradient method takes about the same number of iterations for solving $M_1x = b_1$ and $M_2x = b_2$ though the condition number of M_2 is a little better. However, using ICF, the CG iterations for solving the system with M_1 are more than the iterations 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 the degradation of the performance. In Section 6, while running a practical application, similar situation arises. Furthermore, we may suspect that poor performance of ICF is due to 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 well as the preconditioner is 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 shifts

		condition	Diag.		ICF ($p = 10m^{1/3}$)			
	m	number	cgiter	cgtime	icftime	icfshift	cgiter	cgtime
Prob1-1	500	1.7371e+10	44	0.46	0.54	64	42	0.61
Prob1-2	500	1.1370e+07	44	0.46	0.29	1	28	0.40
Prob2-1	500	1.8610e+09	50	0.52	0.54	64	44	0.63
Prob2-2	500	3.6538e+05	46	0.48	0.29	1	28	0.40

Programs for numerical tests in this section are available at the authors' homepage.[‡]

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

4 Solving Dense Linear Systems from Semidefinite Programming

In the following two sections, we study a practical example: semidefinite programming (SDP). Recently SDP has dawn a lot of attention because of its many potential applications. A recent survey of this topic is available 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:

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

where A_i for $i = 1, \cdots, m$ 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, and $X \succ 0$ means that it is a symmetric and positive definite matrix. The dual of problem (4.1) is as follows.

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

Currently, the only effective way to solve problem (4.1) or (4.2) is through interior-point methods. One of commonly used interior-point methods 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:

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

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:

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

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:

$$
\begin{aligned}
(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)\Delta y_k &= h_1 - \sum_{i=1}^{m} A_i \bullet ((h_3 - X_k h_2)S_k^{-1}) \qquad (4.5)\\
\Delta S_k &= -\sum_{i=1}^{m} \Delta y_i^k A_i + h_2 \qquad (4.6)\\
\Delta X_k &= (-h_3 - X_k \Delta S_k)S_k^{-1}, \qquad (4.7)
\end{aligned}
$$

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

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

Note that $\mathsf{vec}(A_i)$ denotes the vector formed from the columns of A_i.

From (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 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, $X_k A_j S_k^{-1}$ takes $O(n^3)$ operations and hence $O(mn^3 + m^2n^2)$ operations are required to generate the matrix, which may be more expensive than $O(m^3)$ 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, only a constant number of nonzeroes are in each A_i. By using this property, formulating $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$ requires only $O(m^2)$ operations. It is easy to see this result because 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 order 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 of the Conjugate Gradient Method

Similar to 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 combinatorial problems as SDP standard forms (4.1) or (4.2). An SDP relaxation is then solved, thereby obtaining a lower bound of the original problem. The higher 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,

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

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

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

5 Computational Experiments via Testing Semidefinite Programming

5.1 Quadratic Assignment Problems

Our test problems are SDP relaxations of some quadratic assignment problems (QAP) from the library QAPLIB maintained 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:

$$\min \sum_{i,j,k,l} x_{i,j} D_{i,k} F_{j,l} x_{k,l}$$

$$\sum_{j=1}^{r} x_{i,j} = 1, i = 1, \dots, r, \tag{5.1}$$

$$\sum_{i=1}^{r} x_{i,j} = 1, j = 1, \dots, r, \tag{5.2}$$

$$x_{i,j} \in \{0, 1\}$$

$D_{i,k}$ 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].

$$\min \begin{bmatrix} D \otimes F & 0 \\ 0 & 0 \end{bmatrix} \bullet X \tag{5.3}$$

$$\begin{bmatrix} A^T A & 0 \\ 0 & 0 \end{bmatrix} \bullet X = 2r \tag{5.4}$$

$$X_{i,i} - \tfrac{1}{2} X_{i,r^2+1} - \tfrac{1}{2} X_{r^2+1,i} = 0, i = 1, \ldots, r^2 \tag{5.5}$$

$$X_{r^2+1,r^2+1} = 1 \tag{5.6}$$

$$\tfrac{1}{2} X_{i,j} + \tfrac{1}{2} X_{j,i} = 0,\ i = (r-1)k + l, j = (r-1)k + s, l \neq s, 1 \leq k, j, s \leq r \tag{5.7}$$

$$\tfrac{1}{2} X_{i,j} + \tfrac{1}{2} X_{j,i} = 0,\ i = (r-1)l + k, j = (r-1)s + k, l \neq s, 1 \leq k, j, s \leq r \tag{5.8}$$

$$X \succeq 0, \tag{5.9}$$

where A is a $2r \times r^2$ matrix which is the coefficient matrix of (5.1) and (5.2).

Note that Eqs. (5.5)–(5.8) can be easily represented as SDP constraints. Then the solution of the SDP problem (5.3)–(5.9) 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.

5.2 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 $\frac{X_1 \bullet S_1}{n} = \rho^2$. 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

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

and define

$$\begin{array}{rcl} \bar{X}_k & = & X_k + \frac{1}{2} 0.99 \alpha_k^p (\Delta X_k + \Delta X_k^T) \\ \bar{y}^k & = & y^k + 0.99 \alpha_k^d \Delta y^k \\ \bar{S}_k & = & S_k + 0.99 \alpha_k^d \Delta S_k \\ \bar{t}_k & = & \frac{\bar{X}_k \bullet \bar{S}_k}{4n}. \end{array} \tag{5.10}$$

For the corrector direction $(\Delta\bar{X}_k, \Delta\bar{y}^k, \Delta\bar{S}_k)$, we find the largest $\bar{\alpha}_k^p < 1$ and $\bar{\alpha}_k^d < 1$ such that

$$X_k + \frac{\bar{\alpha}_k^p}{2}(\Delta X_k + \Delta X_k^T + \Delta\bar{X}_k + \Delta\bar{X}_k^T) \succeq 0, S_k + \bar{\alpha}_k^d(\Delta S_k + \Delta\bar{S}_k) \succeq 0,$$

and define

$$\begin{array}{rcl} X_{k+1} & = & X_k + \frac{1}{2}0.99 * \bar{\alpha}_k^p(\Delta X_k + \Delta X_k^T + \Delta\bar{X}_k + \Delta\bar{X}_k^T) \\ y^{k+1} & = & y^k + 0.99 * \bar{\alpha}_k^d \Delta\bar{y}^k \\ S_{k+1} & = & S_k + 0.99 * \bar{\alpha}_k^d(\Delta S_k + \Delta\bar{S}_k) \\ t_{k+1} & = & \frac{X_{k+1} \bullet S_{k+1}}{n}. \end{array} \tag{5.11}$$

We use the stopping criteria of [39] for the interior-point algorithm:

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

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 following matrix-vector product has to be calculated:

$$(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)v, \tag{5.12}$$

where v is a working vector.

Note that S_k^{-1} appears in (5.12) 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; especially 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 Eqs. (4.5) and (4.6). For (5.12), the best way to implement is by $\bar{A}\mathsf{vec}(X_k\mathsf{mat}(\bar{A}^T v)S_k^{-1})$ because

$$\begin{array}{rcl} (\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T v)_i & = & A_i \bullet (X_k \mathsf{mat}(\bar{A}^T v)S_k^{-1}) \\ & = & \mathsf{vec}(A_i)^T \mathsf{vec}(X_k \mathsf{mat}(\bar{A}^T v)S_k^{-1}), \quad i = 1, \ldots, m. \end{array} \tag{5.13}$$

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, $A_i \bullet (X_k A_j S_k^{-1}), i = 1, \ldots, m$ are calculated as

$$\sum_{\substack{a,b,c,d:\\(A_i)_{a,b}\neq 0,(A_j)_{c,d}\neq 0}} (A_i)_{a,b}(X_k)_{a,c}(A_j)_{c,d}(S_k^{-1})_{d,b}, \tag{5.14}$$

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 (5.14) is not used for all $j = 1, \ldots, m$. Because there is a dense A_j from (5.4), for only this column, $\bar{A}\mathsf{vec}(X_k A_j S_k^{-1})$ is implemented.

Table 5.1: 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

*: exceed the maximal CG iterations

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

$$\frac{\|(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)\Delta y_k - h_1 + \sum_{i=1}^m A_i \bullet ((h_3 - X_k h_2)S_k^{-1})\|_2}{\| - h_1 + \sum_{i=1}^m A_i \bullet ((h_3 - X_k h_2)S_k^{-1})\|_2} \leq \epsilon, \tag{5.15}$$

where $\epsilon = 10^{-4}$ for the predictor step, $\epsilon = 5 \times 10^{-4}$ 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 (5.15) is smaller. A limit of CG iterations is set to be the maximum of 400 and $m^{2/3}$. We have noticed in Section 2 that l_2 and l_∞ norm scaling may not be suitable for dense matrices. Therefore, the diagonal scaling is implemented.

Table 5.2: 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

*: exceed the maximal CG iterations

5.3 Numerical Results

Table 5.3: NUG12

	Diag. Preconditioner		ICF		
iter	cgiter	cgtime(sec.)	cgiter	icftime(sec.)	cgtime(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

The computational experiments were done on a Sun UltraSPARC2-300 workstation with 1024 MB RAM. 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 (version 5.2) 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 Table 5.1 and 5.2, 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

Table 5.4: TAI25a

	Diag. Preconditioner		ICF		
iter	cgiter	cgtime(sec.)	cgiter	icftime(sec.)	cgtime(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 5.5: 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%

$r = 12$, NUG12 problem is solved and in Table 5.1, 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 5.3 compares the diagonal preconditioner and incomplete Cholesky factorization 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 cgiter and cgtime. For the ICF, the time of preconditioning (the icftime 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 expensive, and hence the benefit of the decrease in CG iterations does not show up.

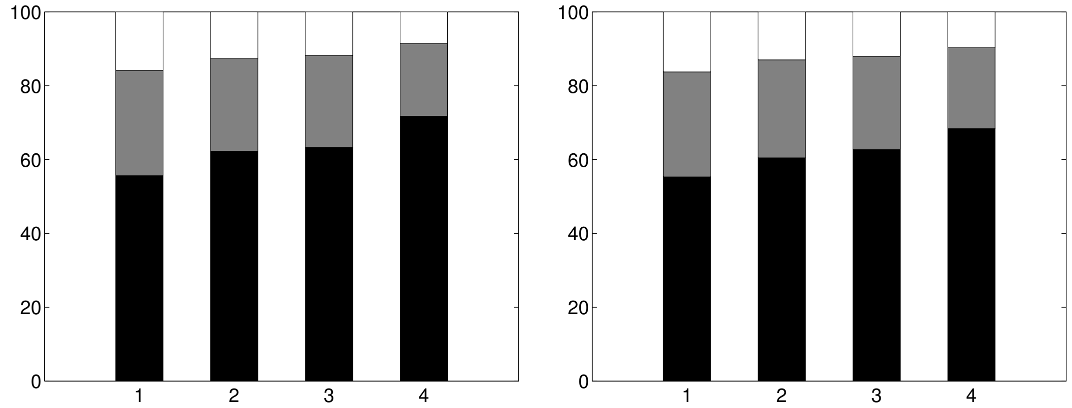
Table 5.4 presents the same comparison for solving problem TAI25a ($r = 25$). It can

Table 5.6: 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%

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 5.5 and 5.6 show the comparison 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.



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

Figure 5.1: Percentage of ICF time (lower: generating matrices, middle: factorization, upper: sorting)

We investigated problems TAI17a and TAI20a, for which ICF approach takes more CG iterations. Though there is no conclusive explanation yet, we think the problem is the switch to the ICF. Tables 5.1 and 5.2 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 use of the ICF is too late, the improvement of using it becomes minor.

We further analyze the computational time for the ICF. Unlike the results shown in Table 3.2, we calculate all matrix elements inside the ICF. Figure 5.1 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 5.7 shows shifts and CG iterations of predictor (column cgiter1) and corrector (column cgiter2) 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 that sometimes the shift itself is the major reason for large number of CG iterations but not the condition number or other properties of the matrix.

We would like to point out that for this special application, on some special architectures, the superiority of the ICF may not show up unless very large systems are solved. The major computational bottleneck of the current approach is on the dense operation $X_k \mathsf{mat}(\bar{A}^T v) S_k^{-1}$ of (5.13); this is why CG iterations are so expensive. On some machines such as IBM RS-6000 workstations, vendors provide specially tuned basic linear algebra subroutines (BLAS). This could dramatically reduce the time of CG iterations. For example, when we do tests on IBM RS-6000 workstations with ESSL package (under the operating system AIX 4.2), the matrix multiplication which ESSL provided is several times faster than the BLAS compiled by users. However, ICF mainly involves sparse operations so it is more difficult to optimize the code. Therefore, the ICF approach cannot compete with the diagonal preconditioning approach unless very large problems are solved.

6 Dense Matrices from Support Vector Machines for Pattern Recognition

The support vector machine (SVM) is a technique of pattern recognition which is used to find important data (support vectors) from training data. In other words, when the

Table 5.7: Shifts of ICF and CG iterations

	TAI20a			TAI25a		
iter	shift	cgiter1	cgiter2	shift	cgiter1	cgiter2
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

number of data points in the training set is huge, instead of using all of them to recognize test data, it is computationally advantageous to select some important ones and use them for classifying test data. Surveys of this subject are, for example, Cortes and Vapnik [6], and Vapnik [35]. Given training vectors $v_i, i = 1, \dots, m$ of length k in two classes, and a vector a defined as follows

$$a_i = \begin{cases} 1 & \text{if } v_i \text{ in class 1} \\ -1 & \text{if } v_i \text{ in class 2,} \end{cases}$$

the support vector technique requires solving a quadratic programming problem:

$$\begin{aligned} \min \quad & \frac{1}{2}x^T Q x - e^T x \\ & 0 \le x_i \le C, i = 1, \dots, m \\ & a^T x = 0, \end{aligned} \tag{6.1}$$

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 (6.1) and $x_i > 0$. For different problems, there are several choices of $Q_{i,j}$. Here we use $Q_{i,j} = a_i a_j e^{-\|v_i - v_j\|_2^2/\gamma}$ and $a_i a_j (\frac{v_i^T v_j}{\gamma})^5$. It has been discussed 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 algorithm such as Newton, 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 method have advantages such as quick local convergence. We are interested in modifying Newton 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. In addition, if the system is only positive *semidefinite* but not positive definite, the performance of the ICF is an interesting issue.

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

$$Qx = Qe,$$

where e is the vector of all ones. 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 6.1, we report the computational results of using $Q_{i,j} = a_i a_j e^{-\|\bar{x}_i - \bar{x}_j\|_2^2/k}$. 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 6.2 is the results of using $Q_{i,j} = a_i a_j (\frac{\bar{x}_i^T \bar{x}_j}{k})^5$. We analyze the condition of these matrices. Due to the memory restriction, we only obtain information of problems german and dna. The condition number of german in Table 6.1 is 8.5003×10^7 but is 1.8293×10^6 in Table 6.2. However, for both Tables 6.1 and 6.2, dna matrices are positive semidefinite but not positive definite. The shifts required by the ICF vary in Tables 6.1 and 6.2. It can be clearly seen that if the shift is smaller, ICF outperforms the diagonal preconditioner. Otherwise, the ICF approach requires the same or a little more iterations. This is consistent with the observation in Section 3.

Table 6.1: Support vector machine: $Q_{i,j} = a_i a_j e^{-\|v_i - v_j\|_2^2/k}$

		Diag.		ICF			
Problem	m	cgiter	cgtime	icftime	icfshift	cgiter	cgtime
german	1000	11	14.5	3.29	32	9	12.05
dna	2000	6	137.6	41.8	16	7	160.8
satimage	4435	4	156.9	118	256	4	157.3
letter	15000	3	1033	832	256	4	1380

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

Table 6.2: Support vector machine: $Q_{i,j} = a_i a_j (\frac{v_i^T v_j + 1}{k})^5$

		Diag.		ICF			
Problem	m	cgiter	cgtime	icftime	icfshift	cgiter	cgtime
german	1000	22	37.4	3.51	4	14	24.0
dna	2000	9	219.6	24.6	1	6	146.6
satimage	4435	14	658	102	128	14	659
letter	15000	15	6363	1050	128	12	5057

two parts, factorization and finding the largest elements is about 3 to 1, similar to the observation in Figure 5.1.

7 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 matrices 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 it is a complicated and difficult issue on preconditioning dense symmetric positive matrices. As ICF is suitable for all general matrices without using any special knowledge of applications, the reduction on iterations and time is not large (comparing to using the diagonal preconditioner). Naturally it is arguable whether a 30% reduction on iterations (or time) is useful or not. In addition, except the matrix itself, many factors affect the applicability of the ICF. Some of them are like 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 systems. In general we think ICF could at least reduce 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 costs 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 is about the same as doing one matrix-vector multiplication, the advantage of ICF may show up more. Overall speaking, preconditioning dense matrices is a very difficult issue; and many unsolved issues still need investigation.

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(3):746–768, 1998.

[2] G. Alléon, M. Benzi, and L. Giraud. Sparse approximate inverse preconditioning for dense linear systems arising in computational electromagnetics. *Numer. Algorithms*, 16:1–15, 1997.

[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.*, 14:897–906, 1998.

[5] R. Burkard, S. Karisch, and F. Rendl. QAPLIB - a quadratic assignment problem library. Technical report, Department of Mathematics, Graz University of Technology, Graz, Austria, 1996.

[6] C. Cortes and V. Vapnik. Support-vector network. *Machine Learning*, 20:273–297, 1995.

[7] A. Edelman. Large dense numerical linear algebra in 1993: The parallel computing influence. *International J. Supercomputer Appl.*, 7:113–128, 1993.

[8] K. Goebel. Getting more out of your new UltraSPARC$^{\mathrm{TM}}$ machine. *Sun Developer News*, 1, 1996.

[9] G. H. Golub and C. F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, second edition, 1989.

[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(4):273–294, 1997.

[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(2):267–278, 1993.

[12] I. Gustafsson. A class of first order factorization methods. *BIT*, 18:142–156, 1978.

[13] T. Joachims. *Making large-scale SVM learning practical*, chapter 11. In Schölkopf et al. [32], 1998.

[14] M. T. Jones and P. E. Plassmann. An improved incomplete Cholesky factorization. *ACM Trans. Math. Software*, 21:5–17, 1995.

[15] D. S. Kershaw. The incomplete Cholesky-conjugate gradient method for the iterative solution of system of linear equations. *J. Comp. Phys.*, 26:43–65, 1978.

[16] L. Y. Kolotilina. Explicit preconditioning of systems of linear algebraic equations with dense matrices. *J. Sov. Math.*, 43:2566–2573, 1988. 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. Preprint MCS-P682-0897, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, 1997. To appear in SIAM Journal on Scientific Computing.

[19] C.-J. Lin and R. Saigal. An infeasible start predictor corrector method for semi–definite linear programming. Technical 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*, pages 41–61. SIAM, Philadelphia, 1979.

[21] T. A. Manteuffel. An incomplete factorization technique for positive definite linear systems. *Math. Comp.*, 34:307–327, 1980.

[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.*, 31:148–162, 1977.

[23] D. Michie, D. J. Spiegelhalter, and C. C. Taylor. *Machine Learning, Neural and Statistical Classification.* Prentice Hall, Englewood Cliffs, N.J., 1994. 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*, 6:206–219, 1980.

[25] R. Natarajan. An iterative scheme for dense, complex-symmetric, linear systems in acoustics boundary-element computations. *SIAM J. Sci. Comput.*, 19(5):1450–1470, 1998.

[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*, chapter 12. In Schölkopf et al. [32], 1998.

[29] J. Rahola. Solution of dense systems of linear equations in the discrete-dipole approximation. *SIAM J. Sci. Comput.*, 17:78–89, 1994.

[30] Y. Saad. SPARSKIT : a basic tool kit for sparse matrix computations. Technical report, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, 1994.

[31] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, Boston, 1996.

[32] B. Schölkopf, C. J. C. Burges, and A. J. Smola, editors. *Advances in Kernel Methods - Support Vector Learning.* MIT Press, Cambridge, USA, 1998.

[33] The MathWorks, Inc. *MATLAB User Guide.* Natick, MA 01760-1500, 1998.

[34] L. Vandenberghe and S. Boyd. Semidefinite programming. *SIAM Rev.*, 38:49–95, 1996.

[35] V. Vapnik. *The Nature of Statisical Learning Theory.* Springer-Verlag, New York, 1995.

[36] S. A. Vavasis. Preconditioning for boundary integral equations. *SIAM J. Matrix Anal. Appl.*, 13:905–925, 1992.

[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.*, 21:345–353, 1981.

[38] S. J. Wright. *Primal-Dual Interior-Point Methods.* SIAM, 1996.

[39] Q. Zhao, S. E. Karisch, F. Rendl, and H. Wolkowicz. Semidefinite programming relaxations for the quadratic assignment problem. Technical report, University of Waterloo, Department of Combinatorics and Optimization, Waterloo, Ontario, N2L 3G1 Canada, 1996.