Incomplete Factorization Preconditioners for Least Squares and Linear and Quadratic Programming

by

Jelena Sirovljevic

B.Sc., The University of British Columbia, 2005

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University of British Columbia

August 31, 2007

Abstract

Many algorithms for optimization are based on solving a sequence of symmetric indefinite linear systems. These systems are often large and sparse, and the main approaches to solving them are based on direct factorization or iterative Krylov-based methods. In this thesis, we explore how incomplete sparse factorizations can be used as preconditioners for the special case of quasi-definite linear systems that arise in regularized linear and quadratic programming, and the case of least-squares reformulations of these systems.

We describe two types of incomplete factorizations for use as preconditioners. The first is based on an incomplete Cholesky-like factorization. The second is based on an incomplete Householder QR factorization. Our approximate factorizations allow the user to prescribe the amount of fill-in, and therefore have predictable storage requirements. We use these incomplete factorizations as preconditioners for SYMMLQ and LSQR, respectively, and present numerical results for matrices that arise within an interior-point context.

Contents

List of Tables

List of Figures

Acknowledgements

There is a small group of people who have, each in their own way, contributed significantly to all the work that went into writing of this thesis. Even though my gratitude to them cannot be adequately expressed in words, I will make an attempt here.

I would like to thank my supervisor Michael Friedlander who provided not only excellent academic support but also constant encouragement and motivation throughout my research. Many thanks also to Chen Greif for reading the thesis and providing a lot of insightful comments.

I am truly grateful to my parents and my sister, without whom I would certainly not be who I am nor where I am today, for their unwavering support and faith in me. All of my accomplishments are theirs as well.

I am very thankful to Thomas for shining a ray of optimism on everything I do, and for always believing that I can do whatever I set my mind to. His enthusiasm and ambition provided a great inspiration for the past two years. And, of course, the daily page count of my thesis was very helpful.

To my best friends, Jelena and Ksenija, I am thankful for never giving up on me, even during the times when it seemed that I will never leave the lab again. I can only hope to be there for them the way they have done for me.

I had the opportunity to meet some great people and gain a few very good friends in the past two years. Liz, who was my friend in combat through many class projects and who provided daily entertainment in the lab; Derek, Chris and Mike with whom even doing homework was a fun activity, and others who will always be a part of fond memories. Past two years would have not been the same without them.

I would also like to thank Ewout and the rest of SCL students for many educational and riveting discussions, and for all the answers to my endless list questions.

Chapter 1

Optimization and Linear Algebra

Solving a linear system of equations,

$$Ax = b,$$

is a fundamental problem in computational mathematics. In many areas of science, such as chemical and electrical engineering, very complicated processes can be modeled as relatively simple linear systems. Thus, solving this problem efficiently is essential for computational mathematics and other areas of science.

There are two distinct approaches to solving a linear system of equations. The first is direct, and uses factorization methods such as Gaussian elimination. The second is iterative and it relies only on matrix-vector product; it is based on computing a sequence of increasingly accurate solution estimates. Examples of iterative approaches include Krylov-based methods such as conjugate gradient. The approximations in these methods either converge to the solution within some specified tolerance or the method reaches the maximum number of iterations specified, and stops without converging.

When dealing with large sparse linear systems storage requirements are critical. Direct methods often introduce a large amount of fill-in entries. These are the entries that were structurally not present in the original matrix, but are introduced in the process of computing a solution with a direct method. In this case, storage becomes an issue and the direct methods become infeasible. Therefore, iterative methods are necessary.

1.1 Background

The effectiveness of iterative solvers depends on the eigenvalue distribution of the system to be solved. The purpose of a preconditioner is to make the eigenvalues of the preconditioned matrix better clustered than they were in the original matrix. The prefect preconditioner would be A^{-1}, where A is the original matrix, because $A^{-1}A = I$ which only has a single eigenvalue. In this case, a conjugate gradient type method would converge in a single iteration. Of course, computing A^{-1} is equivalent to solving the original system using a direct method and defeats the purpose of using an iterative solver. Therefore, a good preconditioner must satisfy two important and somewhat contradictory criteria: it must be close to A^{-1} and easy to compute.

Computing a factorization of A and using the factors as preconditioners is a common practice [25]. The problems arise when dealing with very large sparse systems where the factorization is expensive and the factors are large and difficult to store. The factors often contain large amounts of fill-in entries, and thus require a lot more storage than the original matrix. In order to preserve the sparsity, incomplete factorizations are developed which allow the user to prescribe the amount of storage for each factor and reduce the computation time. These incomplete factors are then used as preconditioners for iterative methods and are thus called *incomplete factorization (IF) preconditioners.*

The purpose of this work is to develop IF preconditioners for two common problem classes that arise in optimization. The first is the least squares problem

$$\underset{x}{\text{minimize}} \quad \|Ax - b\|_2, \tag{1.1}$$

for which we develop an incomplete QR preconditioner. (Henceforth, we use $\|\cdot\|$ to denote the 2-norm of a vector.) The second problem class is linear and quadratic programming,

$$\begin{array}{ll} \underset{x}{\text{minimize}} & c^T x + \frac{1}{2} x^T Q x \\ \text{subject to} & Ax = b,\ x \geq 0, \end{array} \tag{1.2}$$

for which we develop an incomplete Cholesky UDU^T preconditioner in the context of interior-point algorithms. Problem (1.2) is a linear program (LP) if $Q = 0$, and is a convex quadratic program (QP) if Q is symmetric semi definite.

1.2 Optimization Problems

Least-squares problems and LPs and QPs represent a large class of important applications and arise in many areas of science and engineering. They also arise as vital subproblems within algorithms for more general problems.

1.2.1 Least Squares

The least-squares approach solves (1.1) by minimizing the norm of the residual $r = b - Ax$, where A is an m-by-n matrix and b is a vector of length m. Typically, $m \gg n$.

If the rank of the matrix A is n then the solution to the least squares problem is unique. Otherwise there are infinitely many solutions of which only one minimizes the 2-norm of x and this is usually the preferred solution [4, Chapter 1].

Least squares problems are often regularized to improve their conditioning. Tikhonov regularization is the most common approach, and it leads to the problem

$$\underset{x}{\text{minimize}} \quad \tfrac{1}{2}\|Ax - b\|^2 + \tfrac{1}{2}\delta^2\|x\|^2, \tag{1.3}$$

where the δ is a nonnegative parameter that weights the strength of the regularization function.

The necessary and sufficient optimality conditions for the regularized least squares problem (1.3) lead to the linear system

$$\begin{bmatrix} I & A \\ A^T & -\delta I \end{bmatrix} \begin{bmatrix} r \\ x \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}.$$

In case where $\delta = 0$, we have the linear system for the unregularized least-squares problem.

1.2.2 Linear and Quadratic Programming

Interior-point (IP) methods for (1.2) provide a very powerful way of solving LPs and QPs [30]. Each iteration of an infeasible primal-dual IP method is based on applying Newton's method to the perturbed optimality conditions,

$$\begin{aligned} -Qx + A^T y + z &= c \\ Ax &= b \\ XZe &= \mu e \\ (x, z) &> 0, \end{aligned}$$

where μ is a small parameter that tends to zero. In the notation above, X and Z are diagonal matrices, $X = \text{diag}(x_1, x_2, \ldots, x_n)$ and $Z = \text{diag}(z_1, z_2, \ldots, z_n)$; e is an n-vector of all ones. Thus, the condition $XZe = \mu e$ is the same as $x_i z_i = \mu$ for $i = 1, 2, \ldots, n$. Because $\mu \to 0$, this condition eventually becomes $x_i z_i = 0$ for $i = 1, 2, \ldots, n$, which means that for each i one of x_i or z_i must be equal to zero.

Thus, each iteration of an IP method for QPs computes a search direction $(\Delta x, \Delta y, \Delta z)$ as a solution of the linear system

$$\begin{bmatrix} -Q & A^T & I \\ A & 0 & 0 \\ Z & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} c - A^T y \\ b - Ax \\ Xz - \mu e \end{bmatrix}.$$

One way of computing the search direction is by first eliminating the variable Δz and reducing the above 3-by-3 system to the following 2-by-2 system

$$\begin{bmatrix} -H & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}, \tag{1.4}$$

where $H = Q + X^{-1}Z$, $f_1 = c - A^T y - z$, and $f_2 = b - Ax$. The matrix H is symmetric positive definite because Q is symmetric semi definite and X and Z are both diagonal with positive entries.

Similar to the least squares problem, LPs and QPs can also be regularized to improve the conditioning of the linear systems that underly IP methods. A

primal-dual regularization of the QP leads to the problem

$$\begin{array}{ll} \underset{x}{\text{minimize}} & c^T x + \frac{1}{2} x^T Q x + \frac{1}{2}\rho\|x\|^2 + \frac{1}{2}\delta\|r\|^2 \\ \text{subject to} & Ax + \delta r = b,\ x \geq 0 \end{array}$$

where the term $\frac{1}{2}\rho\|x\|^2$ is the primal regularization and the term $\frac{1}{2}\delta\|r\|^2$ is the dual regularization. The nonnegative scalars ρ and δ are primal and dual regularization parameters respectively.

With regularization the augmented system to be solved inside the IP method is

$$\begin{bmatrix} -H - \rho I & A^T \\ A & \delta I \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$$

where H is as defined in (1.4), and f_1 and f_2 are suitably defined. This system belongs to a special category of linear systems, discussed in Section 1.3.1, called symmetric quasi-definite systems. Regularized systems are the focus of this work.

1.3 Linear Algebra and Preconditioning

Preconditioning is a valuable tool within optimization because it allows us to effectively use iterative methods to solve the underlying systems. Some elementary linear algebra theory as well as some more recent ideas are necessary to understand all the pieces in the complex process of computing a good quality preconditioner.

1.3.1 Symmetric Quasi-Definite Systems

Symmetric quasi-definite (SQD) systems have the following block structure

$$K = \begin{bmatrix} -E & A^T \\ A & F \end{bmatrix},$$

where both E and F are symmetric positive definite matrices. These systems have a characteristic property of being strongly factorizable, which means that any symmetric permutation of K yields a matrix which is factorizable, i.e., for any permutation P, $PKP^T = U^T D U$ [28]. This is particularly favorable when dealing with large sparse matrices, as is the case here, since fill-reducing permutations can be applied. Moreover, the permutation P can be chosen solely on the grounds of sparsity instead of numerical stability.

As we discussed in the previous section these systems arise, among other places, in two important classes of optimization problems, least squares and LPs and QPs and solving them efficiently leads to improvement to the solutions to these problems.

It is important to note that if either $E = 0$ or $F = 0$, the system is not strongly factorizable. For example the factorization

$$\begin{bmatrix} -\varepsilon & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & \\ -\frac{1}{\varepsilon} & 1 \end{bmatrix} \begin{bmatrix} -\varepsilon & \\ & \frac{1}{\varepsilon} \end{bmatrix} \begin{bmatrix} 1 & -\frac{1}{\varepsilon} \\ & 1 \end{bmatrix} \tag{1.5}$$

is clearly possible, but the matrix on the left is not strongly factorizable since the symmetric permutation

$$\begin{bmatrix} 0 & 1 \\ 1 & -\varepsilon \end{bmatrix}$$

is not factorizable. On the other hand, if the $(2,2)$ block in the matrix on the left in (1.5) is positive then the linear system is strongly factorizable because

$$\begin{bmatrix} -\varepsilon_1 & 1 \\ 1 & \varepsilon_2 \end{bmatrix} = \begin{bmatrix} 1 & \\ -\frac{1}{\varepsilon_1} & 1 \end{bmatrix} \begin{bmatrix} -\varepsilon_1 & \\ & \varepsilon_2 + \frac{1}{\varepsilon_1} \end{bmatrix} \begin{bmatrix} 1 & -\frac{1}{\varepsilon_1} \\ & 1 \end{bmatrix}, \tag{1.6}$$

and for the symmetric permutation

$$\begin{bmatrix} \varepsilon_2 & 1 \\ 1 & -\varepsilon_1 \end{bmatrix} = \begin{bmatrix} 1 & \\ \frac{1}{\varepsilon_2} & 1 \end{bmatrix} \begin{bmatrix} \varepsilon_2 & \\ & -\varepsilon_1 - \frac{1}{\varepsilon_2} \end{bmatrix} \begin{bmatrix} 1 & \frac{1}{\varepsilon_2} \\ & 1 \end{bmatrix}.$$

In the case when $E = 0$ or $F = 0$, a factorization $U^T B U$ where B is diagonal with 1-by-1 and 2-by-2 blocks, is possible but it is much more expensive than the above $U^T D U$ factorization of an SQD matrix. Thus, we see the savings accomplished by introducing a regularization term in linear and quadratic programming, described earlier.

Stability of this factorization for K depends on the value

$$\theta(K) \equiv \left(\frac{\|A\|_2}{\max(\|E\|_2, \|F\|_2)} \right)^2 \max(\kappa_2(E), \kappa_2(F)),$$

where $\kappa_2(E) = \|E\|_2 \|E^{-1}\|_2$ is the spectral condition number. In other words, if $\theta(K)$ is not too large, the factorization is stable, as shown by Gill et al. [12]. This can be generalized to say that the factorization is stable if $\|A\|_2$ is not too large compared to both $\|E\|_2$ and $\|F\|_2$, and that the diagonals of both E and F are not too ill-conditioned [12, Section 4].

In the example given in (1.6), $\|A\| = 1$, $\|E\| = \varepsilon_1$, and $\|F\| = \varepsilon_2$. Thus, $\kappa_2(E) = \kappa_2(F) = 1$ and the formula for $\theta(K)$ simplifies to

$$\theta(K) \equiv \frac{1}{\max(\varepsilon_1, \varepsilon_2)^2}.$$

Therefore, if ε_1 and ε_2 are not much smaller than 1, the method is stable.

1.3.2 Gaussian Elimination

The attractiveness of the Gaussian elimination comes from the fact that virtually no constraints are put on the matrix being decomposed, other than nonsingularity. In the process of Gaussian elimination entries in the lower triangular part of the matrix A are zeroed out or eliminated and the end result is the upper triangular matrix U. The operations involved in the reduction are all multiplied together to produce a lower triangular matrix L. This is why Gaussian elimination is also known as the LU factorization. Typically, L is constrained to have unit diagonal elements, and in this case, the factorization is unique. If A is nonsingular, then there exists a permutation P such that $PA = LU$.

1.3.3 Cholesky Factorization

Cholesky factorization can be computed as

$$A = R^T R = U^T D U, \tag{1.7}$$

where U is a unit upper triangular, and D is diagonal. This is a special case of Gaussian elimination for symmetric positive definite (SPD) matrices A which satisfy the property $x^T Ax > 0$ for any nonzero vector x. This property of SPD matrices allows for specialized Gaussian elimination which cuts the required work and storage in half. Because A is symmetric, the same matrix that is applied on the left to zero out lower triangular part of a column can be applied on the right in order to zero out the upper triangular part of the corresponding row. This process is repeated for each column-row pair, and the end result is an identity matrix. The matrices applied on the left and the right are all upper triangular and multiplying them together produces the upper triangular Cholesky factor R such that $A = R^T R$; the diagonal matrix D can be introduced to derive the right-hand factorization of (1.7).

There are three ways to compute the Cholesky factorization of matrix A: up-looking, left-looking and right-looking. Left-looking and right-looking methods compute the Cholesky factor column by column starting from the first and the last column respectively. Up-looking algorithm computes the factor row by row starting with the first row and moving down. All there algorithms are described in more detail in Section 3.1.

The Cholesky factorization exists only for SPD systems, and it will fail if the system is not SPD. In fact, attempting to compute this factorization is the cheapest way to check whether a matrix is SPD or not. The reason $U^T U$ factorization fails for non SPD matrices is that it attempts to compute a square root of a negative diagonal entry. This will be clearer in Section 3.2 when the complete algorithm is described step by step. However, here it is important to note that if we compute a factorization of the form $U^T D U$, U will have a unit diagonal and all the diagonal entries will be stored in a diagonal matrix D. Thus, if we allow negative diagonal entries, we can compute a Cholesky-like factorization.

Therefore, Cholesky factorization can be used for symmetric indefinite systems as for SPD systems. In particular, it can be used for SQD matrices discussed in Section 1.3.1.

1.3.4 QR Factorization

The QR factorization of an m-by-n matrix A produces an m-by-m orthogonal matrix Q and an m-by-n upper triangular matrix R. This factorization is unique if there is an additional constraint that all the entries on the diagonal of R are positive.

There are three distinct ways of computing the QR factorization. The Gram-Schmidt method multiplies A by an upper triangular matrix R_i at each step until A is reduced to an orthogonal matrix Q. The upper triangular matrix R

is composed of the inverse of each R_i multiplied together. This algorithm is the simplest way to compute the QR factorization, but even in its modified version it is not as stable as the next algorithm presented [27, Lecture 8].

The second method for computing QR is called Householder or orthogonal triangularization. The basis of this method is computing orthogonal reflectors for each column in A and then applying each reflector in turn to A in order to obtain an upper triangular matrix in the end. This method has been shown to be more stable than the Gram-Schmidt method.

The third way of computing the QR factorization is by means of Givens rotations. In this algorithm the nonzeros in the lower triangular part of A are zeroed out one at a time using one rotation matrix per nonzero entry.

In this work, we choose the Householder reflections as the basis for our QR decomposition. This method allows us to compute a "Q-less" QR factorization since it is not necessary to store the reflectors. In terms of stability this is also a better choice over the Gram-Schmidt method.

1.3.5 Preconditioning

As mentioned earlier, preconditioning has an important role in solving linear systems by iterative methods. The efficiency of the iterative methods often depends on the eigenvalues of the system being solved. In particular, the number of iterations taken by the iterative method is equal to the number of distinct eigenvalues, in the absence of roundoff errors.

Unfortunately, systems encountered in practice often do not have nicely clustered eigenvalues and thus, iterative methods take many iterations when solving these systems. This is where preconditioners come in. Their role is to produce a matrix with as few eigenvalues as possible when multiplied with the original matrix, which can then be solved by an iterative method in fewer iterations. Since the identity matrix I has only one eigenvalue and is solved in one iteration, the prefect preconditioner would be A^{-1}. However, computing the inverse of a matrix is equivalent to solving the linear system of equations directly, so the purpose of computing a preconditioner for an iterative method is defeated. Hence, we need to see other more suitable preconditioners, which are easier to compute than A^{-1}, but are close to it.

As was also mentioned briefly earlier, computing approximate factorizations of the original matrix and using the factors as preconditioners is a common practice. As an extreme example, we can compute the Cholesky-like factorization

$$A = U^T D U$$

of an SQD matrix A. Since A is not positive definite, D will have both positive and negative entries. Our goal is to use this factorization as a preconditioner for SYMMLQ, an iterative method which takes advantage of the symmetry in the system being solved. In order to preserve the sparsity of the preconditioned system we need to apply the Cholesky factor of the preconditioner on both sides of our system. Thus, our preconditioner has to be symmetric positive definite.

In order to use the above factorization as a preconditioner, we have to modify the factors so that they form a positive definite system. Thus, we modify the entries of D such that

$$\bar{D} = |D|,$$

with $|\cdot|$ defined componentwise. Thus,

$$\bar{A} = U^T \bar{D} U$$

is symmetric positive definite, so it can be used as a preconditioner within an iterative method for solving A. When $\bar{A}$ is used as a preconditioner, its inverse is computed and applied to A, so we have

$$\begin{aligned} A\bar{A}^{-1} &= (U^T D U)(U^T \bar{D} U)^{-1} \\ &= U^T D \bar{D}^{-1} U^{-T} \\ &= U^T \bar{I} U^{-T} \\ &= \bar{I}, \end{aligned}$$

because $D\bar{D}^{-1} = \text{diag}(\frac{d_i}{|d_i|}) = \text{diag}(\pm 1) = \bar{I}$, where d_i are entries in the diagonal matrix D. Because $\bar{I}$ is a diagonal matrix with ± 1 on the diagonal, it only has two distinct eigenvalues: 1 and -1. When a Krylov subspace method is used to solve this system, it converges in at most two iterations.

However, in cases of large, sparse systems computing a complete factorization is not only expensive, but the storage requirements are often prohibitively large. During the course of the factorization, fill-in entries are introduced that were not present in the original matrix. Discarding some of these elements as the factors are computed not only saves on storage, but also makes the factorization faster since there are less per entry computations to be done. This is the characterization of the incomplete factorizations.

Different ways in which elements are discarded distinguishes between various algorithms for computing the incomplete factorizations. The elements to be left out can be chosen according to their magnitude or their position in the matrix. If they are chosen according to their magnitude, a certain threshold value is established and all elements smaller than that value in magnitude are discarded. In case of selection by position in the matrix, various strategies exist. We can choose to keep only the entries that were structurally present in the original matrix, in which case the storage requirements are equal to those of the original matrix. If a small amount of additional storage is available, we can choose to store elements of the original matrix plus a small number of fill-in entries per column. Such strategy is called p-incomplete factorization and is the one developed in both Cholesky and the QR factorization presented in this work.

When the p-incomplete factorization is computed, we have that

$$A \approx U^T D U$$

and

$$\bar{A} \approx U^T \bar{D} U,$$

where $\bar{D}$ is also absolute value of D as in the example above. Thus, we hope that AA^{-1} has eigenvalues clustered about $+1$ and -1 and that preconditioned system converges in close to two iterations. The closeness of this approximation and therefore, the clustering of the eigenvalues in $A\bar{A}^{-1}$, depends on the closeness of the incomplete Cholesky factorization to the complete one. This in turn, depends on the amount of fill-in that is left in the incomplete factorization, i.e., in our case the value of p in the p-incomplete factorization.

1.4 Implementation Framework

Implementation details are very important in this work, since efficiency was one of the main criteria. Those details, in turn, depend largely on the way the algorithms are developed and the data structures that are chosen. Therefore, it is important to choose an appropriate base to build our methods on.

The CSparse package [8], written by Tim Davis, is a compact set of basic tools for sparse matrices. It includes most common sparse matrix tools starting from the basic ones like transpose, addition and multiplication up to the more complex ones like LU, Cholesky, and QR factorizations. The entire package is written in the C programming language and totals only about 2200 lines of code. It also includes a `mex` interface which enables the user to call all the function easily from MATLAB.

CSparse was a good starting point for implementation of our incomplete factorization algorithms because it already includes a sparse Cholesky and a sparse QR factorization. The transition from complete to incomplete factorizations was not trivial but a lot of the underlying tools, like elimination trees and sparse triangular solves which will be discussed in more detail later, translated smoothly from complete to incomplete.

1.5 Other Work

Various algorithms have been implemented to compute the incomplete Cholesky and incomplete QR factorizations, which differ in the approach used to compute the factorization and the dropping strategy.

The left-looking Cholesky factorization is the most common algorithm; the columns of the factors are computed sequentially from left to right. The distinguishing factor between these is the dropping strategy used to discard entries.

Meijerink and van der Vorst [18] proposed the first incomplete Cholesky factorization that was based on retaining the sparsity of the original matrix. This strategy allowed only the entries which were structurally present in the original matrix to be in the factor, and is therefore called no fill-in strategy by Jones and Plassmann [16]. A significant benefit of this approach is that memory requirements are completely predictable, but because the dropping does not take into account values of the entries, too much information is sometimes lost.

On the other hand, entries can be dropped based on their magnitude only,

by establishing a threshold value and discarding all entries smaller than that value. This approach was implemented by Munksgaard [19]. The advantage of this strategy over the one that preserves the sparsity is that the largest, and therefore the most significant, elements are preserved in the factorization. The disadvantage, which can play an important role, is that the storage requirements are unpredictable and have to be determined "on the fly" during the factorization. Also, the optimal threshold parameter is chosen by trial and error for each system being solved, which involves too much user input.

An approach that combines advantages of both strategies previously described was implemented by Jones and Plassmann [16]. Their algorithm maintains a fixed number of nonzero entries in each column. However, the sparsity pattern (i.e., the location of nonzero entries) is allowed to change in the factor. This strategy is "black-box" in the sense that it does not require any input parameters from the user, and its memory requirements are predictable.

The algorithm developed by Lin and Moré [17] follows the approach of Jones and Plassmann but allows for some additional entries in each column. In their case, the elements are discarded according to their sparsity pattern only and their value is not taken into account, as opposed to the Jones and Plassmann's approach where only the value is considered. They, however, do not take into account the benefits of fill-reducing reorderings, which prove to be very powerful in our implementation.

In terms of incomplete QR factorizations, much less has been done in the past. This is mostly due to the fact that QR factorizations are more difficult to implement in general, and especially so for sparse systems. Notable works include CIMGS by Wang et al. [29], and a class of incomplete orthogonal factorizations by Bai et al. [3], with the follow-up work by Papadopoulos et al. [23]. The CIMGS algorithm is based on Gram-Schmidt method for computing the QR factorization, and it uses some predetermined dropping set as the strategy for dropping. As mentioned earlier, Gram-Schmidt algorithm is not as stable as other methods of computing QR factorization. Also, CIMGS assumes that the dropping set is available before the factorization starts, which would require the user to provide one.

The implementation discussed in [23] uses Givens rotations algorithm for computing QR factorization, which is more stable than the Gram-Schmidt approach. However, the sparsity pattern of the factors still has to be predetermined. The choices discussed are the sparsity pattern of the original matrix A and the sparsity pattern of A^TA. This can be a disadvantage if A^TA is completely dense, as is the case when A has a dense row.

In our work, we make an effort to combine some of the best qualities of these previously developed algorithms, such as predictable storage and stability of the factorization.

1.6 Software Packages

There are a few software packages available that deal with sparse matrix factorizations. Ones of interest to us are the sequential packages that contain sparse LDL^T factorization with 1-by-1 pivoting and those that have a sparse QR factorization. Notable ones that include LDL^T are CHOLMOD [7], LDL [6], TAUCS [24] and WSMP [14]. For the QR factorization we single out BCSLIB-EXT [2] and SPARSPAK [10]. There are many more software packages that are either parallel or only deal with 2-by-2 pivoting in the factorization, and are therefore not very comparable to implementation in the CSparse package which is used as the basis for all the implementation in this work.

1.7 Notation

We closely follow the notation used in [8]. All matrices are denoted by capital letters and in case of block matrices blocks are denoted by capital letter with a subscript indicating its position in the matrix, for example A_{11} for the (1,1) block submatrix.

Vectors are denoted by lower case roman letters and in the cases where they are part of a matrix their position in the matrix is indicated by the subscript the same way as for matrices. For example, u_{12} is a column vector occupying the $(2,1)$ block of a block matrix.

Scalars are denoted by lower case Greek letters. For block matrices the subscript notation is the same as for matrices and vectors.

Chapter 2

Sparse Matrices

2.1 Data Structures

The choice of data structure for storing a matrix is very important when dealing with algorithms for sparse matrices. Because the data structure in which the sparse matrix is stored influences how the matrix is accessed most efficiently, it also has a large impact on all algorithms involving this matrix.

2.1.1 Triplet

The triplet form has three vectors, `i`, `j`, and `x`, each of size nnz(A) (number of nonzeros in the matrix A), representing row indices, column indices, and values of each entry, respectively. The following example illustrates the triplet data structure. Given the matrix

$$A = \begin{bmatrix} 4.5 & 0 & 1.7 \\ 3.2 & 2.8 & 0 \\ 5.1 & 0 & 6.9 \end{bmatrix},$$

its triple form representation is

```
   int i[]  = { 0,   2,   1,   2,   0,   1   };
   int j[]  = { 2,   2,   1,   0,   0,   0   };
double x[]  = { 1.7, 6.9, 2.8, 5.1, 4.5, 3.2 };
```

(Note that the entries are not necessarily sorted in any order.)

The triplet data structure is very easy to implement and use, but it does not provide any special accessability to the sparse matrix. The next two data structures are organized in a way which makes the access to the sparse matrix easier for algorithms using this matrix.

2.1.2 Column Compressed Storage

The column compressed format (CCS) is also made up of three vectors, `p`, `i`, and `x` but their roles are slightly different than those of vectors in triplet format. In this case, `p` is of length n, where n is the number of columns in A, and it contains the column pointers of each column of A. The last entry in `p` is the total number of nonzeros in the matrix. In other words, k^{th} entry in `p` represents the vector index of `i` at which the k^{th} column starts. The vector `i` has row indices for each entry, and `x` has value of each entry.

The above example matrix in the column compressed form would be

```
 int i[]    = { 0,                  3,    4,          6 };
 int j[]    = { 0,    1,    2,    1,    0,    2       };
double x[]  = { 4.5,  3.2,  5.1,  2.8,  1.7,  6.9     };
```

A significant advantage of this format over the triplet format is that CCS makes it easy for algorithms to access matrices by columns. For example, we know that the k^{th} column starts at index `p[k]` of `i`, and ends at index `p[k+1]-1` of `i` (because the $(k+1)^{\text{th}}$ column starts at `p[k+1]`). Therefore, the row indices of all the entries in k^{th} column are stored in consecutive positions, between index `p[k]` and `p[k+1]-1` in `i`, and all the values are stored between index `p[k]` and index `p[k+1]-1` in `x`.

All input and output matrices of CSparse are stored in CCS format.

2.1.3 Row Compressed Storage

Row compressed storage (RCS) is very similar to CCS except that instead of columns the matrices are easily accessed by rows. Thus, the vector `p` contains row pointers, and the vector `i` contains column indices. As with CCS, vector `x` contains the values of entries. In the RCS format, our example matrix is represented by

```
 int i[]    = { 0,          2,          4,          6 };
 int j[]    = { 0,    2,    0,    1,    0,    2       };
double x[]  = { 4.5,  1.7,  3.2,  2.8,  5.1,  6.9     };
```

In the same way that CCS format allows for convenient column access, the RCS format allows for convenient row access. The k^{th} row starts at index `p[k]` of `i`, and ends at index `p[k+1]-1` of `i`. Thus, the column indices and values of all entries in k^{th} row are stored between indices `p[k]` and `p[k+1]-1` of `i` and `x`, respectively.

As we will see in Section 3.3, there is an interesting connection between CCS and RCS which will be important in the implementation of our p-incomplete Cholesky algorithm.

2.2 Sorting with Heaps

Binary heaps are a subgroup of binary trees that satisfy some additional properties. There are two different kinds of heaps: minimum and maximum. In this section, we discuss maximum heaps, because these are the ones that will be used in our algorithms; the same definitions and rules apply for minimum heaps.

Binary heaps (henceforth referred to as simply *heaps*) are characterized by a rule called the *heap property:* each node in the heap is greater than or equal to each of its children, according to some comparison rule which is specified for the whole data structure. This, in particular, means that the "root" (i.e., the top node) of the heap is the largest node.

Heaps can be stored in a linear array of length n, where n is the number of nodes. Building a heap has a cost of $\mathcal{O}(n)$ [5]. The heap property ensures that the first element in this array is the largest element in the heap by rearranging elements in the array. Thus, extracting the largest element has cost of $\mathcal{O}(1)$. Extracting any other element (that is not the largest), or adding an element, involves the process of restoring the heap property. This process has $\mathcal{O}(\log n)$ cost.

Sorting n elements using a heap amounts to n deletions from the heap, where the largest element is deleted each time and the heap property is restored. Therefore, the heapsort has the total cost of $\mathcal{O}(n \log n)$.

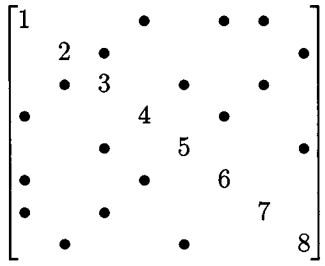
Typically, only the p largest elements (with $p \ll n$) are required in the context of the p-incomplete factorization. Each time, the largest element is extracted and the heap property is restored. This process is repeated p times, and thus, the total cost is $\mathcal{O}(p \log n)$, and because $p \ll n$ this is $\mathcal{O}(\log n)$. Therefore, the total cost of building the heap and extracting p largest elements is $\mathcal{O}(n)$.

2.3 Graphs and Elimination Trees

Elimination trees are, in essence, the result of pruning graphs that represent matrices. Thus, in order to understand the elimination trees, which play an important role in the sparse Cholesky factorization, an introduction to some basic concepts in graph theory is necessary.

First we establish some basic notation for graphs. A graph $G = (V, E)$ is defined by a set of vertices $V = \{1, \ldots, n\}$ and a set of edges $E = \{(i, j) \mid i, j \in V\}$. If the graph is undirected, then $(i, j) \in E$ means that there is an edge from node i to node j, and an edge from node j to node i. In a directed graph, $(i, j) \in E$ means that there is an edge from node i to node j only. A *path* between nodes v_0 and v_k, denoted $v_0 \rightsquigarrow v_k$, represents a sequence of nodes $(v_0, \ldots, v_k)$ such that $(i - 1, i) \in E$ for each $i = 1, \ldots, k$. If a path $i \rightsquigarrow j$ exists, the node j is *reachable* from node i. The set of all nodes reachable from node i in the graph G is denoted by $Reach_G(i)$.

A matrix A is represented by a graph in the following way. Each column of the matrix is represented by a node in V, and each edge in E corresponds to a nonzero entry in A. Thus, if $a_{ij} \neq 0$ then $(i, j) \in E$. Symmetric matrices correspond to undirected graphs because in this case, if $a_{ij} \neq 0$ then $a_{ji} \neq 0$ as well. Consider the matrix



where • represents a nonzero entry. Its corresponding graph is shown in Figure 2.1. Because A is symmetric, its graph is undirected.

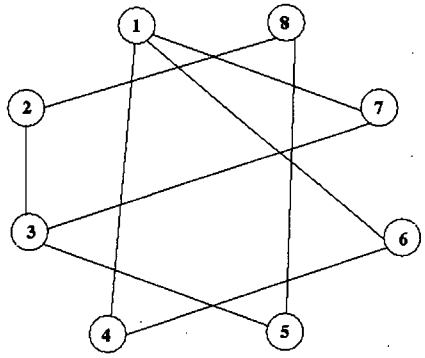


Figure 2.1: Graph representation of A

The concept of reachability is used to compute the sparsity pattern of a row in the lower triangular Cholesky factor. Each row of the factor is computed by a sparse triangular solve

$$Lx = b, \tag{2.1}$$

where x is a row of the factor, L is a submatrix of the computed portion of the factor, and b is a column of the original matrix A. This procedure will be discussed in detail in Section 2.4 and in chapter on the Cholesky factorization. Right now, we focus on how the sparsity pattern of x is computed.

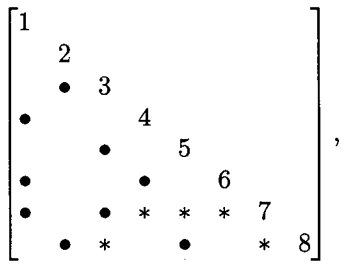
Each nonzero entry b_i in b has a corresponding node i in the graph G representing the submatrix L, and each one of these nodes has its reachable set $Reach_G(i)$. Theorem 3.1 of [8] states that solution of the sparse linear system $Lx = b$ has the sparsity pattern equal to the reachable set of all the nodes i, such that $b_i \neq 0$. In other words, the sparsity pattern of x is equal to $Reach_G(\mathcal{B})$, where $\mathcal{B} = \{i \mid b_i \neq 0\}$.

In order to make the computation of the reachable sets more efficient, graphs are pruned such that the reachability of each node is not changed. This is accomplished by removing the cycles from the graph. End result of the pruning is the elimination tree. Thus, the elimination tree is used to compute the sparsity pattern of each row of the lower triangular Cholesky factor L.

Two relatively simple rules describe the procedure by which the nonzero pattern of the Cholesky factor L can be determined. If we let a_{ij} denote the entries of the original matrix A, and l_{ij} denote the entries of the Cholesky factor L, $a_{ij} \neq 0$ implies $l_{ij} \neq 0$ [8, Theorem 4.2]. Also, if there exist i, j, k such that $i < j < k$, and $l_{ji} \neq 0$ and $l_{ki} \neq 0$, then $l_{kj} \neq 0$ [8, Theorem 4.3].

In terms of the graph representation, if $l_{ji} \neq 0$ and $l_{jk} \neq 0$, the two corresponding edges are in the graph, i.e., $(i, j) \in E$ and $(j, k) \in E$. Thus, there is a path from i to k which does not traverse the edge (i, k). Therefore, this edge

becomes obsolete and can be pruned from the graph. By removing all cycles in the original graph, the elimination tree, shown in Figure 2.2, is obtained. The corresponding Cholesky factor has the following nonzero pattern



where $*$ represents a fill-in entry.

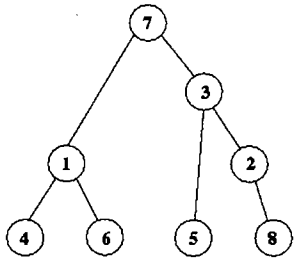


Figure 2.2: Elimination tree

2.4 Sparse Triangular Solves

Each iteration of a Cholesky algorithm involves the solution of a triangular system. In this section we describe how a sparse triangular solve can be efficiently computed in two different ways. The distinction between these two approaches proves to be crucial for the implementation of the p-incomplete Cholesky algorithm.

Because the lower triangular Cholesky factor L is stored in compressed column form, it makes sense to access it by columns. We explicitly express (2.1) as

$$\begin{bmatrix} \ell_{11} & \\ l_{21} & L_{22} \end{bmatrix} \begin{bmatrix} \varepsilon_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \beta_1 \\ b_2 \end{bmatrix},$$

where L_{22} is $(n-1)$-by-$(n-1)$ submatrix of L; l_{21}, x_2 and b_2 are column vectors of length $n-1$ and ℓ_{11}, ε_1 and β_1 are scalars.

The solution can be computed by forming the two equations

$$\begin{aligned} \ell_{11}\varepsilon_1 &= \beta_1 \\ l_{21}\varepsilon_1 + L_{22}x_2 &= b_2, \end{aligned} \tag{2.2}$$

and solving the first equation with $\varepsilon_1 = \beta_1/\ell_{11}$ and then recursively solving the second equation.

An analogous approach can be used to solve $U^T x = b$ where U is upper triangular. This is the same as solving $x^T U = b^T$ for x, and hence the system can be written as

$$\begin{bmatrix}\varepsilon_1 & x_2^T\end{bmatrix}\begin{bmatrix}\omega_{11} & u_{12} \\ & U_{22}\end{bmatrix} = \begin{bmatrix}\beta_1 & b_2^T\end{bmatrix},$$

which leads to the set of equations

$$\begin{aligned}\varepsilon_1\omega_{11} &= \beta_1 \\ \varepsilon_1 u_{12} + U_{11}^T x_2 &= b_2.\end{aligned} \tag{2.3}$$

Because the solution obtained in both of these approaches is the same, they can be used interchangeably. The main difference is that the input for the first algorithm is a lower triangular matrix L and for the second algorithm it is an upper triangular matrix U. This distinction will be of importance in the implementation of the p-incomplete Cholesky factorization.

2.5 Ordering for Sparsity

Factorizations of sparse matrices often introduce a large amount of fill-in in their factors. The fill-in entries are those that were structurally zero in the original matrix, but are nonzero in one of the factors. Large amounts of fill-in are undesirable in any factorization since they slow down the algorithms and increase the amount of storage necessary for the factors. It is well known that certain permutations introduce smaller amount of fill-in than others. Unfortunately, finding the optimal permutation is an NP-complete problem [31], and therefore impossible in practice. The upside is that there are many relatively inexpensive algorithms which do very well in approximating the optimal ordering. The most common one of these algorithms is approximate minimum degree ordering, but others, such as nested dissection ordering and bandwidth reduction algorithms, are used as well.

2.5.1 Minimum Degree

The minimum degree (MD) reordering algorithm considers a matrix in terms of its graph representation, where each node is a column of the matrix, and each edge represents a nonzero entry. The degree of each node is the number of edges that are incident on it; equivalently, the degree of each node is the number of nonzeros in column represented by that node.

At each step of the MD algorithm a node is chosen and removed from the graph along with all of its adjacent edges [9]. The node chosen is the one with the minimum degree, and thus, the algorithm is considered greedy. In theory, this algorithm works only on symmetric matrices which have corresponding graphs

that are undirected. In practice, if A is not symmetric, the method is used on A^TA as is done in CSparse as a pre-processing step for QR factorization.

In practice, this method has storage requirements that dramatically exceed that of the original graph representing matrix and is therefore not very practical. Instead, the approximate minimum degree (AMD) algorithm is used. This method works on the quotient graphs instead of the original graphs representing the matrix [8, Chapter 7]. Since quotient graphs require at most as much storage as the original graph, AMD is more favorable when working on large systems.

An advantage of the MD algorithm, and therefore AMD as well, over other methods of reordering is that it does not consider the values of the entries in the matrix. This means that the reordering can be completed before the factorization is started, which simplifies the factorization itself greatly. Furthermore, this means that for sets of matrices with the same nonzero pattern, which are often encountered in practice, reordering needs to be done only once per set.

2.5.2 Other Methods for Reordering

Nested dissection ordering is another method for reducing the fill-in in factorization of sparse matrices. This method is less commonly used in practice since it is more difficult to get the same results as the minimum degree algorithm [15].

The algorithm works on the basis of choosing and removing a set of separator nodes S which divide the graph into two disconnected subgraphs of roughly the same size [9]. These subgraphs are then divided by the same method and the procedure can be repeated many times. Grouping the nodes in each subgraph together and numbering the separator nodes of each separation last reduces fill-in during the factorization.

Bandwidth reduction algorithms reduce the amount of fill-in in the factors by reducing the bandwidth of the factored matrix. The term bandwidth refers to the number of nonzeros between the first and last nonzero in each row. The most popular of these algorithms is the reverse Cuthill-McKee [25, Chapter 3] ordering which is based on traversing level sets of nodes according to their degree.

Chapter 3

Incomplete Cholesky Factorization

Every positive definite matrix A can be factorized as

$$A = U^T U. \tag{3.1}$$

If we require U to be triangular, as we do here, then (3.1) can be considered as a special case of LU factorization. There are three ways to compute a Cholesky factorization: up-looking, left-looking and right-looking, and the distinction between the three approaches is made based on the order in which the factor is computed. The next section describes all three methods and explains why one was chosen over the others.

3.1 Which Way to Look: Left, Right and Up

The *left-looking* Cholesky algorithm is used most commonly. This method produces a lower triangular factor L, which is computed one column at the time, starting from the first column. It is based on the following decomposition of the original matrix and the factors

$$\begin{bmatrix} L_{11} & & \\ l_{12}^T & l_{22} & \\ L_{31} & l_{32} & L_{33} \end{bmatrix} \begin{bmatrix} L_{11}^T & l_{12} & L_{31}^T \\ & l_{22} & l_{32}^T \\ & & L_{33}^T \end{bmatrix} = \begin{bmatrix} A_{11} & a_{12} & A_{31}^T \\ a_{12}^T & a_{22} & a_{32}^T \\ A_{31} & a_{32} & A_{33} \end{bmatrix}.$$

In the context of sparse matrix computation, this algorithm requires the nonzero pattern of all columns of L to be computed before the numerical factorization starts. It also requires the access to the numerical values of the entries of k^{th} row when the k^{th} column is computed. These requirements make the algorithm more complicated and more expensive than other algorithms presented in this section.

The *right-looking* algorithm is similar to the left-looking algorithm, except that the columns are computed starting at the last column and moving left. It is based on the following decomposition

$$\begin{bmatrix} a_{11} & a_{21}^T \\ a_{12} & A_{22} \end{bmatrix} = \begin{bmatrix} \ell_{11} & \\ l_{21}^T & L_{22} \end{bmatrix} \begin{bmatrix} \ell_{11} & l_{21}^T \\ & L_{22} \end{bmatrix}.$$

Thus, when k^{th} column, l_{21} above, is being computed, columns $k+1$ through n (submatrix L_{22}), are assumed to be completed, and are used in the computation

of k^{th} column. The three equations following from above decomposition form the basis for this algorithm

$$\begin{aligned} a_{11} &= \ell_{11}^2 \\ a_{12} &= \ell_{11} l_{12} \\ A_{22} &= l_{12} l_{12}^T + L_{22} L_{22}^T. \end{aligned}$$

The algorithm used in implementation of the complete Cholesky factorization is known as the *up-looking* Cholesky. This means that the lower triangular Cholesky factor L is computed one row at the time starting from the first row, and moving down. The following block decomposition demonstrates how one row of L is computed

$$\begin{array}{c} \\ n-1 \\ 1 \end{array} \begin{array}{c} \begin{array}{cc} n-1 & 1 \end{array} \\ \begin{bmatrix} A_{11} & a_{12} \\ a_{12}^T & a_{22} \end{bmatrix} \end{array} = \begin{array}{c} \begin{array}{cc} n-1 & 1 \end{array} \\ \begin{bmatrix} L_{11} & \\ l_{12}^T & \ell_{22} \end{bmatrix} \end{array} \begin{array}{c} \begin{array}{cc} n-1 & 1 \end{array} \\ \begin{bmatrix} L_{11}^T & l_{12} \\ & \ell_{22} \end{bmatrix} \end{array}.$$

Thus, when the k^{th} row (represented by l_{12}^T) is being computed, it is assumed that rows 1 through $k-1$ (represented by L_{11}) have already been computed, and that they can be used in the computation. From the above decomposition three equations can be formed that essentially constitute the up-looking algorithm

$$A_{11} = L_{11} L_{11}^T \tag{3.2}$$

$$a_{12} = L_{11} l_{12} \tag{3.3}$$

$$a_{22} = l_{12}^T l_{12} + l_{22}^2. \tag{3.4}$$

Most of the work in this algorithm lies in the sparse triangular solve (3.3). Because this triangular solve is computed with a lower triangular matrix L_{11}, the algorithm used to compute it is the one that solves $Lx = b$ for x, i.e., algorithm described by (2.2) in Section 2.4.

In the p-incomplete Cholesky factorization, a variation of the up-looking Cholesky is implemented. In this case, an upper triangular factor U is computed, and the algorithm is based on decomposition

$$\begin{array}{c} \\ n-1 \\ 1 \end{array} \begin{array}{c} \begin{array}{cc} n-1 & 1 \end{array} \\ \begin{bmatrix} A_{11} & a_{12} \\ a_{12}^T & a_{22} \end{bmatrix} \end{array} = \begin{array}{c} \begin{array}{cc} n-1 & 1 \end{array} \\ \begin{bmatrix} U_{11}^T & \\ u_{12}^T & \omega_{22} \end{bmatrix} \end{array} \begin{array}{c} \begin{array}{cc} n-1 & 1 \end{array} \\ \begin{bmatrix} U_{11} & u_{12} \\ & \omega_{22} \end{bmatrix} \end{array}.$$

Thus, there are three equations that construct this version of the up-looking Cholesky algorithm

$$A_{11} = U_{11}^T U_{11} \tag{3.5}$$

$$a_{12} = U_{11}^T u_{12} \tag{3.6}$$

$$a_{22} = u_{12}^T u_{12} + \omega_{22}^2. \tag{3.7}$$

An important observation here is that the triangular solve, equation (3.6), is computed with an upper triangular matrix U_{11}, instead of the lower triangular matrix as in (3.3). This change was crucial for implementation of the

Algorithm 1: Numeric phase of a complete Cholesky factorization.

Input: Sparse matrix A
Output: Lower triangular matrix L
for k=1,...,n **do**
  Compute nonzero pattern of $L(k,:)$;
  $x = A(1:k,k)$;
  $d = A(k,k)$;
  foreach *Nonzero entry* $L(k,i)$ **do**
    $L(k,i) = x(i)/L(i,i)$;
    $x = x - L(:,i) * L(k,i)$;
    $d = d - L(k,i) * L(k,i)$;
    Store $L(k,i)$;
  $L(k,k) = \sqrt{d}$;
  Store $L(k,k)$;

p-incomplete Cholesky algorithm, and the reasons for this will become more clear in Section 3.3 when we discuss storage issues encountered while implementing the incomplete factorization.

3.2 Complete Cholesky

The method for implementing the complete Cholesky factorization is based on solving the system of equations (3.2)-(3.4). Some of the main concepts from the implementation of this algorithm apply directly for the incomplete factorization and thus, the complete algorithm is described in more detail below.

The Cholesky factorization is divided into symbolic and numerical phases. Characteristics of the factorization which only depend on the sparsity pattern of the input matrix are computed in the symbolic phase. One of the components that depends on the sparsity pattern of the original matrix only is the elimination tree, which is used to compute the sparsity pattern of each row of the factor L. Fill-reducing reordering of the matrix is also computed in the symbolic portion of the method, using the approximate minimum degree algorithm. This approach of computing the reordering separately from the numerical factorization provides a benefit for sets of matrices with the same pattern and different values of entries. In this case, the reordering has to be performed only once for the whole set.

The numerical part of the factorization is summarized in Algorithm 1. This process is described in detail because this is where the bulk of the work of the entire method lies. For each row of the factor, the numerical factorization can be divided into three main parts. The first part is computing the sparsity pattern of a row. Sparsity pattern is computed from the elimination tree using the idea of reachability, discussed in Section 2.3. In this portion of the method, contents of the upper triangular part of the column of A corresponding to the

row currently being computed are copied into a dense vector x. From here onward, all the computation is done using this dense vector, in order to allow sufficient space for possible fill-in elements. Because the input matrix A is assumed to be symmetric, only its upper triangular part is accessed.

Once the nonzero pattern of a row of L has been computed, a sparse triangular solve is used to compute the values of the entries in this row. Referring to the set of equations (3.2)-(3.4), which are the basis of this algorithm, this triangular solve corresponds to (3.3). The row being computed is the transpose of the solution to this equation, i.e., l_{12}^T. As the name up-looking Cholesky suggests, computation of this row requires looking up at the rows that have already been computed, in other words the submatrix L_{22} from (3.3).

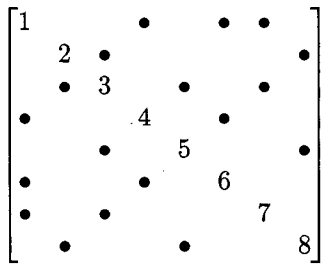
The final step is the computation of the diagonal entry, which is omitted from the triangular solve. Computing the scalar d corresponds to computing ℓ_{22}^2 in (3.4). The square root of d is then the diagonal entry ℓ_{22}.

3.3 Column vs. Row Storage

The main issue in implementing the p-incomplete algorithm came from the fact that the complete Cholesky factorization in CSparse is computed in a row fashion. The goal was to keep p fill-in entries per row of the matrix, and since these entries were chosen according to their magnitudes, it was impossible to predict column indices of these entries.

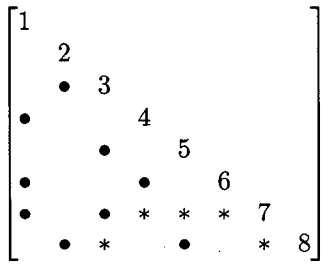
Because the matrix is stored CCS format, entries in one column are stored in consecutive positions, and the columns are stored consecutively as well. In other words, all entries in the first column are stored at the beginning of the array, followed by all entries in the second column and so on. When computing the p-incomplete factorization, we chose p fill-in elements in a row according to their magnitude, and therefore we do not know which column they belong to ahead of time. Thus, we do not know where they belong in the array, and cannot leave adequate space for them.

To illustrate this issue, we go back to the example considered previously in Section 2.3:



It turns out that its complete Cholesky factor will have the following nonzero

pattern



We would like to compute p-incomplete factorization of the above matrix, with $p = 2$. When computing row seven of this factor, which has three fill-in elements marked by $*$, suppose that we decide to keep elements $(7, 4)$ and $(7, 6)$ as these are the two largest in magnitude. Because the matrix is accessed by columns, and computed by rows, column pointers need to be computed before the factorization so that the algorithm can access the matrix by columns. In order to do this, we have to know how many entries there will be in each column at the end of the factorization. In the complete factorization, this is easy, since all fill-in entries are kept. In our case, however, we would have to know before the factorization that the entries $(7, 4)$ and $(7, 6)$ will be kept in the end, and leave space for them in columns 4 and 6 respectively. But the decision to keep these entries was made after the row was already computed, and thus, the conflict is clear.

The solution to this problem is to compute the factor column-by-column instead. This way we keep p largest fill-in entries per column, and each time an entry (fill-in or not) is computed it is stored in the next position in the array.

The easiest way to implement this change while still preserving the structure of the original algorithm, is to change the way the matrices are stored. If a matrix A is stored in column compressed form, but viewed in terms of row compressed form, the transpose of the matrix, A^T, is obtained. This switch simply means that the row and column indices are interchanged, which is the definition of the transpose.

Therefore, if our lower triangular matrix L is stored by rows, but we view it by columns, we obtain upper triangular matrix $U = L^T$. Now, the transpose of that U^T is a lower triangular matrix which is stored by columns. Thus, when we compute the factorization, we compute columns instead of the rows.

This change in storage has the most effect on the sparse triangular solve part of the algorithm. In the complete Cholesky algorithm, a row was computed by the sparse triangular solve of the form $L_{11}l_{12} = a_{12}$, where the transpose of l_{12} was the new row. After the change in storage, in the p-incomplete factorization, we compute the column using the variation of the sparse triangular solve $U_{11}^T u_{12} = a_{12}$, where u_{12} is the new column. From Section 2.4 and the above discussion, we can deduce that the two vectors, l_{12}^T and u_{12}, are simply transposes of each other, and thus, the factor U (if the complete factorization is computed) is just a transpose of the factor L in the complete Cholesky factorization.

Algorithm 2: Numeric phase of our p-Incomplete Cholesky factorization.

Input: Sparse matrix A, p, diagonal tolerance
Output: Upper triangular matrix U, diagonal matrix D
for k=1,...,n **do**
  Compute nonzero pattern of $U(:,k)$;
  $x = A(1:k,k)$;
  ix = sparsity pattern of triu($A(:,k)$);
  $d = A(k,k)$;
  foreach *Nonzero entry* $U(i,k)$ **do**
    $x = x - U(:,i) * x$;
  foreach *Nonzero entry* $U(i,k)$ **do**
    $x(i) = x(i)/D(i)$;
    $d = d - U(k,i) * U(k,i)$;
    Mark all fill-in entries;
  Store all fill-in entries in heap;
  Extract p largest fill-in from heap;
  Store fill-in and non fill-in in U;
  If d is small, modify it;
  Set $D(k) = d$;

3.4 p-Incomplete Cholesky

The p-incomplete factorization takes the best qualities from other incomplete factorizations described previously in Section 1.5. Since only p fill-in elements per column are allowed in addition to the elements in the original matrix, the additional storage requirements are not only predictable, but are also determined by the user. On the other hand, the largest, and therefore, the most significant, fill-in entries are preserved, which ensures that the incomplete factor is as close as possible to the complete factor, for a particular p.

It is important to note that the p-incomplete Cholesky factorization implemented here is of the form U^TDU, where U is an unit upper triangular, and D is a diagonal matrix. Unlike the complete Cholesky factorization described in the previous section which only works for SPD systems, this p-incomplete factorization can be used for indefinite systems as well. In particular, we are interested in factoring SQD systems, in which case the diagonal factor will have both positive and negative entries.

Similarly to the complete factorization, the p-incomplete factorization is divided into symbolic and numeric parts. Symbolic factorization computes the elimination tree and the fill-reducing ordering, which depend on the sparsity pattern of the original matrix only, and not on the values of the entries. The numerical p-incomplete factorization is summarized in Algorithm 2 and is described in more detail below.

The numerical part of the factorization begins similarly to that of the complete factorization algorithm, by computing the sparsity pattern of the k^{th} col-

umn of the factor. It is important to note that initially we compute the sparsity pattern of the complete column, and later decide which entries to keep according to their magnitude. Once the pattern is computed, we go through the k^{th} column of the original matrix, and scatter the values and row indices into dense vectors x and ix respectively. These will be used to store all the values and their row indices (including those for fill-in entries) during the triangular solve, and therefore they need to be dense.

The next phase is to compute the values of all entries in the k^{th} column by means of a sparse triangular solve. As described in the previous section, triangular solve used here is one that solves $U^T x = b$, unlike the complete Cholesky factorization where $Lx = b$ is solved for x. Both algorithms are described in more detail in Section 2.4.

Once all the entries have been computed, we must choose which p fill-in entries will be kept. In case where the total number of fill-in entries in a column is less than p, all of the entries are kept. Otherwise, all fill-in entries of a column are stored in a maximum heap and p largest elements are extracted from the heap one by one. As discussed in Section 2.2, this operation is very efficient because p is typically much smaller than the total number of fill-in entries. The fill-in entries chosen are then marked to be stored in the factor U.

After choosing and marking p elements to keep, those and all entries structurally present in A are stored in the factor U. Finally, the diagonal entry is computed and stored in the diagonal matrix D. Because of the successive dropping in each column some of the diagonal entries become very small. If the magnitude of the diagonal entry is smaller than the specified tolerance, the entry is modified. In the current implementation this tolerance is a user input. In future implementations, we would prefer to base it on considerations of keeping the norm of the factors bounded. In our numerical experiments (Section 5.1.3), this modification is important for some systems, but for majority of the problems tested it does not create significant improvement in terms of preconditioning.

As discussed previously in Section 1.3.3, the p-incomplete Cholesky factorization is applied to SQD systems, in which case the diagonal entries are both positive and negative. In this case, $U^T U$ Cholesky factorization would not work, as it would break down as soon as it encounters a negative diagonal entry. In $U^T D U$ Cholesky this is not an issue, since the diagonal entries are stored in a separate diagonal matrix D. In this case, factor U is unit upper triangular, and thus, unique.

Chapter 4

Incomplete QR Factorization

Given an m-by-n matrix A, its QR factorization is given by

$$A = QR, \tag{4.1}$$

where Q is orthogonal, and R is upper triangular. In this chapter, we consider methods for computing the "thin" QR decomposition; in this case, Q is m-by-n with orthogonal columns, and R is n-by-n upper triangular.

The QR factorization is a tool most commonly used to solve the generic least-squares problems which minimize the 2-norm of the residual and are formulated as

$$\underset{x}{\text{minimize}} \quad \|Ax - b\|. \tag{4.2}$$

The factor Q of A has orthogonal columns, and any vector can be multiplied by it without changing its 2-norm. Therefore, we can compute the QR factorization of A and use the factor Q to reformulate the least squares problem (4.2) as

$$\underset{x}{\text{minimize}} \quad \|Q^TAx - Q^Tb\|.$$

Because Q has the property that $Q^TQ = I$, we have that $Q^TA = R$, and thus, the above least squares problem is simplified into a problem with an upper triangular system, which is much easier to solve.

In this work, however, QR factorization is used to precondition least squares problems and use an iterative solver, LSQR, to solve the preconditioned system. In this case, only the factor R is needed to solve (4.2) as

$$\underset{x}{\text{minimize}} \quad \|AR^{-1}y - c\| \quad \text{and} \quad Rx = y.$$

In this case the factor Q does not need to be stored.

As described in Section 1.3.4, QR decomposition can be computed in three ways. The algorithm presented here is based on the Householder reflection method, otherwise known as orthogonal triangularization.

Orthogonal triangularization works on the basis of successively computing orthogonal reflectors corresponding to each column of the original matrix A, and then applying each reflector to A. Figure 4.1 shows the results of applying

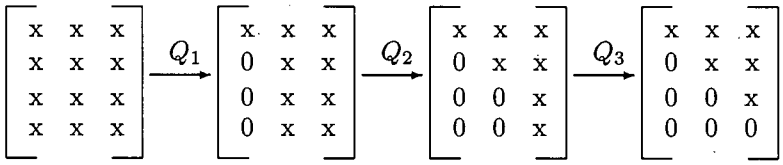


Figure 4.1: Orthogonal triangularization

orthogonal matrices, Q_1, Q_2, and Q_3 to the original matrix. Each Q_j is of the form

$$Q_j = \begin{bmatrix} I_j & 0 \\ 0 & F_j \end{bmatrix},$$

where F_j is a $(n-j+1)$-by-$(n-j+1)$ Householder reflector that zeros out the lower triangular part of column j. Because of the identity matrix in the $(1,1)$ block, Q_j only affects columns j through n; Q_j does not touch columns 1 through $j-1$ which already had their appropriate entries zeroed out by Q_1 through Q_{j-1}.

Each reflector F has a special structure which zeros out all the entries below the main diagonal and puts their weight in the diagonal entry. For example, if we denote by x_j the lower triangular part of column j, including the diagonal entry, then after applying F_j, the column will contain $\|x_j\|$ on the diagonal and zeros below the diagonal. The upper triangular part of the column is left unchanged.

The structure of F that makes this possible is

$$F = I - \beta v v^T \quad \text{with} \quad \beta = 2/\|v\|, \tag{4.3}$$

where β and v are chosen such that for any x,

$$Fx = \pm\|x\| e_1, \tag{4.4}$$

with $e_1 = (1, 0, \ldots, 0)^T$. The decision on whether to use $+$ or $-$ is made based on the sign of x_1, the first entry in column x. More details on this whole procedure are available in [27, Lecture 10].

The order in which the reflectors are computed and applied distinguishes between two different algorithms described in the next section.

4.1 Which Way to Look: Left and Right

Algebraically, the order in which the Householder reflectors are computed and applied does not have any effect on the end result of the factorization when computing a complete QR factorization. However, the distinction turns out to be crucial when implementing the incomplete QR factorization. There are two ways algorithms for computing the sparse Householder QR factorization: left-looking and right-looking.

Both algorithms compute the upper triangular matrix R corresponding to factor R in (4.1), and lower triangular matrix V which holds all the reflection vectors v (see (4.3)). The orthogonal matrix Q can be built from the matrix V, but will not be stored in practice.

The k^{th} column R_k of R and k^{th} column V_k of V are computed simultaneously. We begin with the original matrix A and at each step compute a reflector from the lower-triangular part of k^{th} column A_k of A, as described in the example above. We have a choice between applying this reflector to all columns immediately, or storing the reflector and then applying all the reflectors at once to each column. This is where the two algorithms take different approaches.

In the left-looking Householder QR algorithm, all the reflectors are computed and stored in V. When computing columns R_k and V_k, all of $V_1, \ldots, V_{k-1}$ are applied to R_k and V_k at once. After that, a reflector is computed from V_k, which is simply A_k modified by all the reflections, and stored back into V_k.

Each time one of the reflectors is applied to a column, there is a possibility of fill-in being introduced into that column. This result comes from [8, Theorem 5.2], which states that the sparsity pattern of any row V_i, after applying the Householder reflector F, is the union of all rows modified by that Householder vector.

Thus, there is a chance that columns will become denser as each successive reflection is applied. In fact, the end result can be a completely dense column. Because the sparsity pattern of the reflector computed from V_k is the same as the sparsity pattern of that column, there is a possibility that the reflector will be completely dense. Therefore the matrix V which holds all the reflectors could become very dense, and for large initial A, storage becomes an issue. In some cases, if A is large and structured in such a way that reflectors become very dense, storing V can be impossible.

The right-looking Householder QR algorithm computes the reflector from V_k, and applies it to columns $k+1$ through n immediately. Once the reflector is computed and applied to all the appropriate columns, it does not have to be stored. Instead, we store the result of applying the reflector to each column. In this case, we can choose to make each result as sparse as we would like. The trade-off between this strategy and the one in which the sparse reflectors are stored themselves, is discussed in Section 4.3.

Because we only use the matrix R for preconditioning and the right-looking method allows us to compute the "Q-less" QR factorization, we use this variant in the implementation of our p-incomplete Householder QR factorization.

4.2 Complete Householder QR

As with algorithms for sparse Cholesky, sparse QR factorization algorithms proceed in two phases—symbolic and numeric. The symbolic factorization of A is computed first because it does not depend on the numerical values of the entries. The elimination tree of A^TA is used to compute the sparsity patterns of R and V. To see why this can be done, we consider the fact that $A^TA = R^TR$,

where R is the QR factor of A. More details on this can be studied in proof of Theorem 5.3 in [8]. Symbolic phase computes the total number of entries in R and V which is needed for allocating enough space for these matrices in the numerical factorization. The bulk of the work of the method, however, lies in the numeric phase.

The numerical phase of the Householder QR factorization is computed according to Algorithm 3. In this section, we describe the work done to compute the k^{th} column of the factor.

First, sparsity patterns of the k^{th} column R_k of R and the k^{th} column V_k of V are computed. Computing the pattern of R_k is done by first finding the leftmost entry in the row corresponding to each entry in the k^{th} column of A, and then traversing the elimination tree from that node to the root node. In other words, we go down A_k, and for each nonzero entry in A_k we find the leftmost entry in its row. Then we traverse the elimination tree from the node denoting the column index of this leftmost entry all the way up to the root. Each node encountered in the traversal is a row index of a nonzero entry in R_k. If the entry is below diagonal, i.e., its row index is less than k, then it is added to the pattern of V_k. During this process, all nonzero entries of A_k are assigned to a working vector x of length n.

The second step is to compute a value for each entry in the sparsity pattern. Since x holds A_k, all the reflectors corresponding to columns 1 through $k-1$ are applied to x. Then, row indices and values for each nonzero entry of R_k (known from the pattern computed in the previous step) are copied from x to R. Values of entries in V_k are stored next. This is accomplished by keeping track of the number of entries present in V at the beginning of column k, and the number of entries at the end of column k. The difference between these two is clearly the number of nonzeros in V_k.

The last step, once all the reflectors have been applied and all entries stored in their corresponding matrices, is to compute the reflector v and scalar β, such that (4.4) is satisfied with $x = V_k$.

It is important to note that in the process of computing R_k and V_k, all reflectors $V_1, \ldots, V_{k-1}$ are applied to R_k and V_k first. Only then is the reflector V_k computed. This is the characteristic of the left-looking algorithm described in the previous section.

4.3 Dropping Strategy

For a large matrix A, storing a possibly dense complete factor Q might be infeasible. Incomplete factorization deals with this problem by dropping entries according to some pre-established rule. The sparsity of the end result depends largely on the dropping rule chosen. On the other hand, the quality of incomplete factorization has to be considered, since dropping too many entries might result in a factorization that is too far away from the complete one and therefore not very useful for preconditioning.

There are three possible strategies for dropping elements during the factor-

Algorithm 3: Numeric phase of the left-looking complete Householder QR factorization.

Input: Sparse matrix A
Output: Upper triangular sparse matrix R, lower triangular sparse matrix V
for *k=1,...,n* **do**
  Compute nonzero pattern of $R(:,k)$;
  Compute nonzero pattern of $V(:,k)$;
  Store A(:,k) in x;
  foreach *Nonzero entry R(i,k)* **do**
    Apply reflector $V(:,i)$ to $R(i,k)$;
    Store result in x;
    Copy $x(i)$ to $R(i,k)$;
  foreach *Nonzero entry* $V(i,k)$ **do**
    Copy $x(i)$ to $V(i,k)$;
  Compute reflector $V(:,k)$ and β_k;
Finalize and return;

ization, each with its own benefits and drawbacks. Here we describe all three and give reasons why one is chosen over the other two.

The first strategy involves computing complete reflectors, the same way it was done in the complete factorization, and applying these full reflectors to all the appropriate columns according to the right-looking rule described in Section 4.1. The full results of these reflections are then stored in the matrix R and dropping is done as a post-processing step, once all the reflectors have been applied to all the appropriate columns.

While this strategy has the benefits of being the closest to the complete factorization (because the entries are computed exactly and dropped at the very end) and being the simplest one to implement, it also has some undesirable qualities. Since all the reflectors and all the reflections are computed completely, there are no improvements in speed and efficiency of the algorithm as compared to the complete algorithm. Also, a complete factor R is stored in the intermediate steps, before the dropping process, and therefore there are no savings in memory requirements either.

The second strategy is to compute full reflectors, and drop all but p largest fill-in entries each time a reflector is applied to a column. Of course, since the reflectors are computed from those columns from which the entries were already dropped, they will be sparser than the complete reflectors.

The benefit of this strategy is that throughout the computation only the incomplete results are stored. This means that the storage required is set to the number of entries in A plus p fill-in entries per column. Also, since the reflectors are sparse less entries are affected by each reflector, and therefore less computation needs to be done. This improves the speed of the overall algorithm. On the other hand, the entries are dropped each time a reflector is applied is to

a column, which makes the factorization less exact and therefore further from the complete factorization.

Third and final strategy considered here is computing the reflectors and then dropping entries before they are applied to all the appropriate columns. In this case, the entries are dropped from the results as well, the same way as in the previous approach. The benefit is that the reflectors are sparse and so when they are applied to each column the results are sparser than they would be when complete vectors are applied. Therefore, there is even more improvement in speed as compared to the previous method. The drawback is that the factors become too sparse and thus, too far away from the complete factorization.

Considering all the benefits and drawbacks of all three methods, we chose the second method as the basis for our implementation of p-incomplete Householder QR factorization. It is close enough to the complete factorization to be considered useful in preconditioning, and the storage requirements are predictable.

4.4 p-Incomplete Householder QR

The p-incomplete Householder QR factorization algorithm uses the same dropping rule that was previously used in the Cholesky factorization: for each column we drop all but the p largest fill-in elements. The difference for the QR factorization is that elements from each column are dropped numerous times, more precisely each time a reflector is applied to the column, as opposed to dropping only once as in the p-incomplete Cholesky factorization.

Our implementation is described in detail in Algorithm 4, which is described in more detail below. The starting point for this implementation is the complete Householder QR summarized in Algorithm 3. The major distinction between the two is that the complete algorithm is left-looking and the incomplete algorithm is right-looking.

Similarly to the complete Householder factorization, the first step in p-incomplete factorization is to compute the sparsity pattern of each column. While the sparsity pattern of each column in the complete algorithm is computed on the fly, i.e., for each column separately, in the incomplete algorithm the entire sparsity patterns of both R and V are computed at once. The reason for this is that the columns are computed from left to right, starting at the first column, but the reflectors are applied to columns on the right, which have not been computed yet. Therefore, we need to know the sparsity pattern of these columns in order to know how to apply the reflectors, i.e., to know which entries will be affected by each reflector. We compute the sparsity pattern of the full factorization and decide which p entries to keep later on, during the numerical factorization.

The next step is to prepare for the numerical factorization by copying all entries in upper triangular part of A into R, and all entries in the lower triangular part of A into V. This way the reflectors can be applied directly to the appropriate columns in these matrices, and there is no need for a working vector, like vector x in Algorithm 3.

Algorithm 4: Numeric phase of the right-looking p-incomplete Householder QR factorization.

Input: Sparse matrix A
Output: Upper triangular sparse matrix R, lower triangular sparse matrix V
for k=*1,...,n* **do**
  Compute nonzero pattern of $R(:,k)$;
  Compute nonzero pattern of $V(:,k)$;
for k=*1,...,n* **do**
  Copy $A(1:k,k)$ into $R(1:k,k)$;
  Copy $A(k+1:n,k)$ into $V(k+1:n,k)$;
for k=*1,...,n* **do**
  Compute Householder reflector v and β;
  Apply v and β to all columns $k+1$ through n of R and V;
  Mark fill-in vs. non fill-in entries in R and V;
  Put all fill-in into a heap for R and V;
  Extract largest p fill-in from R heap and V heap;
  Store p largest fill-in and original entries in R and V;
Finalize and return;

Once R and V have been initialized, we can begin the numerical part of the computation. First, a Householder reflector v and a scalar β are computed which satisfy (4.3) and (4.4), with $x = V_k$. Then v and β are applied to all columns to the right of the current column in both R and V, i.e., if the column being computed is k, the Householder reflector is applied to $R_{k+1}, \ldots, R_n$ and $V_{k+1}, \ldots, V_n$.

Since our strategy for computing p-incomplete factorization is to drop after a reflector is applied to a column, the next step is to decide which p fill-in elements should be kept in addition to the elements which were already structurally present in A. This is done by storing all fill-in elements from R_k and all fill-in elements from V_k into two separate heaps described in Section 2.2. The largest element is then extracted from each of the heaps p times, and the heap property is restored each time.

The chosen p fill-in elements for each of R_k and V_k are then stored along with the non fill-in elements in their respective matrices.

In order to see the distinction between this incomplete algorithm and the complete algorithm described previously, it is important to note that each of the reflectors is computed first and then applied to all the appropriate columns. This is the characteristic of the right-looking algorithm described in Section 4.1.

Chapter 5

Incomplete Factorization Preconditioning

This chapter describes the results of applying our incomplete factorizations to precondition systems that arise in solving linear programs and linear least squares problems.

5.1 Interior-point Methods for Linear Programming

Interior-point (IP) methods are an effective way of solving LPs and QPs. The IP methods work on the basis of satisfying the inequality constraints strictly, which is where their name initially came from [30]. Primal-dual methods are a subgroup of IP methods which simultaneously solve the primal and the dual of an LP or a QP. In this section, we summarize the steps of a primal-dual IP method for an LP, but the same ideas can be easily extended to QPs.

An LP in standard form is given by

$$\begin{array}{ll} \underset{x}{\text{minimize}} & c^T x \\ \text{subject to} & Ax = b,\ x \geq 0, \end{array} \tag{P}$$

and its dual is given by

$$\begin{array}{ll} \underset{y,z}{\text{maximize}} & b^T y \\ \text{subject to} & A^T y + z = c,\ z \geq 0, \end{array} \tag{D}$$

where y is the vector of dual variables, and z is the vector of dual slacks. Duality theory, which explains the relationship between feasible sets of (P) and (D), states that dual objective gives a lower bound on the primal objective and vice versa [30]. In particular, this means that the two objective functions coincide at the solution, i.e., given x_* that solves (P), and (y_*, z_*) that solves (D), we have that $c^T x_* = b^T y_*$ and that $x_*^T z_* = 0$.

Solutions to (P) and (D), must satisfy the following optimality conditions,

otherwise known as Karush-Kuhn-Tucker conditions:

$$A^T y + z = c, \tag{5.1a}$$
$$Ax = b, \tag{5.1b}$$
$$x_i z_i = 0, \quad i = 1, 2, \ldots, n, \tag{5.1c}$$
$$(x, z) \geq 0. \tag{5.1d}$$

Conditions (5.1a) and (5.1b) are feasibility conditions for the primal and the dual respectively, and (5.1c) is the complementarity condition, since it states that x and z must have zeros in complementary positions. A triple (x_*, y_*, z_*) is a primal-dual solution of LP if and only if it satisfies (5.1a)-(5.1d).

Primal-dual interior point methods find the solution to system of equations (5.1a), (5.1b) and (5.1c), and enforce the nonnegativity condition (5.1d). This is accomplished by the mapping

$$F(x, y, z) = \begin{bmatrix} A^T y + z - c \\ Ax - b \\ XZe \end{bmatrix} = 0 \quad \text{and} \quad (x, z) \geq 0, \tag{5.2}$$

where $X = \text{diag}(x_1, x_2, \ldots, x_n)$, $Z = \text{diag}(z_1, z_2, \ldots, z_n)$ and $e = (1, 1, \ldots, 1)^T$ and applying Newton's method. Each iteration of Newton's method for (5.2) requires the solution of the system

$$J(x, y, z) \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = -F(x, y, z),$$

where J is the Jacobian of F. Substituting in F from (5.2), and its corresponding Jacobian, we get

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ Z & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} c - A^T y - z \\ b - Ax \\ XZe \end{bmatrix}. \tag{5.3}$$

The central path is defined by the set of solutions of (5.2), where the complementarity condition is changed to

$$x_i z_i = \mu, \quad i = 1, 2, \ldots, n.$$

Thus, (5.3) becomes

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ Z & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} c - A^T y - z \\ b - Ax \\ XZe - \mu e \end{bmatrix}.$$

Eliminating Δz from the above produces a 2-by-2 system

$$\begin{bmatrix} -X^{-1}Z & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} c - A^T y - z + \mu X^{-1} e - Ze \\ b - Ax \end{bmatrix}, \tag{5.4}$$

which is usually referred to as the augmented system.

In order to improve the conditioning of the linear algebra subproblems, the LP can be regularized and an interior point method applied to the problem

$$\begin{array}{ll} \underset{x,r}{\text{minimize}} & c^T x + \frac{1}{2}\rho\|x - x_k\|^2 + \frac{1}{2}\delta\|r + y_k\|^2 \\ \text{subject to} & Ax + \delta r = b,\ x \geq 0, \end{array}$$

where x_k and y_k are approximate solutions. This approach is based on using proximal-point terms to regularize the LP. They have the benefit that the solution of the original LP is recovered as $x_k \to x_*$ and $y_k \to y_*$. The proximal-point term $\frac{1}{2}\rho\|x - x_k\|^2$ is the primal regularization, and the proximal-point term $\frac{1}{2}\delta\|r + y_k\|^2$ is the dual regularization, where the nonnegative scalars ρ and δ are primal and dual regularization parameters, respectively. Primal-dual IP methods are used on regularized LPs as well, and the procedure is identical to the one described step by step above, except that the regularization terms are included. The augmented system obtained for this problem is

$$\begin{bmatrix} -X^{-1}Z - \rho I & A^T \\ A & \delta I \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} c - A^T y - z + \mu X^{-1} e - Ze \\ b - Ax \end{bmatrix}. \tag{5.5}$$

The matrix in (5.5) is symmetric quasi-definite (see Section 1.3.1) because both $X^{-1}Z + \rho I$ and δI are positive definite. This comes from the fact that both X and Z are diagonal matrices with positive entries and therefore so is $X^{-1}Z$. Also, ρ and δ are nonnegative. Thus, the system is strongly factorizable and a Cholesky factorization exists for all symmetric permutations.

5.1.1 The Normal Equations

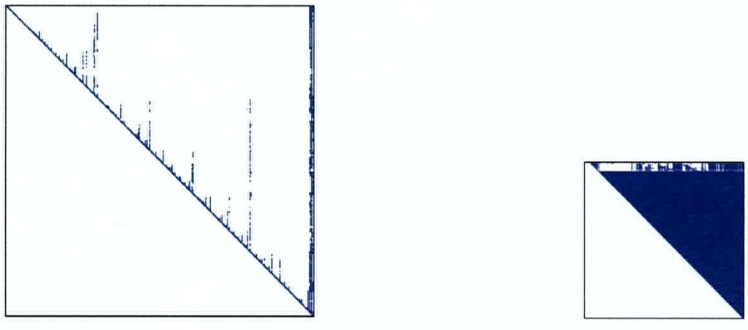
By eliminating Δx from (5.4), we obtain the normal equations

$$AH^{-1}A^T\Delta y = AH^{-1}f_1 + f_2,$$

where $H = X^{-1}Z$, $f_1 = c - A^T y - z + \mu X^{-1}e - Ze$, and $f_2 = b - Ax$. The matrix $AH^{-1}A^T$ is symmetric positive definite and a Cholesky factorization can be applied to solve it. However, Gill et al. [11] showed that even if the original LP and the augmented system are well conditioned, these normal equations can be ill-conditioned. Therefore, solving the augmented system directly is numerically favorable.

A second reason as to why the augmented system is better than the normal equations comes from comparing the sparsity patterns of the two systems. In a case where A has one dense or nearly dense column, $AH^{-1}A^T$ is dense and thus its Cholesky factor is dense as well. On the other hand, symmetric permutations can be applied to the augmented system in order to minimize fill-in in the Cholesky factors. Figure 5.1 shows Cholesky factors of the augmented system of the Netlib LP `lp_cre_a`, and its Schur complement. The figure on the left is the $(m+n)$-by-$(m+n)$ factor of the augmented system, which has 49,000 nonzero entries; the figure on the right is the n-by-n factor of the Schur complement,

which has 4.9 million nonzero entries. In practice, there exist methods for handling dense columns, for example see [1], but they are not further discussed in this work. Instead, we focus on solving the augmented system.



(a) (b)

Figure 5.1: Sparsity patterns of the Cholesky factor U of (a) the augmented system (49,000 nonzeros) and of (b) the Schur complement (4.9 million nozeros)

5.1.2 Preconditioning

Many good iterative tools are available readily, but the one chosen for solving SQD systems in this work is SYMMLQ, which takes advantage of the symmetry of the system. SYMMLQ [21] is a conjugate-gradient type iterative method for solving systems of linear equations

$$Ax = b,$$

where A is symmetric but not necessarily positive definite. It attempts to solve the linear system within the tolerance specified, and either converges or stops when it reaches the maximum number of iterations specified. Throughout this work a default tolerance of 10^{-6} is maintained and the maximum number of iterations is 5000. SYMMLQ was chosen as an iterative solver, over MINRES, because in this work we are dealing with ill-conditioned systems that are compatible, and SYMMLQ has been shown to produce smaller residuals than MINRES in this case [26].

Because it falls into the category of Krylov subspace methods, the number of SYMMLQ iterations depends on the clustering of eigenvalues of the system being solved. As discussed in the example in Section 1.3.5, if the matrix has only two distinct eigenvalues, a Krylov subspace method converges in at most two iterations. Therefore, the goal of preconditioning is to produce systems that have tightly clustered eigenvalues.

The procedure for solving an SQD system

$$K = \begin{bmatrix} -E & A^T \\ A & F \end{bmatrix} \tag{5.6}$$

is the following. First, we use the approximate minimum degree (AMD) reordering algorithm (see Section 2.5.1) to compute a permutation P which orders (5.6) in a way that minimizes fill-in in the factors. In the cases where we have sets of matrices with the same sparsity pattern but different values, we can do this reordering only once.

The next step is to compute a p-incomplete Cholesky $U^TDU = P^TKP$ using Algorithm 2 in Section 3.4. The factorization produces an incomplete unit upper triangular factor U and a diagonal factor D. Since SQD systems are indefinite, D has both positive and negative entries. SYMMLQ requires the preconditioner to be positive definite and therefore, the signs of any negative entries in D have to be flipped such that only positive entries are present. In other words, we compute

$$\bar{D} = |D|,$$

so that $U^T\bar{D}U$ is symmetric positive definite (SPD). If the factorization is complete, multiplying the inverse of this SPD matrix with the original system gives a diagonal matrix with ± 1 entries, $\bar{I} = \text{diag}(\pm 1)$, as shown in Section 3.4. In the p-incomplete case, this product is an approximation to $\bar{I}$, where the closeness of the approximation depends on the value of p.

In some cases, entries in $\bar{D}$ maybe be very small in magnitude, because of the inexactness of the p-incomplete factorization. If this is the case, the entries are modified to equal some specified threshold. Currently, this threshold value is chosen by the user, but in the future implementations it should be chosen such that the norm of the factor U is bounded. The effect of this modification on the quality of the preconditioner is shown in Section 5.1.3.

The last step in the preconditioning procedure is to use

$$M_1 = U^T\bar{D}^{\frac{1}{2}},$$

as a preconditioner for SYMMLQ. With this preconditioner, SYMMLQ is actually solving the preconditioned system

$$M^{-\frac{1}{2}}AM^{-\frac{1}{2}}y = M^{-\frac{1}{2}}b,$$

where

$$M = M_1M_1^T = U^T\bar{D}U.$$

Because M is an approximation to A, we expect $M^{-\frac{1}{2}}AM^{-\frac{1}{2}}$ to be an approximation to $\bar{I}$, which we know only has two eigenvalues. Therefore, we expect this product to have better eigenvalue clustering than the original system. Because the closeness of the approximate factorization to the complete factorization depends on the value of p, we expect the eigenvalue clustering to improve as p is increased.

5.1.3 Numerical Experiments

Figure 5.2 shows the results of applying the preconditioner with various values of p to SYMMLQ. The horizontal axis represents the values of p. The vertical axis represents the ratio of the number of iterations taken by SYMMLQ when a p-incomplete preconditioner is applied and the number of iterations taken when no preconditioner is applied. We expect the number of iterations to be the highest when no preconditioner is applied, and the number of iterations with the preconditioner to be a small fraction of that, depending on the value of p. The problems tested here come from the Netlib library of LPs [20], and the figure shows, for each LP, results of applying SYMMLQ to a single SQD system drawn from the tenth iteration of the IP method.

Each line in Figure 5.2 represents one LP, and we can see that for most of them we get the results we expected, i.e., the ratio is small even for small values of p. For five out of eight problems shown in the figure the number of iterations of SYMMLQ with p-incomplete preconditioner is less than 25% of that for the unpreconditioned system for each value of p, which can be seen by the clustering on the bottom of the graph. Thus, even for very small values of p, the number of iterations taken by SYMMLQ is decreased drastically when a preconditioner is used. In most cases, even with $p = 0$, meaning that no fill-in was allowed in the factors, number of iterations taken for the preconditioned system is only 20% of the number of iterations taken for the original system.

SQD systems tend to get more ill-conditioned with each iteration of IP method, and because our SQD system come from the tenth iteration, we do not get the results we expected for some of the test problems. For example, `lp_agg`, `lp_agg2`, and `lp_agg3` have ratios that oscillate with the value of p, meaning the the number of iterations goes up as p is increased, which is explained by the extreme ill-conditioning of these systems at the advanced stage of the IP method.

The drastic reduction in the number of iterations of SYMMLQ for the preconditioned system is, of course, a great result, but the trade-offs of this approach must be considered as well. First, we need to consider the time taken to compute the preconditioner. This is equivalent to the time taken to compute the Cholesky factorization of the SQD system, and this should be less than the time taken for the additional SYMMLQ iterations needed if the preconditioner is not applied. From Table 5.1, we can see that this is the case for all of our problems. Factorization time goes up only slightly as p is increased, but the time taken by SYMMLQ to compute the solution goes down by a factor of 2 or 3 each time p is increased by 2, in most cases. For $p = 10$, the reduction factor varies from 2 to as high as 15 for `lp_scsd8`. Only one problem, `lp_agg3`, shows the increase in time as p is increased.

Secondly, the number of nonzeros in the factor goes up as the value of p is increased and thus, more storage is required. From Figure 5.3, we can see that for most of our problems the storage required is only 20–50% (for various values of p) of the storage required for the complete factorization. Looking at the same problems in Figure 5.2, we see that the number of iterations of the pre-

		No precon		With p-incomplete precon				
Name	$n+m$	itns	T	p	itns	fact T	SYMMLQ T	total T
lp_agg	1103	1218	0.54	0	4603	0.02	25.25	25.27
				2	1827	0.22	10.20	10.42
				4	2443	0.22	13.55	13.77
				6	536	0.02	3.01	3.03
				8	180	0.02	1.14	1.16
				10	56	0.02	0.31	0.33
lp_agg2	1274	2297	1.18	0	2002	0.03	13.13	13.16
				2	3505	0.23	22.31	22.54
				4	2315	0.03	15.28	15.31
				6	817	0.03	5.42	5.45
				8	359	0.03	2.40	2.43
				10	194	0.03	1.25	1.28
lp_agg3	1274	1927	0.97	0	2636	0.03	16.52	16.56
				2	4688	0.23	30.99	31.22
				4	4332	0.03	28.34	28.37
				6	3956	0.23	25.53	25.76
				8	320	0.03	2.32	2.35
				10	318	0.03	2.12	2.15
lp_d6cube	6599	3347	36.29	0	325	2.12	11.41	13.53
				2	944	2.11	32.81	34.92
				4	303	2.10	10.55	12.65
				6	127	2.31	4.53	6.84
				8	134	2.11	5.03	7.14
				10	142	2.10	5.10	7.20
lp_scsd6	1497	4966	2.82	0	151	0.04	1.16	1.20
				2	109	0.03	0.86	0.89
				4	123	0.03	0.89	0.92
				6	38	0.03	0.29	0.32
				8	26	0.24	0.06	0.30
				10	8	0.24	0.02	0.26
lp_scsd8	3147	4687	13.79	0	412	0.55	5.71	6.26
				2	99	0.35	1.47	1.82
				4	61	0.54	0.68	1.22
				6	48	0.54	0.63	1.17
				8	73	0.55	0.94	1.49
				10	27	0.54	0.33	0.87
lp_ship08l	5141	4536	16.37	0	1193	1.19	26.50	27.69
				2	831	1.24	18.36	19.60
				4	615	1.20	13.54	14.74
				6	661	1.40	14.77	16.17
				8	443	1.20	9.89	11.09
				10	430	1.20	9.62	10.82
lp_truss	9806	4416	41.83	0	836	4.45	37.51	41.96
				2	321	4.25	14.71	18.96
				4	147	4.20	6.60	10.80
				6	132	4.20	5.99	10.19
				8	96	4.22	4.51	8.73
				10	77	4.46	3.90	8.36

Table 5.1: Timings for factorization and SYMMLQ for increasing p on various LP problems. "T" denotes time in seconds.

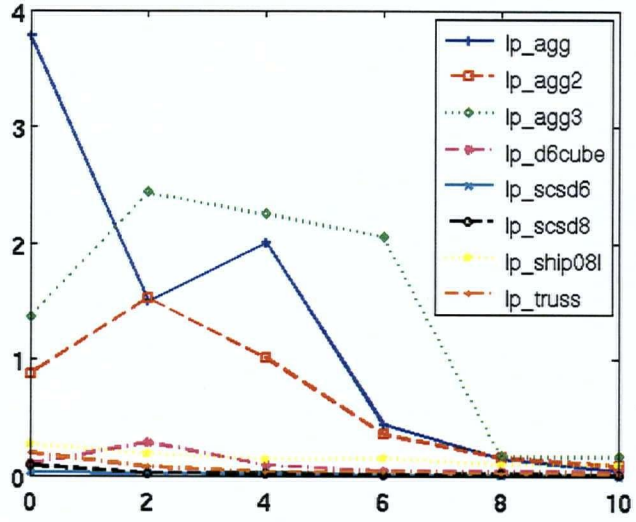


Figure 5.2: Ratio of the number of iterations of SYMMLQ with and without p-incomplete preconditioner for various p. For five out of eight problems number of iterations with p-incomplete preconditioner is less than 50% of the number of iterations without the preconditioner.

conditioned system has 70–90% decrease as compared to the unpreconditioned system. In other words, the storage required for the p-incomplete preconditioner is only a fraction of the storage required for the complete factorization, and the number of iterations is drastically reduced as compared to the unpreconditioned system. Therefore, we conclude that this increased storage requirements is a fair price to pay for such a drastic decrease in the number of iterations. In the cases where even this increase in storage is infeasible, we can use $p = 0$ which does not require any additional storage, and the decrease in the number of iterations is still 75–80% in most cases. For some problems, however, p-incomplete preconditioner with $p = 0$ gives very poor results, as can be seen for `lp_agg` and `lp_agg2` in the figure.

As mentioned earlier, the diagonal entries in $\bar{D}$ sometimes need to be modified in order to improve the norm of the triangular factor of the preconditioner. Figure 5.4 shows that this has the most effect on the systems which are ill-conditioned to begin with, namely `lp_agg`, `lp_agg2` and `lp_agg3`. For these systems increasing the minimum magnitude threshold for the diagonal entries improves the conditioning of $U^T\bar{D}U$, and thus, reduces the number of iterations necessary when this preconditioner is applied. For the other five systems clustered on the bottom of the figure, the ratio of number of iterations for the preconditioned system and the number of iterations for the unpreconditioned

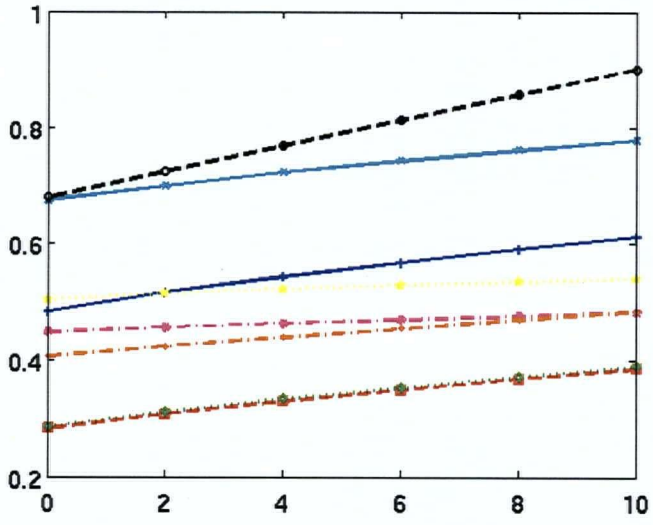


Figure 5.3: For each problem, ratio of the number of nonzeros in the p-incomplete Cholesky factor to the full Cholesky factor for various p.

system remains low, about 10–15% regardless of the magnitudes of the diagonal entries in $\bar{D}$. Thus, we conclude that modifying the entries in D in this way, does not have a significant effect on the majority of the problems tested, but does show an improvement for the problems that are known to be extremely ill-conditioned.

5.2 Least squares

The goal of the least-squares (LS) problem, introduced in Section 1.2.1, is to minimize the 2-norm of the residual

$$r = b - Ax.$$

Regularization is often introduced into LS problems in order to improve conditioning and thus they are usually encountered in the form

$$\underset{x}{\text{minimize}} \quad \tfrac{1}{2}\|Ax - b\|^2 + \tfrac{1}{2}\delta\|x\|^2. \tag{5.7}$$

By differentiating (5.7) and equating it to zero, we get the first order optimality conditions

$$-A^T r + \delta x = 0 \quad \text{and} \quad r = b - Ax.$$

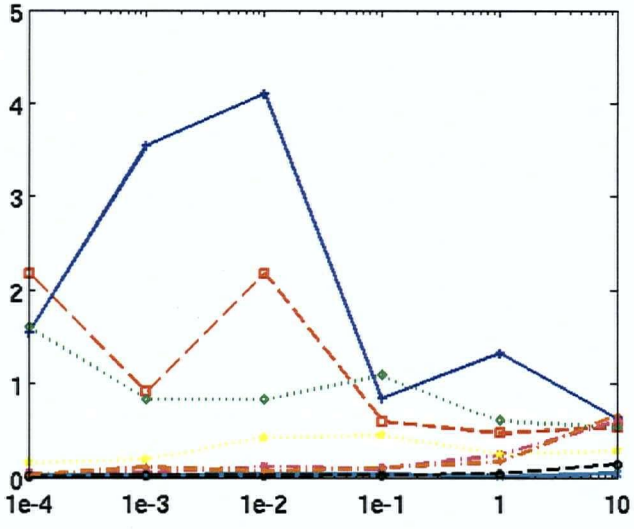


Figure 5.4: For each problem, ratio of the number of iterations of SYMMLQ with and without the p-incomplete preconditioner, for increasing threshold values for diagonal entries in D. Larger threshold values imply triangular factors with smaller norm.

Combining these two linear equations together gives the linear system

$$\begin{bmatrix} I & A \\ A^T & -\delta I \end{bmatrix} \begin{bmatrix} r \\ x \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}.$$

This system is in the category of SQD systems, discussed in Section 1.3.1, because both I and δI are symmetric positive definite matrices.

The regularized augmented system (5.5) can be reformulated as the least squares problem

$$\underset{x}{\text{minimize}} \left\| \begin{pmatrix} (H+\rho I)^{-\frac{1}{2}} A^T \\ \sqrt{\delta} I \end{pmatrix} x - \begin{pmatrix} (H+\rho I)^{-\frac{1}{2}} f_1 \\ f_2/\sqrt{\delta} \end{pmatrix} \right\|,$$

where $H = -X^{-1}Z$, $f_1 = c - A^T y - z + \mu X^{-1} e - Ze$, and $f_2 = b - Ax$. It is important to note that this reformulation is only possible for the dual regularized augmented system. In other words, if the $(2,2)$ block of (5.5) is 0, it would not be possible to compute $1/\sqrt{\delta}$, and thus, the LS reformulation would fail.

5.2.1 Alternatives

The most common way of solving the generic LS problem

$$\underset{x}{\text{minimize}} \quad \|Ax - b\|^2 \tag{5.8}$$

is via the normal equations method [13]. This method solves (5.8) by first computing

$$C = A^T A \quad \text{and} \quad d = A^T b.$$

Then a Cholesky factorization of C is computed

$$C = GG^T,$$

and the equation

$$Gy = d$$

is solved for y. Finally,

$$G^T x = y$$

is solved for x, where x is the least squares solution.

The problems that arise in this method are similar to the problems discussed in Section 5.1.1, where the Cholesky factor of the symmetric positive definite matrix $A^T A$ is very dense. Also, in general, it is not a good idea to form this matrix as loss of information might occur in this process [13].

As mentioned in the previous chapter, another way of solving LS problems is computing the QR factorization of the matrix A and then solving the problem

$$\underset{x}{\text{minimize}} \quad \|Q^T Ax - Q^T b\|,$$

where Q is the orthogonal factor. Since $A = QR$ and $Q^{-1} = Q^T$, we know that $Q^T A = R$, which is an upper triangular matrix. Therefore the above problem is simplified to a problem with a triangular matrix, which is a lot easier to solve than a general system.

Similar problems arise in this method as in the previous one, when applied to a large sparse system, because again we have to compute the full QR factorization where the factors could be arbitrarily dense.

5.2.2 Preconditioning

We now turn to iterative solvers for sparse linear least squares problems and use the iterative method LSQR [22] to solve this problem. LSQR is similar to SYMMLQ, discussed in Section 5.1.2, in the sense that they are both Krylov subspace methods, which in a nutshell means that their convergence rates depend on the clustering of the eigenvalues of the problem. Thus, the goal of our approach is improve the efficiency of this method by means of preconditioning the least squares problem with a preconditioner which improves the eigenvalue clustering of the problem.

To accomplish this, we precondition the linear least squares problem (5.7) with the factor R of the QR factorization. Thus, we get

$$\underset{y}{\text{minimize}} \quad \tfrac{1}{2}\|AR^{-1}y - b\|^2 + \tfrac{1}{2}\delta^2\|y\|^2 \quad \text{and} \quad Rx = y.$$

Because $A = QR$, we know that

$$Q = AR^{-1},$$

and because Q is orthogonal its eigenvalues are all on the unit circle. Therefore, we expect our Krylov subspace method to converge with one iteration on this preconditioned system.

However, we are interested in computing the p-incomplete QR factorization, and thus the preconditioned system AR^{-1} is an approximation to Q, rather than Q exactly. The closeness of this approximation depends on the value of p, i.e., the more fill-in entries we keep in the factor, the closer it will be to the true factor. As it turns out, even with very small value of p, the approximation is good enough to dramatically reduce the number of iterations taken by LSQR. We will see more on this in Section 5.2.3.

The procedure for preconditioning starts off with computing the AMD reordering of the symmetric positive definite system A^TA. This reordering turns out to minimize the fill-in in the QR factorization of A itself, because the Cholesky factorization of A^TA is R^TR, where R is the same as the upper triangular QR factor of A. In the case that A has one or more dense rows, these rows are removed before forming A^TA. A row is considered dense if it has more than $10\sqrt{n}$ entries, where n is the number of columns in A.

The next step is to compute p-incomplete QR factorization of reordered matrix A. The value of p is decided upon by the user, according to the amount of storage available and the quality of the preconditioner desired. In general, quality of the preconditioner is proportional with the value of p and the memory storage is, of course, inversely proportional. In some cases, the diagonal entries of the p-incomplete factor R are then modified using a user defined threshold value, in order to improve the conditioning of the preconditioner and finally, the factor is passed to LSQR.

The results of this procedure are presented in the next section.

5.2.3 Numerical Experiments

Results of preconditioning LSQR with a p-incomplete QR factorization turn out to be even better than the ones seen in Section 5.1.3 for preconditioning SYMMLQ with an incomplete Cholesky factorization. Figure 5.5 shows that even for a very small p, such as $p = 2$, the number of iterations of LSQR taken for the preconditioned problem is less than 10% of that for the unpreconditioned problem, in most cases. Again, like in Section 5.1.3, we have some deviations from this trend, but in this case even the ill-conditioned systems that created trouble for SYMMLQ work well in preconditioned LSQR. In fact, all but one problem (`lp_scsd6`), require less than 10% of iterations of the unpreconditioned system for $p = 4$.

The trade-offs in this case are very similar to those described in Section 5.1.3 for preconditioning SYMMLQ. The number of nonzeros in the factor goes up as p is increased. However, as Figure 5.6 shows, for $p = 2$ and $p = 4$ which were shown to be sufficient for drastic reduction of the number of iterations, the number of nonzeros in the factor is less than 10% of the number of nonzeros in the full QR factor, in most cases. This is definitely a small sacrifice for such a large reduction in the number of iterations of LSQR.

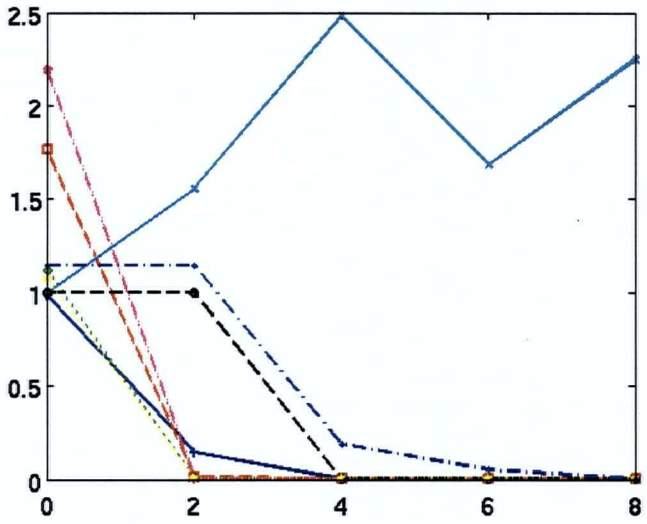


Figure 5.5: For each problem, ratio of the number of iterations of LSQR with and without p-incomplete QR preconditioner for various p.

The other trade-off that needs to be considered here is the time it takes to compute the p-incomplete QR factorization. This turns out to be significantly longer than the time for computing a p-incomplete Cholesky factorization for the same system. The main cause of such a high factorization time is the repeated application of the Householder reflectors to each column, and the dropping procedure needed each time a reflector is applied to a column. In comparison, our implementation of the Cholesky factorization computes the whole column at once, and drops the entries only once per column. Table 5.2 shows the times for factorization and the time that LSQR takes to solve the preconditioned system. In all cases, the number of iterations and therefore the time for LSQR is reduced by a large factor as p is increased. However, factorization time alone is significantly longer than the time to solve the unpreconditioned system. Therefore, we see that we have a high quality preconditioner, but its implementation seems to be slow.

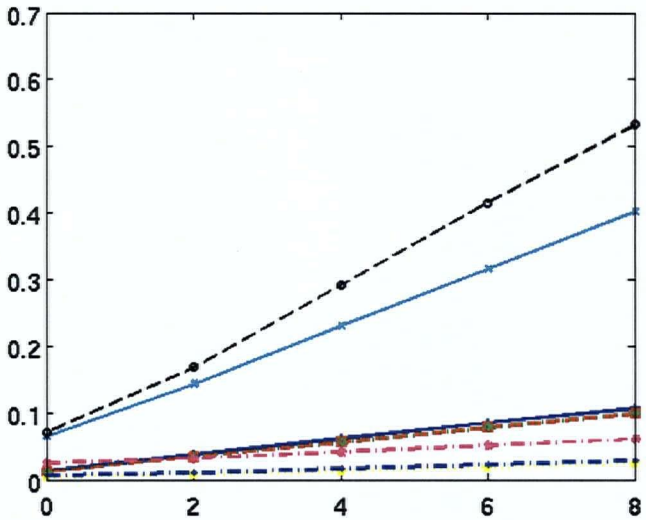


Figure 5.6: For each problem, ratio of the number of nonzeros in p-incomplete factor R and full factor R for various p.

			No precon		With p-incomplete precon					
Name	m	n	itns	T	p	itns	nnz(R)	fact T	LSQR T	total T
lp_agg	1103	488	1895	0.90	0	1867	511	19.27	1.79	21.07
					2	274	1463	19.23	0.67	19.90
					4	1	2416	22.80	0.01	22.81
					6	1	3350	23.77	0.01	23.78
					8	1	4164	23.85	0.01	23.86
lp_agg2	1274	516	2838	3.62	0	5000	538	29.07	14.52	43.59
					2	28	1548	26.92	0.02	26.94
					4	5	2552	27.63	0.01	27.64
					6	3	3532	28.81	0.01	28.82
					8	1	4492	30.00	0.01	30.01
lp_agg3	1274	516	4481	5.66	0	5000	538	28.55	15.04	43.59
					2	7	1548	29.13	0.01	29.14
					4	1	2543	27.89	0.01	27.90
					6	1	3533	29.28	0.01	29.29
					8	1	4491	29.97	0.01	29.98
lp_d6cube	6599	415	2279	10.82	0	5000	1968	75.61	32.28	107.89
					2	7	2553	76.76	0.01	76.77
					4	1	3303	77.42	0.01	77.43
					6	1	4090	77.64	0.01	77.65
					8	1	4904	77.97	0.01	77.98
lp_scsd6	1497	147	1197	1.25	0	1192	181	2.35	1.86	4.21
					2	1864	400	2.48	3.31	5.79
					4	2972	642	2.62	5.46	8.08
					6	2018	879	2.42	3.83	6.25
					8	2698	1122	2.44	5.11	7.55
lp_scsd8	3147	397	5000	7.82	0	5000	418	34.70	13.82	48.52
					2	5000	995	34.21	14.78	48.99
					4	5	1723	34.30	0.01	34.31
					6	3	2451	34.66	0.01	34.67
					8	17	3148	35.22	0.22	35.44
lp_ship08l	5141	778	1969	4.25	0	2113	802	216.07	9.46	225.53
					2	1	1684	209.89	0.01	209.90
					4	4	2781	212.61	0.01	212.62
					6	4	3986	212.99	0.01	213.00
					8	4	5441	215.66	0.20	215.86
lp_truss	9806	1000	4378	16.88	0	5000	1774	637.61	35.39	673.01
					2	5000	3019	637.32	37.84	675.16
					4	823	4668	641.30	6.87	648.17
					6	235	6221	649.31	2.08	651.39
					8	9	7908	649.50	0.03	649.53

Table 5.2: Timings for factorization and LSQR for increasing p on various LP problems. "T" denotes time in seconds.

Bibliography

[1] Knud D. Andersen. A modified schur-complement method for handling dense columns in interior-point methods for linear programming. *ACM Trans. Math. Softw.*, 22(3):348–356, 1996.

[2] Cleve Ashcraft, Roger G. Grimes, and John G. Lewis. Accurate symmetric indefinite linear equation solvers. *SIAM J. Matrix Anal. Appl.*, 20(2):513–561, 1999.

[3] Zhong-Zhi Bai, Iain S. Duff, and Andrew J. Wathen. A class of incomplete orthogonal factorization methods. I: Methods and theories. *Bit Numerical Mathematics*, 41(1):53–70, 2001.

[4] Åke Björck. *Numerical Methods for Least Squares Problems.* Society of Industrial and Applied Mathematics, Philadelphia, 1996.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, USA, 1990.

[6] Timothy A. Davis. Algorithm 849: A concise sparse Cholesky factorization package. *ACM Trans. Math. Softw.*, 31(4):587–591, 2005.

[7] Timothy A. Davis. CHOLMOD users' guide. `http://www.cise.ufl.edu/research/sparse/cholmod/`, 2005.

[8] Timothy A. Davis. *Direct methods for sparse linear systems*, volume 2 of *Fundamentals of Algorithms.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2006.

[9] Iain S. Duff, Albert M. Erisman, and John K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, New York, NY, 1986.

[10] Alan George and Joseph W. Liu. *Computer Solution of Large Sparse Positive Definite.* Prentice Hall Professional Technical Reference, 1981.

[11] Philip E. Gill, Walter Murray, Dulce B. Ponceleón, and Michael A. Saunders. Preconditioners for indefinite systems arising in optimization. *SIAM J. Matrix Anal. Appl.*, 13(1):292–311, 1992.

[12] Philip E. Gill, Michael A. Saunders, and Joseph R. Shinnerl. On the stability of Cholesky factorization for symmetric quasidefinite systems. *SIAM J. Matrix Anal. Appl.*, 17(1):35–46, January 1996.

[13] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, second edition, 1989.

[14] Anshul Gupta. Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices. *SIAM J. Matrix Anal. Appl.*, 24(2):529–552, 2002.

[15] Bruce Hendrickson and Edward Rothberg. Improving the run time and quality of nested dissection ordering. *SIAM Journal on Scientific Computing*, 20(2):468–489, 1998.

[16] Mark T. Jones and Paul E. Plassmann. An improved incomplete Cholesky factorization. *ACM Trans. Math. Softw.*, 21(1):5–17, 1995.

[17] Chih-Jen Lin and Jorge J. Moré. Incomplete Cholesky factorizations with limited memory. *SIAM J. Comput.*, 21(1):24–45, 1999.

[18] J A. Meijerink and Henk A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Mathematics of Computation*, 31(137):148–162, 1977.

[19] N. Munksgaard. Solving sparse symmetric sets of linear equations by preconditioned conjugate gradients. *ACM Trans. Math. Softw.*, 6(2):206–219, 1980.

[20] NETLIB linear programming library. `http://www.netlib.org/lp/data/`, 2006.

[21] Christopher C. Paige and Michael A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM Journal on Numerical Analysis*, 12(4):617–629, 1975.

[22] Cristhopher C. Paige and Michael A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Trans. Math. Software*, 8:43–71, 1982.

[23] Andreas T. Papadopoulos, Iain S. Duff, and Andrew J. Wathen. A class of incomplete orthogonal factorization methods. II: Implementation and results. *Bit Numerical Mathematics*, 45(1):159–179, 2005.

[24] Vladimir Rotkin and Sivan Toledo. The design and implementation of a new out-of-core sparse cholesky factorization method. *ACM Trans. Math. Softw.*, 30(1):19–46, 2004.

[25] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[26] Michael A. Saunders. Notes 2: Iterative Methods for Symmetric $Ax = b$. `http://www.stanford.edu/class/msande318/`, 2007.

[27] Lloyd N. Trefethen and David Bau III. *Numerical Linear Algebra.* Society of Industrial and Applied Mathematics, Philadelphia, 1997.

[28] Robert J. Vanderbei. Symmetric quasidefinite matrices. *SIAM J. Optim.*, 5(1):100–113, 1995.

[29] Xiaoge Wang, Kyle A. Gallivan, and Randall Bramley. CIMGS: An incomplete orthogonal factorization preconditioner. *SIAM J. Sci. Comput.*, 18(2):516–536, 1997.

[30] Stephen J. Wright. *Primal-Dual Interior-Point Methods.* Society of Industrial and Applied Mathematics, Philadelphia, 1997.

[31] Mihalis Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2(1):77–79, 1981.