Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 97-065

Robust Preconditioning for Sparse Linear Systems

by: Edmond Chow
(Ph.D. Thesis)

ROBUST PRECONDITIONING FOR SPARSE LINEAR SYSTEMS

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

EDMOND TEN-FU CHOW

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Yousef Saad, Advisor

December 1997

Acknowledgments

I wish to thank Yousef Saad, without whom this thesis would not be possible. His guidance was perfectly balanced to let me benefit from both his exceptional perception, and a freedom to discover at my own pace. Wei-Pai Tang introduced me to numerical linear algebra and scientific computing when I was an undergraduate at the University of Waterloo. I am grateful for his trust in me at that time, and for his continued advice and interest in my work and career. I also appreciate the encouragement and support of Michael Heroux, who gave me three summers' worth of time and resources at Cray Research (SGI). The BPKIT block preconditioning toolkit was motivated by him.

My teachers during the weekends and late-night hours were two of my officemates and friends, Kesheng Wu and Andreas Stathopoulos. Our endless discussions helped refine my understanding of iterative methods. Andrew Chapman and Barry Rackner deserve thanks for providing codes, advice, and support. Sandra Carney "showed me the ropes" when I first arrived in Minnesota. Shang-Hua Teng set an example for me, both as a friend and as a professional, and Tony Chan kept a watchful eye over me from UCLA. I also wish to thank my readers Daniel Boley and Graham Candler for their time and interest in my work. Above all, I wish to express my gratitude to my friends and family for their love and support.

Grateful acknowledgment is made to the publishers who provided permission to excerpt the following articles: [37, 43, 46], copyright 1997 and 1998 by the Society for Industrial and Applied Mathematics; [45], copyright 1997 by John Wiley & Sons, Ltd.; [42], copyright 1998 by the Association for Computing, Inc.; [44], copyright 1998 by Elsevier Science–NL.

This work was supported by the National Science Foundation under grants CCR-9618827 and CCR-9214116, by NASA under grant NAG2-904, by the Minnesota Supercomputer Institute, and by Cray Research (SGI).

— E. C.

To my parents.

Abstract

Preconditioned iterative methods have become standard linear solvers in many applications, but their limited robustness in some cases has hindered the ability to efficiently solve very large problems in some areas. This thesis proposes several new preconditioning techniques that attempt to extend the range of iterative methods, particularly to solving nonsymmetric and indefinite problems such as those arising from incompressible computational fluid dynamics.

First, we present an iterative technique to compute sparse approximate inverse preconditioners. This new technique produces approximate inverses comparable in quality with others in the literature, but at a lower computational cost, and with a simpler method to determine good sparsity patterns for the approximate inverse. This class of preconditioners is very attractive for parallel computing and avoids the stability problems that may occur with incomplete LU (ILU) preconditioners on nonsymmetric or indefinite matrices.

Next, we demonstrate more effective uses of approximate inverses than using them to precondition the entire matrix. We show how to use approximate inverse techniques to construct block preconditioners, particularly for the discrete fully-coupled incompressible Navier-Stokes equations and matrices that are partitioned by nonoverlapping domain decomposition. Approximate inverse techniques are used to generate sparse approximate solutions whenever these are needed in forming the preconditioner. The storage requirements for these preconditioners are much less than for ILU preconditioners for many tough, large-scale problems.

To try to improve ILU preconditioners, we experimentally analyze their causes of failure, particularly on indefinite linear systems. We categorize the failure modes and design statistics that can be routinely computed to monitor the factorization or determine the cause of failure. Through this better understanding, we show how these causes of failure can sometimes be circumvented through pivoting, reordering, scaling, perturbing diagonal elements, and preserving symmetric structure.

Finally, we present an object-oriented framework that implements some of the preconditioning algorithms above. The framework allows different data structures for block matrices to be used with the same preconditioners. Various methods are provided for inverting diagonal or pivot blocks, possibly with an approximate inverse technique.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

This thesis considers the problem of computing the solution x of the linear system

$$Ax = b \tag{1.1}$$

by an iterative method, when A is a large, sparse matrix. The convergence behavior of the iterative method depends on the properties of A, most usually, its definiteness and condition number. Other relevant properties of A are difficult to quantify, but these include whether or not the eigenvalues of A are clustered, and the separation of the extremal eigenvalues (those near the outside part of the spectrum) from other eigenvalues [142]. These properties governing convergence are fairly well understood when A is symmetric. When A is nonsymmetric, however, additional non-spectral properties also affect convergence, although these are poorly understood. Properties that may be relevant include A's departure from normality, and the conditioning of the eigenvectors of A.

The purpose of preconditioning is to improve the convergence properties of the problem (1.1) by transforming it, for example, into

$$M^{-1}Ax = M^{-1}b$$

with a matrix or operator M, called a *preconditioner*. The matrix $M^{-1}A$ is called the *preconditioned matrix*, and its properties now determine the convergence behavior of the iterative method. If $M = A$, then the preconditioned matrix is the identity, the ideal case for an iterative method. The operation $M^{-1}v$ for a vector v, however, should be much less expensive to compute than $A^{-1}v$. Preconditioners must be selected in such a way that the cost of using and constructing them is offset by the improved convergence properties they achieve.

Preconditioners of several types have become very successful in many applications, by helping solve very large linear systems quickly and reliably. In some other areas, however, it is difficult to find preconditioners that are robust. This thesis proposes several new preconditioning techniques

that attempt to extend the range of iterative methods, particularly to solving nonsymmetric and indefinite problems, such as those arising from the incompressible Navier-Stokes equations.

The properties of the matrices that arise in numerical simulations depend on several factors. Consider, as a motivating example, the numerical simulation of the flow of a Newtonian fluid. For simplicity, we will neglect body forces, thermal conduction, and heat sources. The equations modeling the flow are the Navier-Stokes equations over a domain with appropriate boundary conditions. In primitive variables, the equations are

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) = 0 \tag{1.2}$$

$$\frac{\partial}{\partial t}(\rho \boldsymbol{u}) + \nabla \cdot (\boldsymbol{u}\rho\boldsymbol{u} - \boldsymbol{\sigma}) = 0 \tag{1.3}$$

$$\frac{\partial}{\partial t}(\rho e) + \nabla \cdot (\boldsymbol{u}\rho e - \boldsymbol{\sigma} \cdot \boldsymbol{u}) = 0 \tag{1.4}$$

where t denotes the time, $\boldsymbol{u}$ the velocity, e the energy, ρ the density, and $\boldsymbol{\sigma}$ the stress tensor. The stress tensor for a Newtonian fluid is given by

$$\boldsymbol{\sigma} = -pI + \lambda(\nabla \cdot \boldsymbol{u})I + \mu\left(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T\right)$$

where p denotes the pressure, and μ and λ are the first and second coefficients of viscosity. Additional thermodynamic relations relate the pressure, density, and energy to form a full set of equations for the variables. Note that when the viscous forces are negligible (μ and λ are very small), the equations simplify to the Euler equations in the limit.

An important application is the modeling of incompressible fluids, i.e., when density is treated as constant. These models raise particular difficulties in their numerical solution. For further simplicity we consider the isothermal case,

$$\rho\left(\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u}\right) + \nabla p - \mu \nabla^2 \boldsymbol{u} = 0 \tag{1.5}$$

$$\nabla \cdot \boldsymbol{u} = 0 \tag{1.6}$$

again over a domain with appropriate boundary conditions. A popular test problem, and one that is used in this thesis, is the steady, square (2-D) lid-driven cavity problem [75]. In this case, the boundary conditions are $\boldsymbol{u} = (1,0)^T$ on the top edge of the square, and $\boldsymbol{u} = (0,0)^T$ on the other three sides and the corners. The reference pressure specified at the bottom-left corner is 0.

For a discretization of a 2-D problem (by the Galerkin Finite Element Method, for example) a fully-coupled solution method (i.e., all variables solved simultaneously) leads to a linear system of the following form that needs to be solved at each nonlinear iteration:

$$\begin{pmatrix} K_u & K_{uv} & -C_x \\ K_{vu} & K_v & -C_y \\ C_x^T & C_y^T & 0 \end{pmatrix} \begin{pmatrix} u \\ v \\ p \end{pmatrix} = \begin{pmatrix} F_u \\ F_v \\ b_c \end{pmatrix}. \tag{1.7}$$

If the fixed point iteration, or Picard method is used as the nonlinear solver, then u and v are vectors of discrete velocity components, p here is a vector of discrete pressures, the K submatrices compose a convection-diffusion operator plus the effect of temporal terms, the C submatrices are pressure gradient operators, and the C^T submatrices are velocity divergence operators. If a Newton method is used, then the matrix is a Jacobian, and u, v, and p are updates to the current nonlinear solution, while the right-hand side vector is the current residual.

The system (1.7) is *indefinite* (the symmetric part of the matrix has both positive and negative eigenvalues) because of the divergence-free continuity constraint, which is the source of the zero diagonal block in the global coefficient matrix. In addition, the system is generally nonsymmetric. These two properties make the system very difficult to solve for iterative methods. Note that for very slow flow, however, the convection term $\boldsymbol{u} \cdot \nabla \boldsymbol{u}$ is negligible and the resulting equations are the self-adjoint Stokes equations.

The time derivative is handled through a time marching procedure and can be implicit or explicit. In implicit methods, the size of the time steps is restricted by the convergence of the method, but small time steps also lead to a more diagonally dominant K submatrix, which makes linear systems involving K easier to solve. Another factor that influences the conditioning of K is the discretization parameter h, which describes the smallest resolution of the simulation. For many discretizations, the conditioning of the matrix varies with $1/h^2$. Thus highly accurate simulations, or simulations with cells or elements with very large aspect ratios, will give rise to poorly conditioned linear systems.

The block structure of the coefficient matrix in (1.7) arises from ordering together equations of the same type (e.g., x-direction momentum equations) as well as variables of the same type (e.g., x-direction velocity variables) in a system of partial differential equations (PDE's). When the variables are at least conceptually ordered this way, preconditioners can be designed to reduce the coupling between the different variables within each type of equation.

Discretizations of coupled PDE's also give rise to another form of block structure that can be exploited in preconditionings. This blocking may be imposed by ordering together the equations and unknowns at a single grid point, or those of a subdomain. Here, preconditioners can be designed to reduce the couplings between different equations in the PDE system, or between variables of a localized subdomain. The relative robustness of preconditionings that exploit block structure comes partly from being able to solve accurately for the strong couplings within these blocks.

Figure 1.1 shows a block matrix of dimension 24 with two equations per grid point (forming dense 2 by 2 blocks), and three grid points blocked together (forming sparse 6 by 6 blocks).[1] Note that each grid point may not always have the same number of variables, and the number of grid points in each large sparse block also do not need to be the same.

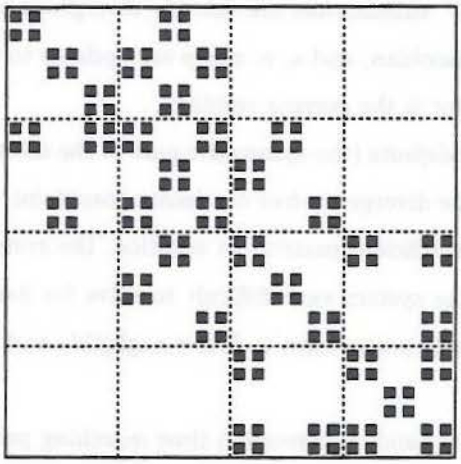


Figure 1.1: Block matrix structure arising from two equations per grid point, and three grid points blocked together.

Besides the fully-coupled solution method above, several other solution procedures are common when solving incompressible Navier-Stokes problems, often leading to linear systems that are definite and/or symmetric. These will be described below.

To create an explicit equation for pressure, or transform (1.7) into a positive definite system, the *penalty* method [25] modifies the continuity condition (1.6) as

$$\nabla \cdot \boldsymbol{u} = -\epsilon p$$

[1] A tool that is useful for visualizing the block structure and numerical values in a matrix is described by Bramley and Loos [26].

where ϵ is a small parameter. This adds slight compressibility to the fluid being modeled, and puts ϵI in the position of the zero block in (1.7). The new set of equations can be written

$$\begin{pmatrix} K & -C \\ C^T & \epsilon I \end{pmatrix} \begin{pmatrix} u \\ p \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}$$

and when p is eliminated, the system reduces to

$$\left(K + \tfrac{1}{\epsilon} C C^T\right) u = f_1 + \tfrac{1}{\epsilon} C f_2$$

which is positive definite, but poorly conditioned, with condition number on the order of $1/\epsilon$. Direct methods are almost always used to solve these linear systems. Our experience is that the penalty method creates linear systems that can be very difficult to solve for iterative methods.

A second approach is Uzawa's method [3], which is essentially a method on the reduced system. By writing (1.7) in simplified form,

$$\begin{pmatrix} K & -C \\ C^T & 0 \end{pmatrix} \begin{pmatrix} u \\ p \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}$$

the system is solved starting with u^0 and p^0, with the following iteration:

- Solve $Ku^{i+1} = f_1 + Cp^i$
- Update $y^{i+1} = y^i + \omega(C^T u^{i+1} - f_2)$

where ω is a scalar relaxation parameter. This is a Richardson iteration on the reduced system

$$-C^T K^{-1} C p = C^T K^{-1} f_1 - f_2.$$

At each iteration, a linear system involving K is solved, often with an iterative method. The method can be expensive if it is difficult to solve with K.

A related approach of handling the divergence-free continuity constraint is the *segregated* formulation [83], where the variables u, v, and p (and possibly temperature, chemical concentrations, and turbulence variables, if necessary) are not solved simultaneously, but separately within a loop that makes the final solution consistent. To describe this approach, we rearrange (1.7) and define

f_u and f_v so that

$$K_u u - C_x p = F_u - K_{uv} v \equiv f_u \tag{1.8}$$

$$K_v v - C_y p = F_v - K_{vu} v \equiv f_v \tag{1.9}$$

$$C_x^T u + C_y^T v = b_c. \tag{1.10}$$

An equation for pressure may be derived by manipulating these three matrix equations to get

$$(C_x^T K_u^{-1} C_x + C_y^T K_v^{-1} C_y) p = -C_x^T K_u^{-1} f_u - C_y^T K_v^{-1} f_v. \tag{1.11}$$

The matrix on the left-hand side is known as the full consistent pressure matrix (FCPM). It is not practical to use because it is a dense matrix (therefore the name "full"), since K_u^{-1} and K_v^{-1} are dense in general. It is possible to construct simplified consistent pressure equations (SCPE's) by replacing the dense inverses with $\tilde{K}_u^{-1}$ and $\tilde{K}_v^{-1}$ which are sparse or easy to compute. (Often they are diagonal matrices created by lumping the original elements onto the diagonal.) It is also possible to construct non-consistent pressure equations (NCPE's), an alternative to directly manipulating the discretized momentum and continuity equations. For example, NCPE's can be constructed by discretizing the Poisson equation for pressure. The equations are called non-consistent because their solutions are only approximate with the solution of (1.7).

In the SIMPLE-type algorithms [120], an equation for pressure or pressure correction is derived, and beginning with an initial guess for the velocity or pressure field (respectively), an approximation to all the flow variables is found, i.e., the algorithms solve for each of u, v, and p individually in each step of a nonlinear iteration. For the *pressure correction* algorithm, the equation

$$(C_x^T \tilde{K}_u^{-1} C_x + C_y^T \tilde{K}_v^{-1} C_y) \Delta p = -C_x^T u^i - C_y^T v^i + b_c \tag{1.12}$$

is first solved for the pressure correction Δp. With the i superscript variables denoting current values, the variables are then corrected with

$$u = u^i + \tilde{K}_u^{-1} C_x \Delta p \tag{1.13}$$

$$v = v^i + \tilde{K}_v^{-1} C_y \Delta p \tag{1.14}$$

$$p = p^i + \Delta p \tag{1.15}$$

and the momentum equations

$$K_u u^{i+1} = f_u^* + C_x p \tag{1.16}$$

$$K_v v^{i+1} = f_v^* + C_y p \tag{1.17}$$

are solved, where "*" denotes a quantity computed with the latest available variables. The procedure is then iterated until convergence. All the K submatrices above are the ones current for that iteration. Note that to simplify the above explanation, we left out the relaxation factors that are normally used. The advantage of SIMPLE-type methods is that the system for pressure is symmetric positive definite, while the other systems are advection-diffusion type equations, and are nonsymmetric but also definite. Iterative methods are useful for these systems. For strongly-coupled problems, however, the number of SIMPLE-type steps required for convergence is often very high, and the fully-coupled approach with an iterative method may be a better alternative. It is interesting to note that the SIMPLE and Uzawa type of segregated algorithms were developed originally partially to accommodate small computer memories.

A difficulty related to the lack of an explicit equation for pressure is spurious zero or checkerboard solutions for velocity or pressure. This can be overcome in finite difference and finite volume methods by using staggered grids, or in the finite element method by using unequal order interpolation for the pressure and velocity variables. A finite element technique that circumvents the restriction of unequal-order elements is the streamline upwind Petrov-Galerkin stabilization [29]. A simple variant due to Brezzi and Pitkäranta [27] and described in Fortin [71] amounts to changing the continuity condition in the weak form of the problem. In the strong form, this corresponds to a change in the continuity equation

$$\nabla \cdot \boldsymbol{u} = -\epsilon \nabla^2 p$$

plus a zero Neumann boundary condition on p. The checkerboard modes are highly oscillatory and are removed by the smoothing effect of the Laplace operator [71]. The additional boundary condition, however, reduces the accuracy of p near the boundary, and often additional terms are added to the weak form to compensate for this error.

To get a symmetric positive definite matrix, alternative schemes such as the least-squares finite element method [35] can be used. In this method, the finite element bases can also be of equal order, since locking and spurious pressure modes do not occur. To derive the method, the differential equations are first reduced to a system of first order equations. In the case of the steady

2-D incompressible Navier-Stokes equations, three equations are transformed into four with the introduction of the scalar vorticity variable $\zeta = \partial v/\partial x - \partial u/\partial y$,

$$\begin{aligned}
\rho\left(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}\right) + \frac{\partial p}{\partial x} + \mu\frac{\partial \zeta}{\partial y} &= 0 \\
\rho\left(u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y}\right) + \frac{\partial p}{\partial y} - \mu\frac{\partial \zeta}{\partial x} &= 0 \\
\zeta - \frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} &= 0 \\
\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} &= 0.
\end{aligned}$$

Alternatively, in the stress formulation of the incompressible Navier-Stokes equations, the system can be transformed to six first order equations with the introduction of the normal and shear stress variables. This is suitable, for example, for modeling generalized Newtonian fluids. Then, defining r to be the residual of the above first order system, the minimization of

$$I = \int_\Omega r^T r \, dx \, dy$$

over the domain Ω with respect to the finite element approximations of the variables leads to a symmetric positive definite system to be solved. These systems, however, are also poorly conditioned, bearing some similarity to the normal equations.

Instead of the primitive variable formulation of the incompressible Navier-Stokes equations, the stream-function vorticity formulation (in 2-D) or the velocity-potential vorticity formulation (in 3-D) may be used. Neither of these formulations involve pressure, and definite linear systems are produced. These formulations have their own pros and cons—note in particular that six variables are involved in the 3-D case, rather than four in the primitive variable formulation.

We have discussed several factors that influence some of the general properties of linear systems. Many factors, such as boundary conditions, have not been mentioned, but some of the main problem formulations and their effects have been discussed. A preconditioner attempts to improve certain relevant, usually spectral, properties of the coefficient matrix. In many cases, a preconditioner is constructed by explicitly trying to control the properties that affect convergence, for example, the largest and smallest magnitude eigenvalues, and the condition number. In other cases, the preconditioner is simply some approximation to the coefficient matrix without trying to control these properties.

A very popular preconditioner is the incomplete factorization of the matrix A. It has been used successfully in many problems, but in other problems, difficulties may arise. For a numerical example of the difficulties that may be encountered, consider the modeling of the axisymmetric flow past a circular cylinder at Reynolds number 40. (The Reynolds number Re is a dimensionless parameter that characterizes the flow, and is proportional to a velocity scale, a length scale, and ρ/μ.)

The FIDAP fluid dynamics package [67, 68] was used to generate the fully-coupled matrix arising from the first step of a Picard iteration. The Galerkin Finite Element Method was used to discretize the problem, using quadrilateral elements with biquadratic basis functions for velocities, and linear discontinuous basis functions for pressure. On an 11 by 11 grid, this leads to a system of size 2532 with 50380 nonzero elements.

Consider an incomplete factorization calculated for this coefficient matrix by a threshold-based ILU preconditioner, ILUT [127], such that $A \approx LU$. The preconditioning operation is $(LU)^{-1}$. Figure 1.2 shows the lower bound of $\|(LU)^{-1}\|_\infty$ computed by $\|(LU)^{-1}e\|_\infty$, where e is the vector of all ones, for different amounts of allowable fill-in (*lfil*) in the factors. The graph show that this bound can be as high as 10^{250}, strongly suggesting that the factors have no use. The bound means that preconditioning a vector of norm 1 can result in a vector of norm 10^{250}. Interestingly, this is often *not* due to very large values in the factors themselves.

This type of instability often occurs for for indefinite matrices or matrices with large non-symmetric parts. The long recurrences associated with solving with factors such as these are unstable [32, 65], producing solutions with extremely large components. As can be seen in the graph, this instability tends to decrease as the accuracy of the factorization is improved. The norm also tends to decrease when the factors are sparser since fewer elements can contribute to the growth of the recurrence values. This example is particularly striking because it can be solved by an iterative method *without preconditioning*.

There are a number of preconditioners that can be designed to avoid this problem, and this is one of the topics of this thesis. One technique, described in Chapter 3, is to avoid triangular solves altogether and approximate the inverse of a matrix directly. Another technique, described in Chapter 5, is to try to stabilize ILU factorizations.

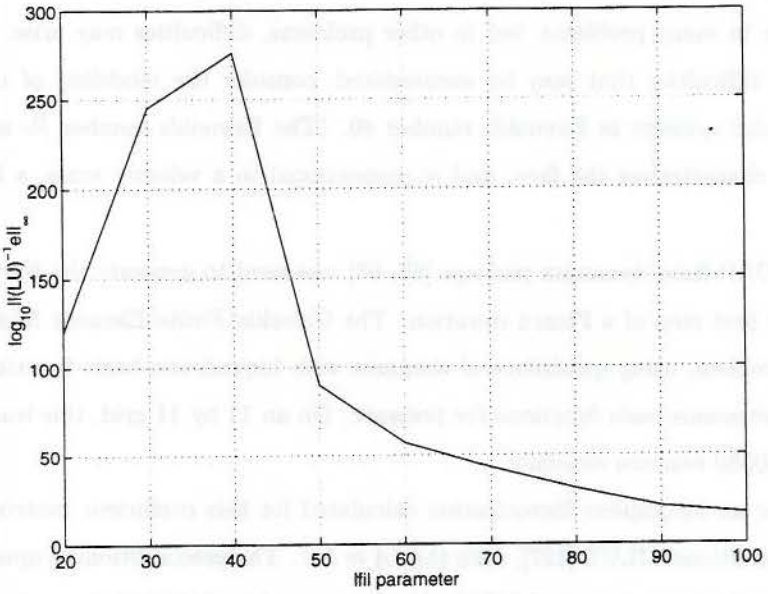


Figure 1.2: Instability of the ILUT(*lfil*) factors for FIDAPM03.

1.2 Test problems

The test problems used in this thesis were selected from sparse matrix collections for testing and comparing linear solvers and other numerical linear algebra algorithms. The first of these collections, the Harwell-Boeing collection [59], also set a file format called the Harwell-Boeing format. Both the collection and the format have become de facto standards in research and some industrial communities. Two other well-known collections are those associated with the SPARSKIT[2] and UMFPACK[3] packages. All the collections cover a wide range of application areas. Recently a single repository for the Harwell-Boeing and some other collections has been established by the National Institute of Standards and Technology. The repository, called "MatrixMarket,"[4] provides search functions and makes the matrices available under a common hypertext interface.

The SPARSKIT collection contains two subcollections that are problems derived from solving the incompressible Navier-Stokes equations: driven cavity problems and FIDAP example problems. For the FIDAP problems, like the example described in Section 1.1, the first linear systems

[2]http://www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html
[3]http://www1.cise.ufl.edu/~davis/
[4]http://math.nist.gov/MatrixMarket/

in the nonlinear iterations were extracted. We chose segregated or fully-coupled solution techniques to generate different types of matrices. All the FIDAP matrices have a symmetric pattern.

The driven cavity problems were discretized by the Galerkin Finite Element Method using rectangular elements, with biquadratic basis functions for velocities, and linear discontinuous basis functions for pressure. The solution at higher Reynolds numbers were obtained by solving a sequence of problems with Reynolds number ramped in increments of 100. The first Jacobian system of the Newton iterations at each Reynolds number was extracted. The matrices have symmetric structure. Only the matrix for Reynolds number 0 is symmetric. Table 1.1 gives the sizes and numbers of nonzero elements for these problems for two different mesh sizes.

Elements	n	nnz
20 by 20	4562	138,187
40 by 40	17922	567,467

Table 1.1: Driven cavity problems.

The degrees of freedom in the coefficient matrices were numbered element by element. We use other orderings, to be described later, in different tests. Scaling of the matrix is also an issue. Rows in the Jacobian matrix express either continuity or the conservation of momentum. The momentum equations contain the product of the velocity gradient and the Reynolds number. At high Reynolds numbers, the velocity gradient will be large, and the magnitude of the momentum equations is much larger than that of the continuity equations. This difference in scales causes poor behavior in threshold incomplete factorizations. Thus, we scaled the linear systems so that each row has unit 2-norm, and then scaled again so that each column has unit 2-norm.

1.3 Summary of thesis

This thesis proposes several new preconditioning techniques. Chapter 2 first gives some background on accelerators, how accelerators accommodate preconditioning, and discusses ILU and approximate inverse preconditioners in particular. A recent development covered in this chapter is the treatment of nearly symmetric systems with a symmetric positive definite (SPD) preconditioner. In this case, the preconditioned matrix is nearly symmetric, and truncated versions of iterative methods can be nearly as effective as untruncated versions, such as the full generalized minimum residual (GMRES) method. Algorithms such as Biconjugate Gradient (Bi-CG) are also less prone to break down.

With Chapter 3 begins a discussion of one of the main topics of this thesis. The chapter opens with a review of sparse approximate inverses and then proposes a new technique to compute sparse approximate inverse preconditioners based on approximate minimization with an iterative method. To make this method economical, implementations must take advantage of sparse-sparse operations, such as the product of a sparse matrix with a sparse vector, or the dot product of two sparse vectors. We introduce the idea of self-preconditioning, whereby the linear subproblems to be solved can be preconditioned by the current approximation to the inverse. Initial guesses for the approximations also influence the quality of the preconditioner. This new technique produces approximate inverses comparable in quality with others in the literature, but at a lower computational cost, and with a simpler method to determine good sparsity patterns.

Chapter 4 describes applications of the approximate inverse. The theme is to demonstrate more effective uses of approximate inverses than using them to precondition the entire matrix. The chapter begins by discussing how to improve an existing preconditioner with an approximate inverse, how to compute approximate pseudo-inverses, as well as how to construct factorized preconditioners with the approximate inverse technique. A promising use of approximate inverses is in the construction of block preconditioners, particularly for the discrete fully-coupled incompressible Navier-Stokes equations and matrices that are partitioned by nonoverlapping domain decomposition. Approximate inverse techniques are used to generate sparse approximate solutions whenever these are needed in forming the preconditioner, e.g., sparse approximations to the Schur complement or its inverse. The storage requirements for these preconditioners are much less than for ILU preconditioners for many tough, large-scale problems.

In Chapter 5 we experimentally analyze the causes of failure of ILU preconditioners, particularly on indefinite linear systems. The failure modes can be categorized as:

1. inaccuracy due to very small pivots, leading to unstable and inaccurate factorizations, and indicated by large off-diagonal elements in L and U;
2. unstable triangular solves, indicated by very large values of $\|L^{-1}\|$ and $\|U^{-1}\|$ while the off-diagonal entries of L and U may remain small;
3. inaccuracy due to dropping, i.e., the norm of $E = A - LU$ is large;
4. zero pivots, which may be induced structurally, or numerically, when a factorization is unstable.

By directly estimating the condition number of the factorization, measuring the size of the smallest pivots, and measuring the pivot growth (instability and inaccuracy of the factorization) for several

variants of ILU preconditioners, we found that the best strategy when using ILU factorizations is different for structured and very unstructured matrices. If zero pivots are common, partial pivoting is useful for unstructured matrices; partial pivoting will destroy the symmetric structure of structured matrices, and some form of preordering should be used instead. If $\|(LU)^{-1}\|$ is very large, pivoting again should be used for unstructured matrices; for structured matrices, some form of stabilization by augmented diagonal elements should be used. In our experiments, incomplete factorizations were always stable unless there were very small pivots, which differs from the case of complete factorizations.

Finally, we present an object-oriented framework that implements some of the preconditioning algorithms above. General software for preconditioning the iterative solution of linear systems is greatly lagging behind the literature. This is partly because specific problems need specific matrix and preconditioner data structures in order to be solved efficiently; i.e., multiple implementations of a preconditioner with specialized data structures are required. Chapter 6 presents a framework to support preconditioning with various, possibly user-defined, data structures for matrices that are partitioned into blocks. The main idea is to define data structures for the blocks, and an upper layer of software which uses these blocks transparently of their data structure. This transparency can be accomplished by using an object-oriented language. Thus various preconditioners, such as block relaxations and block incomplete factorizations, only need to be defined once, and will work with any block type. In addition, it is possible to transparently interchange various approximate or exact techniques for inverting pivot blocks, or solving systems whose coefficient matrices are diagonal blocks. This leads to a rich variety of preconditioners that can be selected. Operations with the blocks are performed with optimized libraries or fundamental data types. Comparisons with an optimized Fortran 77 code on both workstations and Cray supercomputers show that this framework can approach the efficiency of Fortran, as long as suitable block sizes and block types are chosen.

Chapter 2

Background

2.1 Acceleration

The original iterative methods for solving $Ax = b$ were analyzed in terms of splittings of the matrix A. Given a matrix M, the splitting $A = M - N$ defines an iterative method as

$$\begin{aligned} Mx_{k+1} &= Nx_k + b \\ x_{k+1} &= M^{-1}Nx_k + M^{-1}b \\ &= (I - M^{-1}A)x_k + M^{-1}b \end{aligned} \tag{2.1}$$

where x_0 is an initial guess. It should be inexpensive to solve a set of equations with M as the coefficient matrix. In the Jacobi method, M is the diagonal of A, and in the Gauss-Seidel method, M is the lower triangular part of A. The method (2.1) converges for any x_0 if and only if the spectral radius of the iteration matrix is less than 1, i.e.,

$$\rho(I - M^{-1}A) < 1.$$

If A is an M-matrix, then we have the additional result that the method (2.1) converges if and only if M, N is a *regular* splitting, i.e., M is nonsingular and M^{-1} and N are nonnegative (have no negative elements).

If the iterative method (2.1) is symmetrizable, that is, if the iteration matrix $I - M^{-1}A$ can be transformed by a matrix W into a new iteration matrix $W(I - M^{-1}A)W^{-1}$ that is SPD, then polynomial acceleration methods such as the Chebyshev or conjugate gradient (CG) methods are faster. In modern terms, these schemes are viewed as iterative *accelerators*, such as Chebyshev or CG, applied with a preconditioner M. Accelerators and preconditioners are the two ingredients of modern iterative solution techniques. This section will discuss acceleration. The next sections will discuss how accelerators accommodate preconditioning, and details about particular preconditioners.

A large and popular class of accelerators is the class of Krylov subspace methods. These

methods can be individually derived from optimization principles, but can also be elegantly placed together under the common framework of projection methods. Projection methods for solving $Ax = b$ determine an approximate solution x_m from the m-dimensional subspace

$$x_0 + \mathcal{K}_m \tag{2.2}$$

where x_0 is an initial guess to the solution, by imposing the Petrov-Galerkin condition

$$b - Ax_m \perp \mathcal{L}_m \tag{2.3}$$

where $\mathcal{L}_m$ is another subspace of dimension m. When $\mathcal{K}_m = \mathcal{K}_m(A, r_0)$ is the Krylov subspace

$$\mathcal{K}_m(A, r_0) = \text{span}\{r_0, Ar_0, A^2r_0, \ldots, A^{m-1}r_0\}$$

where $r_0 = b - Ax_0$, then the method is called a Krylov subspace method. In the following, $\mathcal{K}_m$ denotes $\mathcal{K}_m(A, r_0)$ unless otherwise indicated.

The main classes of Krylov subspace accelerators fall into three categories, depending on how the subspace $\mathcal{L}_m$ is chosen:

- $\mathcal{L}_m = \mathcal{K}_m$. When A is SPD, this choice of $\mathcal{L}_m$ minimizes the A-norm of the error. Examples are the conjugate gradient method in the symmetric case, and the full orthogonalization method (FOM) in the nonsymmetric case.

- $\mathcal{L}_m = A\mathcal{K}_m$. This choice of $\mathcal{L}_m$ minimizes the residual norm $\|b - Ax_m\|_2$. Examples are GMRES and its equivalents, including the conjugate residual (CR) method in the symmetric case.

- $\mathcal{L}_m = \mathcal{K}_m(A^T, r_0^*)$, where r_0^* is a vector not parallel to r_0. This choice of $\mathcal{L}_m$ was designed for nonsymmetric A and provides short-term recurrence relations for the Krylov subspace bases for this case. Thus these methods may require less storage than GMRES and FOM, but are also more prone to break down. Examples are Bi-CG, QMR, CGS, Bi-CGSTAB, and TFQMR.

To derive the GMRES method from the point of view of projection methods, let $V_m = \{v_1, v_2, \ldots, v_m\}$ be a basis for $\mathcal{K}_m$ and let $W_m = AV_m$ be a basis for $\mathcal{L}_m$. GMRES uses the Arnoldi method to build an orthogonal basis V_m for $\mathcal{K}_m$ for an initial vector $v_1 = r_0/\beta$ of 2-norm 1 as follows.

ALGORITHM **2.1.1** *Arnoldi (Modified Gram-Schmidt version)*

1. *Choose a vector* v_1 *of norm 1*
2. *For* $j = 1, 2, \ldots, m$ *do*
3. $\quad w = Av_j$
4. $\quad$ *For* $i = 1, \ldots, j$ *do*
5. $\quad\quad h_{ij} = (w, v_i)$
6. $\quad\quad w = w - h_{ij}v_i$
7. $\quad$ *EndDo*
8. $\quad h_{j+1,j} = \|w\|_2$. *If* $h_{j+1,j} = 0$ *then Stop.*
9. $\quad v_{j+1} = w/h_{j+1,j}$
10. *EndDo*

The Arnoldi method generates a set of vectors which satisfy

$$AV_m = V_{m+1}\bar{H}_m \tag{2.4}$$

where $\bar{H}_m$ is an upper Hessenberg matrix defined by the algorithm, assuming that the Arnoldi process does not break down. The approximate solution x_m in the space (2.2) has the form

$$x_m = x_0 + V_m y_m \tag{2.5}$$

for some vector y_m. The orthogonality condition (2.3) gives

$$W_m^T A V_m y_m = W_m^T r_0 \tag{2.6}$$

$$V_m^T A^T A V_m y_m = V_m^T A^T r_0. \tag{2.7}$$

Substituting the relation (2.4) leads to

$$\bar{H}_m^T V_{m+1}^T V_{m+1} \bar{H}_m y_m = \bar{H}_m^T V_{m+1}^T r_0 \tag{2.8}$$

$$\bar{H}_m^T \bar{H}_m = \bar{H}_m^T (\beta e_1) \tag{2.9}$$

$$y_m = \arg\min_y \|\beta e_1 - \bar{H}_m y\|_2 \tag{2.10}$$

where βe_1 is the 2-norm of r_0 times the first unit coordinate vector. The solution of the normal equations give the least-squares solution because the upper Hessenberg matrix $\bar{H}_m$ is full rank. The GMRES algorithm performs the minimization (2.10) and substitutes y_m into (2.5) to get the approximate solution x_m. In practice, m is fixed to control storage, and the algorithm is *restarted* if the approximate solution is not satisfactory after m iterations are taken, as shown here:

ALGORITHM **2.1.2** *Restarted GMRES(m)*

1. *Compute* $r_0 = b - Ax_0$; $\beta = \|r_0\|_2$; $v_1 = r_0/\beta$
2. *Apply* m *steps of the Arnoldi algorithm above*
3. *Compute the minimizer* y_m *of* $\|\beta e_1 - \bar{H}_m y\|_2$
4. *Compute the approximate solution* $x_m = x_0 + V_m y_m$
5. *If satisfied Stop; else set* $x_0 = x_m$ *and goto 1.*

It is possible to cheaply estimate the 2-norm of the residual $r_i = b - Ax_i$ of the approximate solution after each Arnoldi iteration. Thus the Arnoldi process can be exited early, before all m steps are taken, if a residual norm tolerance has been satisfied.

Finally, it is important to note that GMRES and other iterative algorithms only access the matrix A through a matrix-vector product operation.

2.2 Preconditioned iterations

Preconditioning transforms the system $Ax = b$ into an equivalent system, one that has the same solution, but better convergence properties. Given a preconditioner M which is an approximation to A, this transformation can be accomplished by *left*-preconditioning,

$$M^{-1}Ax = M^{-1}b, \tag{2.11}$$

right-preconditioning,

$$AM^{-1}u = b, \quad x = M^{-1}u \tag{2.12}$$

or if $M = LU$, by *split*-preconditioning,

$$L^{-1}AU^{-1}u = L^{-1}b, \quad x = U^{-1}u. \tag{2.13}$$

To make an iterative algorithm such as Algorithm 2.1.2 accommodate preconditioning, it only needs to be written for one of the above preconditioned equations. For right-preconditioning, only line 3 of the Arnoldi algorithm and line 4 of the GMRES(m) algorithm need to be changed. With the inclusion of M^{-1} operations, they should now read:

3. *(Arnoldi)* $w = AM^{-1}v_j$
4. *(GMRES(m)) Compute the approximate solution* $x_m = x_0 + M^{-1}V_m y_m$

As seen in these lines, M does not need to be explicitly available as a matrix. The *preconditioning operator* M^{-1} is a sequence of operations that somehow approximates the effect of A^{-1} on a vector. In the case of split-preconditioning with a non-SPD matrix $M = LU$, only the operations of L^{-1} and U^{-1} on a vector are required. These operations are generally applied once every time the iterative method performs a matrix-vector product with A, and thus they should be relatively inexpensive operations to perform.

All three preconditioned matrices $M^{-1}A$, AM^{-1}, and $L^{-1}AU^{-1}$ have the same eigenvalues, and in practice, when A is nonsymmetric, iterative methods employing any one of these forms of preconditioning behave similarly. When only the preconditioned residual is available within an iterative algorithm (as in the GMRES algorithm), there is a slight advantage in using right-preconditioning, since the right-preconditioned residual is equal to the unpreconditioned residual. When a nearly symmetric matrix A is preconditioned by a symmetric preconditioner $M = LL^T$, then there is a slight advantage in using split-preconditioning. This preserves the qualitative properties of near-symmetry and near-normality, as will be discussed in the next section. When M is poorly conditioned, the three forms of preconditioning may behave very differently, but in this case, attention should be paid to improving the preconditioner M.

When A and M are symmetric positive definite, split-preconditioning is normally used, otherwise the preconditioned matrix will not be SPD. Also when A and M are SPD, split-preconditioning can be implemented such that factoring M is not necessary. This can be done by using a different inner product. Specifically, $M^{-1}A$ is self-adjoint for the M-inner product,

$$(x, y)_M = (Mx, y) = (x, My)$$

since we have

$$(M^{-1}Ax, y)_M = (Ax, y) = (x, Ay) = (x, M(M^{-1}A)y) = (x, M^{-1}Ay)_M.$$

Similarly, AM^{-1} is self-adjoint for the M^{-1}-inner product.

Note that even when M-inner products are used, we will not need to multiply by M which may not be available explicitly; we only need to solve linear systems with the matrix coefficient M. It can also be seen that with a change of variables, the iterates produced by a split preconditioned algorithm are the same as those produced by left-preconditioning using the M-inner product, and right-preconditioning using the M^{-1}-inner product. All three are thus equivalent.

2.3 Preconditioning nearly symmetric systems

When A is only nearly symmetric and M is SPD, there may also be an advantage in using split preconditioning with $M = LL^T$, or equivalently, M-inner products with left-preconditioning, or M^{-1}-inner products with right-preconditioning in an iterative algorithm [37]. This is because by using regular left- or right-preconditioning, the systems (2.11) and (2.12) respectively are no longer symmetric. The use of split preconditioning or alternative inner products preserve the initial degree of symmetry in A. We refer to these latter preconditionings as "symmetric."

Ashby, Manteuffel, and Saylor [4] have fully considered the case of using alternate inner products when the matrix A is symmetric. Work of which we are aware that consider the use of alternate inner products when A is near-symmetric are Young and Jea [151] and Meyer [108]. In the latter, the $A^T M^{-1} A$ inner product (with M SPD) is used with ORTHOMIN and ORTHODIR.

We now discuss right symmetric preconditioning, i.e., right-preconditioning while using the M^{-1}-inner product. In the Arnoldi algorithm, note that $M^{-1}z$ is available when z needs to be normalized in the M^{-1} norm. However, $M^{-1}z$ is normally computed at the next iteration in the standard Arnoldi algorithm; a slight reorganization of the Arnoldi algorithm yields the following.

ALGORITHM **2.3.1** *Arnoldi with M^{-1}-inner products*

1. *Choose a vector v_1 of M^{-1}-norm 1, compute $w_1 = M^{-1}v_1$*
2. *For $j = 1, 2, \ldots, m$ do*
3. $\quad z = Aw_j$
4. $\quad$ *For $i = 1, \ldots, j$ do*
5. $\quad\quad h_{ij} = (z, w_i)$
6. $\quad\quad z = z - h_{ij}v_i$
7. $\quad$ *EndDo*
8. $\quad w = M^{-1}z$
9. $\quad h_{j+1,j} = (z, w)^{1/2}$. *If $h_{j+1,j} = 0$ then Stop.*
10. $\quad w_{j+1} = w/h_{j+1,j}$
11. $\quad v_{j+1} = z/h_{j+1,j}$
12. *EndDo*

Note that the preconditioned vector is computed in line 8, while in the standard algorithm it is computed before line 3. Both the v's and the w's need to be saved, where $w_j = M^{-1}v_j$ in this case.

The additional storage of the w's, however, makes this algorithm naturally "flexible," i.e., it accommodates the situation where M varies at each step as when $M^{-1}v$ is the result of some unspecified computation. If M^{-1} is not a constant operator, then a basis for the right-preconditioned Krylov subspace cannot be constructed from the v's alone. However, the vectors $w_j = M_j^{-1}v_j$ do

form a basis for this subspace, where M_j^{-1} denotes the preconditioning operation at the j-th step. The use of this extra set of vectors is exactly how the standard flexible variant of GMRES is implemented [125]. Such flexibility can allow preconditioners to change when an iterative method is stagnating, and thus provides additional control for robustness. This type of flexibility is also used to help implement "deflated" and "augmented" Krylov subspace techniques (see [40] and the references therein).

We can now write preconditioned GMRES algorithms using these implementations of alternative inner products in the Arnoldi process. In the left-preconditioned case, we can minimize the M-norm of the preconditioned residual vector $M^{-1}(b-Ax)$ by simply minimizing the 2-norm of the projected system in the standard GMRES algorithm. To be formal, we state the following theorem without proof [37].

Theorem 2.3.1 *The approximate solution x_m obtained from the left-preconditioned GMRES algorithm with M-inner products minimizes the residual M-norm $\|M^{-1}(b-Ax)\|_M$ over all vectors of the affine subspace $x_0 + K_m$ in which*

$$K_m = \text{span}\{z_0, M^{-1}Az_0, \ldots, (M^{-1}A)^{m-1}z_0\} \tag{2.14}$$

where $z_0 = M^{-1}r_0$. In addition, the approximate solution x_m obtained from the right-preconditioned GMRES algorithm with M^{-1}-inner products minimizes the residual M^{-1}-norm $\|b-Ax\|_{M^{-1}}$ over the same affine subspace.

We can show that both left and right symmetric preconditioning are mathematically equivalent to split preconditioning. All that must be noticed is that minimizations of

$$\|L^{-1}(b-Ax)\|_2 = \|b-Ax\|_{M^{-1}} = \|M^{-1}(b-Ax)\|_M$$

are the same, and that the minimizations are over the same subspace in each of the left, right, and split preconditioning options [135, Sec. 9.3.4]. We emphasize in particular that it is the split preconditioned residual that is minimized in all three algorithms.

2.3.1 Truncated iterative methods

Truncated iterative methods are an alternative to restarting, when the number of steps required for convergence is large and the computation and storage of the Krylov basis becomes excessive. When

A is exactly symmetric, a three-term recurrence governs the vectors in the Arnoldi process, and it is only necessary to orthogonalize the current Arnoldi vector against the previous two vectors. If A is nearly symmetric, an incomplete orthogonalization against a small number of previous vectors may be advantageous over restarted methods. The advantage here may offset the cost of maintaining the extra set of w vectors in Algorithm 2.3.1 to maintain the initial degree of symmetry. The incomplete Arnoldi procedure stores only the previous k Arnoldi vectors, and orthogonalizes the new vectors against them. It differs from the full Arnoldi procedure described in Algorithm 2.1.1 only in line 4, which should now read:

4. *For* $i = \max\{1, j-k+1\}, \ldots, j$ *do*

which would normally be a loop from 1 to j. It can be considered to be the full Arnoldi procedure when k is set to infinity.

The truncated version of GMRES uses this incomplete Arnoldi procedure and is called Quasi-GMRES [30]. The practical implementation of this algorithm allows the solution to be updated at each iteration, and is thus called a "direct" version, or DQGMRES [132].

We hypothesize that we may use DQGMRES with small k when A (or its preconditioned version) is nearly symmetric or nearly skew symmetric. To test this idea of using symmetric preconditionings with truncated iterative methods for nearly symmetric systems, we selected the lid-driven cavity problem described in Section 1.1. In particular, we considered the expression of the conservation of momentum,

$$Re(\boldsymbol{u} \cdot \nabla \boldsymbol{u}) = -\nabla p + \nabla^2 \boldsymbol{u}$$

where $\boldsymbol{u}$ denotes the vector of velocity variables, p denotes the pressure, and Re is the Reynolds number. The degree of symmetry in the matrices is parameterized by the Reynolds number. The matrices are the initial Jacobians at each Newton iteration, assuming a zero pressure distribution. For convenience, however, we chose the right-hand sides of the linear systems to be the vector of all ones. A mesh of 20 by 20 elements was used, leading to momentum equation matrices of order 3042 and having 91204 nonzero entries. The nodes corresponding to the boundaries were not assembled into the matrix, and the degrees of freedom were numbered element by element. For Reynolds number 0, the matrix is SPD, and is equal to the symmetric part of the matrices with nonzero Reynolds number.

We generated matrices with Reynolds number less than 10, which gives rise to the nearly symmetric case. For Reynolds number 1, the degree of symmetry measured by

$$\frac{\|A - A^T\|_F}{\|A + A^T\|_F}$$

has value 7.5102×10^{-4} and this measure increases linearly with the Reynolds number (at least up to $Re = 10$).

In the numerical experiments below, we show the number of matrix-vector products consumed by GMRES(k) and DQGMRES(k) to reduce the *actual* residual norm to less than 10^{-6} of the original residual norm, with a zero initial guess. Several values of k are used. A dagger (†) in the tables indicates that there was no convergence within 500 matrix-vector products. The incomplete Cholesky factorization IC(0) of the $Re = 0$ problem was used as the preconditioner in all the problems.

For comparison, we first show in Table 2.1 the results using the standard form of right-preconditioning. Table 2.2 shows the results using right-preconditioning with M^{-1} inner products, or equivalently, split preconditioning. In the latter case, care was taken to assure that GMRES did not stop before the *actual* residual norm was within twice the tolerance. For DQGMRES, since an accurate residual norm estimate is not available within the algorithm, the exact residual norm was computed and used for the stopping criterion for the purpose of this comparison. The right-preconditioned methods have a slight advantage in this comparison (by as many as 20 matrix-vector products), since they directly minimize the actual residual norm, whereas the symmetrically preconditioned methods minimize a preconditioned residual norm.

	GMRES(k)			DQGMRES(k)								
Re.	5	10	∞	2	3	4	5	6	7	8	9	10
0	232	129	59	76	220	105	72	92	†	160	83	75
1	218	126	69	76	276	131	81	94	†	97	90	79
2	233	126	70	78	391	258	82	95	†	98	94	78
3	208	126	71	87	346	232	85	95	†	99	97	79
4	214	128	72	94	345	†	88	95	477	108	98	86
5	210	128	72	192	394	†	91	95	326	128	99	90
6	214	128	73	446	361	†	94	97	258	197	100	94
7	215	129	73	†	345	†	97	99	239	229	101	96

Table 2.1: Mat-Vec's for convergence for right-preconditioned methods.

	GMRES(k)			DQGMRES(k)								
Re.	5	10	∞	2	3	4	5	6	7	8	9	10
0	243	119	57	58	58	58	58	58	58	58	58	58
1	243	119	67	75	74	74	75	74	74	74	75	75
2	244	120	68	78	78	78	79	78	78	78	78	78
3	244	121	69	88	87	87	87	87	86	86	87	87
4	244	122	70	108	95	95	95	93	91	91	93	95
5	244	126	70	†	105	103	105	100	97	96	101	104
6	244	127	71	†	118	111	119	108	101	101	110	117
7	243	128	71	†	131	121	139	117	104	105	121	139

Table 2.2: Mat-Vec's for convergence for symmetric right-preconditioned methods.

The results in Table 2.1 show the irregular performance of DQGMRES(k) for these small values of k when the preconditioned system is not symmetric. The performance is entirely regular in Table 2.2, where the preconditioned system is near symmetric. For Reynolds numbers up to 3, the systems are sufficiently symmetric so that DQGMRES(2) behaves in the same way as DQGMRES with much larger k. The performance remains regular until beyond Reynolds number 7, when the number of steps to convergence begins to become irregular, like in the right-preconditioned case.

GMRES(k) with either right or symmetric preconditioning does not show any marked difference in performance; apparently the symmetry of the preconditioned system is not as essential here for this problem. However, the results do show that DQGMRES(k) with small values of k may perform as well, in terms of number of steps, as GMRES(k) with large values of k, particularly if near-symmetry is preserved. Since the former is much more efficient, the combination of preserving symmetry and truncated iterative methods may result in a much more economical method, as well as the more regular behavior shown above.

We also performed the same experiments with orthogonal projection methods, in particular, the full orthogonalization method (FOM) and its truncated variant, the direct incomplete orthogonalization method (DIOM) [135]. The results were very similar to the results above, and are not shown here. Indeed, the development of the algorithms above is identical for these methods.

For interest, we also performed tests where an ILU(0) preconditioner was constructed for each matrix and compared right and split preconditioning. For the near-symmetric systems here, there was very little difference in these results compared to using IC(0) constructed from the $Re = 0$ case for all the matrices. Thus the deterioration in performance as the Reynolds number increases is not entirely due to a relatively less accurate preconditioner, but is more due to the increased nonsym-

metry and non-normality of the matrices. Although the eigenvalues of the preconditioned matrices are identical, their eigenvectors and hence their degree of non-normality may change completely. Unfortunately, it is difficult to quantitatively relate non-normality and convergence.

2.3.2 Symmetric preconditioning in Bi-CG

Both left-symmetric and right-symmetric preconditioning of the Bi-CG algorithm are relatively straightforward, and no extra vectors are required. Like GMRES, both left and right symmetric preconditioned versions of Bi-CG are equivalent to the split preconditioned version, and this can be shown by a change of variables. However, in both left and right symmetric preconditioned versions, the exact, rather than the split preconditioned residual is available.

The unpreconditioned Bi-CG algorithm cannot break down before finding the exact solution if A is SPD and r_0^* is chosen to be r_0. This is because $r_j^* = r_j$ and $p_j^* = p_j$ for all j and the vectors Ap_j, p_j^* never become orthogonal (the p and p^* vectors are the search directions in the Bi-CG algorithm). In fact, the cosine

$$\frac{(Ap_j, p_j^*)}{\|Ap_j\|\|p_j^*\|} = \frac{(Ap_j, p_j^*)}{\|p_j\|\|p_j^*\|}\frac{\|p_j\|}{\|Ap_j\|}$$

can be bounded below by the reciprocal of the condition number of A.

Similarly, in the symmetric right-preconditioned version of Bi-CG, if both A and M are SPD, and $r_0^* = r_0$, then $r_j^* = r_j$ and $p_j^* = p_j$ for all j, and

$$\begin{aligned}\frac{(Ap_j, p_j^*)}{\|Ap_j\|\|p_j^*\|} &\geq \text{cond}^{-1}(A)\\ \frac{(M^{-1}r_j, r_j^*)}{\|M^{-1}r_j\|\|r_j^*\|} &\geq \text{cond}^{-1}(M).\end{aligned}$$

We measure the cosines rather than the quantities (Ap_j, p_j^*) and $(M^{-1}r_j, r_j^*)$ because the p and r vectors have magnitudes going to 0 as the algorithms progress. Recall that in the case when $(M^{-1}r_j, r_j^*) = 0$ and $r_j = 0$, we have a *lucky breakdown.*

For the case of regular right- or left-preconditioning, or if $r_0^* \neq r_0$ in the symmetrically preconditioned cases, then no such lower bounds as the above exist, and the algorithms are liable to break down.

When A is near-symmetric, it is our hypothesis that the probability of breakdown is lower in the symmetrically preconditioned cases. To test the breakdown behavior of Bi-CG, Matlab was

used to generate random matrices of order 300 with approximately 50 percent normally distributed nonzero entries. The matrices were adjusted so that

$$A \leftarrow \frac{A + A^T}{2} - (\sigma_{min} + 10^{-5})I + \varepsilon \frac{A - A^T}{2},$$

i.e., the symmetric part was shifted so that the lowest eigenvalue was 10^{-5} and then ε times the skew-symmetric part was added back. The parameter ε was altered to get varying degrees of nonsymmetry.

For each ε that we tested, 100 matrices were generated, and the smallest value of the cosines corresponding to the denominators in the algorithms were recorded. In the right-preconditioned case, we recorded the minimum of

$$\frac{(AM^{-1}p_j, p_j^*)}{\|AM^{-1}p_j\|\|p_j^*\|} \quad \text{and} \quad \frac{(r_j, r_j^*)}{\|r_j\|\|r_j^*\|}$$

for all j, and for the symmetric right-preconditioned case, we recorded the minimum of

$$\frac{(Ap_j, p_j^*)}{\|Ap_j\|\|p_j^*\|} \quad \text{and} \quad \frac{(M^{-1}r_j, r_j^*)}{\|M^{-1}r_j\|\|r_j^*\|}$$

for all j. The relative residual norm reduction was 10^{-9} when the iterations were stopped. The initial guesses were zero, and r_0^* was set to r_0. IC(0) of the symmetric part was used as the preconditioner.

Table 2.3 shows the frequencies of the sizes of minimum cosines for the right-preconditioned (first row of each pair of rows) and the symmetrically-preconditioned cases (second row of each pair of rows). For example, all 100 minimum cosines were between 10^{-3} and 3×10^{-3} in the symmetrically-preconditioned case. The average number of Bi-CG steps and the average minimum cosine is also shown. The last column, labeled "better," shows the number of times that the minimum cosine was higher in the improved algorithm.

The table shows that the right-preconditioned algorithm can produce much smaller cosines, indicating a greater probability for breakdown. The difference between the algorithms is less as the degree of nonsymmetry is increased. For $\varepsilon = 0.1$, there is almost no difference in the breakdown behavior of the algorithms. The table shows that the number of Bi-CG steps is not significantly reduced in the new algorithm, nor is the *average* minimum cosine of the modified algorithm significantly increased. It is the probability that a small cosine is not encountered that is better.

Note that this behavior only applies when r_0^* is set to r_0. When r_0^* is chosen randomly, there is no gain in the symmetrically-preconditioned algorithm, as shown in Table 2.4.

ε $\left(\frac{\|A-A^T\|_F}{\|A+A^T\|_F}\right)$	steps	3e-6 1e-5	1e-5 3e-5	3e-5 1e-4	1e-4 3e-4	3e-4 1e-3	1e-3 3e-3	3e-3 1e-2	1e-2 3e-2	3e-2 1e-1	average $\times 10^{-3}$	better
0.000	32.51	1	4	9	19	26	30	11			1.35	
(0.)	31.77						100				1.87	74
0.005	30.57		1	2	8	16	34	34	5		3.51	
(2.3e-3)	29.97					2	1	82	15		8.54	92
0.010	29.27	1	0	3	2	10	29	32	20	3	7.25	
(4.5e-3)	28.94				1	2	1	6	88	2	15.79	77
0.050	27.53		1	3	8	13	36	31	8		4.15	
(2.3e-2)	27.32			2	3	7	18	39	26	5	9.47	69
0.100	26.38		1	11	18	39	27	4			0.88	
(4.5e-2)	26.42		3	4	15	40	27	11			1.26	57

Table 2.3: Frequencies of minimum cosines for right-preconditioned (first row of each pair of rows) and symmetrically-preconditioned (second row of each pair of rows) Bi-CG.

ε	steps	3e-6 1e-5	1e-5 3e-5	3e-5 1e-4	1e-4 3e-4	3e-4 1e-3	1e-3 3e-3	3e-3 1e-2	1e-2 3e-2	3e-2 1e-1	average $\times 10^{-3}$	better
0.000	33.05	1	4	11	24	26	30	4			0.92	
	32.54	1	1	3	11	24	56	4			1.31	63

Table 2.4: Frequencies of minimum cosines when r_0^* is chosen randomly.

Table 2.5 shows the number of steps and the minimum cosines for the two algorithms applied to the driven cavity problem described above. Figure 2.1 shows a plot of the minimum cosines as the two algorithms progress for the $Re = 1$ problem. Note that the minimum cosines are higher and much smoother in the symmetrically-preconditioned case. In the $Re = 7$ problem (not shown), the cosines are still higher, but the smoothness is lost.

	Bi-CG steps		min cosines	
Re.	right	symm	right	symm
0	70	62	1.52e-4	1.45e-1
1	71	68	1.08e-4	6.73e-3
2	74	72	2.44e-4	5.12e-4
3	73	72	2.02e-4	9.07e-3
4	77	72	1.93e-5	6.52e-3
5	80	75	5.54e-5	5.19e-4
6	80	78	1.91e-4	4.30e-5
7	80	80	1.87e-4	1.02e-3

Table 2.5: Steps and minimum cosines for the driven cavity problem.

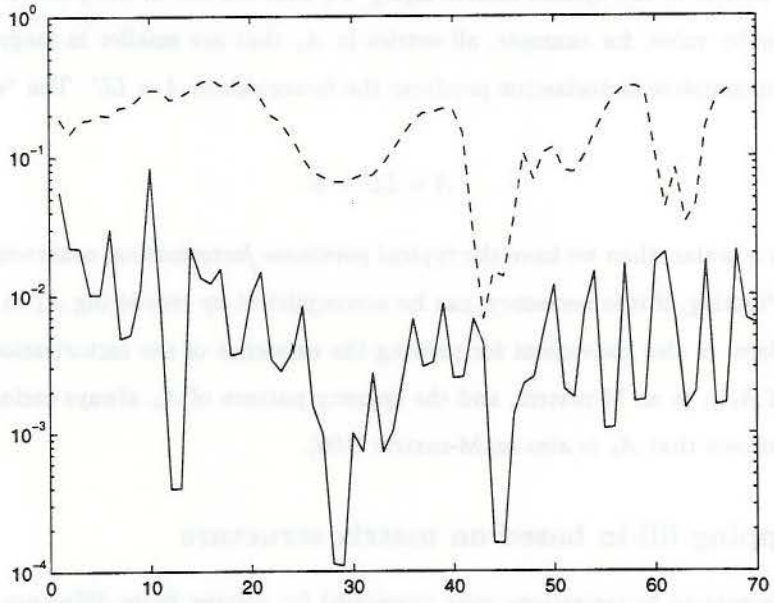


Figure 2.1: Minimum cosines in right-preconditioned Bi-CG (solid line) and symmetrically-preconditioned Bi-CG (dashed line) for the $Re = 1$ problem.

2.4 Incomplete LU preconditioners

The incomplete LU factorization of a matrix is conveniently described using the outer-product form of elimination. Starting with $A_0 = A$, denote

$$A_{k-1} = \begin{pmatrix} B_k & F_k \\ E_k & C_k \end{pmatrix}$$

and consider step k of the outer-product form of Gaussian elimination

$$A_{k-1} = \begin{pmatrix} I & 0 \\ E_k B_k^{-1} & I \end{pmatrix} \begin{pmatrix} B_k & F_k \\ 0 & A_k \end{pmatrix}$$

where $A_k = C_k - E_k B_k^{-1} F_k$. To make the factorization *incomplete*, entries are dropped in A_k, i.e., the factorization proceeds with

$$\tilde{A}_k = A_k - R_k \tag{2.15}$$

where R_k is a matrix of dropped entries. Dropping can be performed by position, for example, dropping those entries in the update matrix $E_k B_k^{-1} F_k$ that are not in the pattern of C_k. Dropping can also be done by value, for example, all entries in A_k that are smaller in magnitude than some tolerance. The incomplete factorization produces the factorization $A \approx LU$. The "error" is the term E in

$$A = LU + E.$$

If B_k is a scalar, then we have the typical *pointwise factorization*, otherwise we have a *block factorization*. Pivoting, if it is necessary, can be accomplished by reordering A_k at every step. This outer-product form is also convenient for proving the existence of the factorization for M-matrices by induction: if A_{k-1} is an M-matrix, and the sparsity pattern of A_k always includes the diagonal, then it can be shown that A_k is also an M-matrix [106].

2.4.1 Dropping fill-in based on matrix structure

The original incomplete factorizations were developed for solving finite difference equations for elliptic partial differential equations. For these problems, the structure of the incomplete triangular factors was chosen based on the structure of the gridpoint operators [33, 111, 112, 145] (see also the review [39]) and the resulting structure of the error matrix E. In most cases, the gridpoint operator

was a five-point stencil, and the stencil for the lower (upper) triangular factor was chosen to have the same pattern as the lower (upper) triangular part of the original stencil. These stencils are illustrated in Figure 2.2, along with stencils for the approximation LU and its error $E = A - LU$.

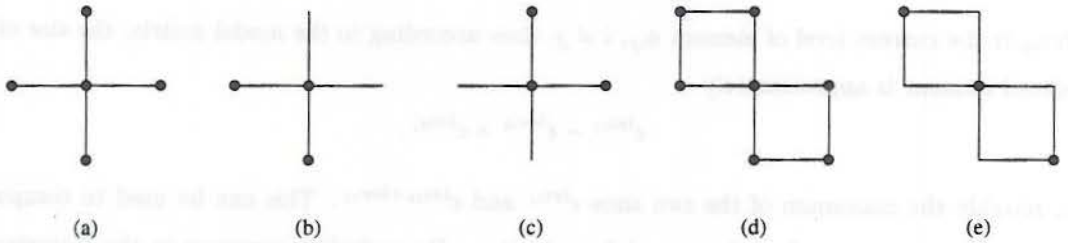


Figure 2.2: Stencils of (a) A, (b) L, (c) U, (d) LU, (e) $A - LU$.

To get a more accurate factorization, a larger stencil for the factors can be chosen, for example, by attempting to reduce the error $A - LU$. This can be done by considering the stencil of LU as the new stencil to be approximated [82]. Successively larger stencils for the lower-triangular factor defined this way are shown in Figure 2.3.

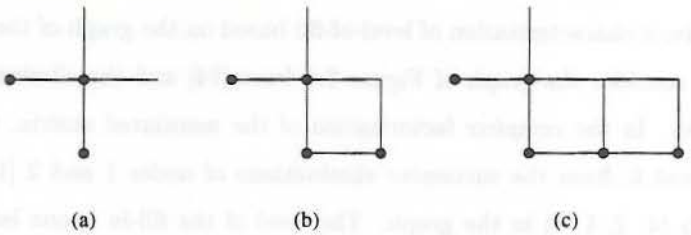


Figure 2.3: Increasingly larger stencils for L.

Incomplete factorizations were named as such when an approximate Gaussian elimination process was defined that gives sparse factors of any given pattern [106]. This generalized the earlier work on stencils to arbitrarily structured M-matrices. Choices of effective sparsity patterns for the factors were given for five- and seven-point matrices [107]. Note that the sparsity pattern should include the full diagonal, even if there are zeros on the diagonal, as in the case of some indefinite matrices.

To get more accurate factorizations for general sparse matrices, the concept of level-of-fill was introduced [147]. Suppose a matrix A has diagonal elements of size $O(\epsilon^0)$ and off-diagonal elements of size $O(\epsilon^1)$, with $\epsilon < 1$, and the exponent on ϵ indicating the *level* of the nonzero element.

As an incomplete factorization proceeds, an element a_{ij} is updated with an expression of the form

$$a_{ij} := a_{ij} - a_{ik}a_{kj}.$$

If lev_{ij} is the current level of element a_{ij}, $i \neq j$, then according to the model matrix, the size of the updated element is approximately

$$\epsilon^{lev_{ij}} - \epsilon^{lev_{ik}} \times \epsilon^{lev_{kj}},$$

i.e., roughly the maximum of the two sizes $\epsilon^{lev_{ij}}$ and $\epsilon^{lev_{ik}+lev_{kj}}$. This can be used to compute a level for each element before the actual factorization. By excluding nonzeros in the factorization that have high level, i.e., that are created by a chain of induced nonzeros, then essentially small nonzeros are dropped. For five-point matrices, retaining successively higher level elements gives the same successively more accurate stencils as those in Figure 2.3. Note that to agree with the literature, we need to redefine level as one less than what was used above. The ILU(k) factorization is thus defined as a factorization that retains all elements with level up to k. ILU(0) retains only the original nonzeros of the matrix.

There is also a characterization of level-of-fill based on the graph of the original matrix [51]. To illustrate this, consider the graph of Figure 2.4 from [74] and the elimination of the nodes in the numbered order. In the *complete* factorization of the associated matrix, there will be a fill-in between nodes 4 and 6, from the successive eliminations of nodes 1 and 2 [119]. This is because there exists a path (4, 2, 1, 6) in the graph. The level of the fill-in is one less than the length of the shortest path between nodes 4 and 6 through the eliminated nodes 1 and 2. In this case, the level is 2. Assuming that the nodes are eliminated in the natural order, then in general, the level of element a_{ij} is equal to one less than the length of the shortest path $(i, u_1, \ldots, u_m, j)$ in the original matrix, where the u_k, the eliminated nodes, are numbered less than both i and j. Note the very strong dependence on the order of elimination.

Each new edge in the path corresponds to multiplying by ϵ and inducing a nonzero in the model matrix. This graph-based characterization can also be used to determine the stencils in Figure 2.3. The important nonzeros or edges determined by these structural dropping schemes are, in some sense, those between nearby nodes.

In practice, any form (order of the loops) of Gaussian elimination may be used to compute an incomplete factorization. However, the most computationally efficient form (for nonsymmetric matrices) is probably the row-wise or column-wise form. A full-length work vector is used to hold the

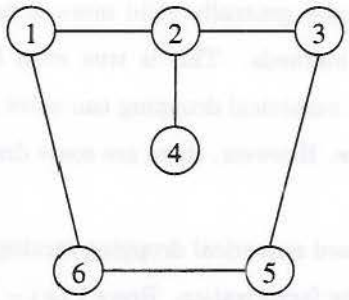


Figure 2.4: Example graph.

current row or column that is being computed, which helps minimize searching for nonzero entries. The "submatrix" form is more expensive to use, but it is more flexible and makes it possible to perform symmetric pivoting [51]. The "bordered" form will be introduced in Section 4.4.

Algorithm 2.4.1 illustrates the row-wise incomplete factorization of a matrix A. This form is suitable for sparse matrices stored in row-contiguous data structures. The algorithm computes each row of L and U together, where L is unit lower triangular, and U is upper triangular. The predetermined sparsity pattern of $L+U$ is S, and w is the full-length work vector mentioned above.

ALGORITHM **2.4.1** *Row-by-row ILU factorization*

1. *For* $i = 1, \ldots, n$ *do*
2. $\quad w_j := a_{i,j}$, $(i,j) \in S$, $\quad w_j = 0$ *otherwise*
3. $\quad$ *For* $k = 1, \ldots, i-1$ *and if* $(i,k) \in S$ *do*
4. $\quad\quad w_k := w_k / u_{kk}$
5. $\quad\quad$ *For* $j = k+1, \ldots, n$ *and if* $(i,j) \in S$ *do*
6. $\quad\quad\quad w_j := w_j - w_k u_{kj}$
7. $\quad\quad$ *EndDo*
8. $\quad$ *EndDo*
9. $\quad l_{ij} := w_j, \quad j = 1, \ldots, i-1$ *and* $l_{ii} := 1$
10. $\quad u_{ij} := w_j, \quad j = i, \ldots, n$
11. *EndDo*

2.4.2 Dropping strategies based on numerical threshold

For many types of matrices, particularly non-diagonally dominant and indefinite matrices, the model for the level-of-fill concept described above is inappropriate. For these matrices, level-of-fill may be less effective at predicting the locations of the largest entries in the factorization. As an alternative to dropping techniques based on structure, fill-in can be dropped *during* the factorization, based on their *numerical* size [109, 127, 116, 152]. This is a kind of greedy approach to minimizing E in (2.4).

Numerical dropping strategies generally yield more accurate factorizations with the same amount of fill-in than level-of-fill methods. This is true even for some diagonally dominant M-matrices. In general, ILU based on numerical dropping can solve more problems, and in fewer steps than ILU based on matrix structure. However, there are some drawbacks which will be described in Chapter 5.

To describe a threshold-based numerical dropping strategy, Figure 2.5 shows $A = LU$ in the row-wise computation of row i of the factorization. Rows 1 to $i-1$ in L and U have been completed, and rows $i+1$ to the end in A have not yet been accessed. The matrix equation represented by the

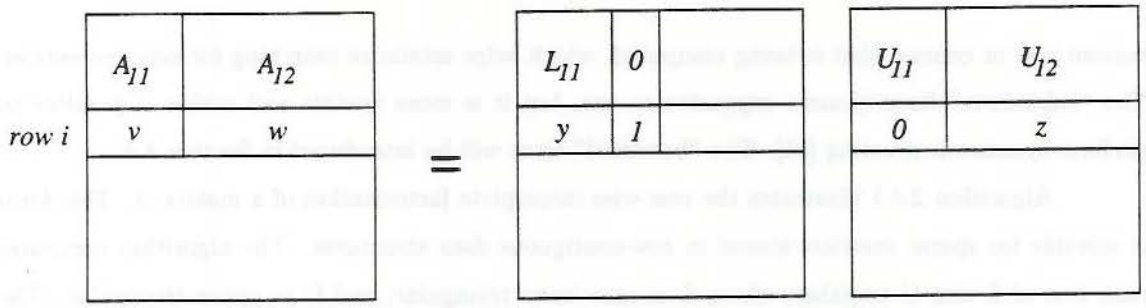


Figure 2.5: $A = LU$ in the computation of row i of the factorization.

shaded regions in the figure is

$$\begin{pmatrix} A_{11} & A_{12} \\ v & w \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ y & 1 \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & z \end{pmatrix} \tag{2.16}$$

which means that y and z (the rows to be computed in L and U) can be determined by first solving a lower-triangular system

$$U_{11}^T y^T = v^T \tag{2.17}$$

and then substituting y into

$$z = w - yU_{12}. \tag{2.18}$$

The simplest way to perform numerical dropping is to drop small entries in y and z after these vectors are computed. For example, given a parameter *droptol*, entries in y less than *droptol* are set to zero, while entries in z less than $z_1 \times$ *droptol* are set to zero, where z_1 is the first component of the vector z. (A threshold of $\|z\| \times$ *droptol* is often used instead if there is a danger that z_1 is very small.)

However, it is also possible to drop small entries in y^T during the triangular solve (2.17).

Algorithm 2.4.2 shows this operation without dropping. Note that the triangular solve is performed with SAXPY operations because only the columns of U_{11}^T are available, and not all columns of U_{11}^T may be needed.

ALGORITHM **2.4.2** *Solving $U_{11}^T y^T = v^T$ for the factorization using SAXPY operations, without dropping*

1. $y := v$
2. *For* $k = 1, \ldots, i-1$ *do*
3. $\quad y_k := y_k / u_{kk}$
4. $\quad$ *For* $j = k+1, \ldots, i-1$ *do*
5. $\quad\quad y_j := y_j - y_k u_{kj}$
6. $\quad$ *EndDo*
7. *EndDo*

To make this algorithm approximate, values of y_k less than *droptol* can be dropped after line 3. Then the work in loop 4–6 (i.e., column k of U_{11}^T) can be saved. In fact, this type of dropping introduces less error into the factorization. Consider the factorization

$$\begin{pmatrix} b & f \\ e & c \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ e/b & 1 \end{pmatrix} \begin{pmatrix} b & f \\ 0 & c - ef/b \end{pmatrix}. \tag{2.19}$$

If e/b is small and is dropped *before* it is used, then the error E of the factorization is

$$\begin{pmatrix} b & f \\ e & c \end{pmatrix} - \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} b & f \\ 0 & c \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ e & 0 \end{pmatrix}.$$

If e/b is dropped *after* it is used in the loop 4–6, then the error in the factorization is

$$\begin{pmatrix} b & f \\ e & c \end{pmatrix} - \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} b & f \\ 0 & c - ef/b \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ e & ef/b \end{pmatrix}.$$

This means it is sometimes better to drop small elements rather than eliminating and then dropping them.

Practical implementations of threshold-based ILU include an additional parameter besides *droptol* called *lfil*. This is the maximum number of nonzeros in each row y and z when a row of the factorization is computed, i.e., the largest *lfil* entries are retained in each of y and z. This implementation is called ILUT(*droptol*, *lfil*) [127]. The *lfil* parameter allows the storage requirements of the preconditioner to be known beforehand. The ILUT preconditioner, and a variant which performs partial pivoting called ILUTP will be used extensively in this thesis for comparison purposes. Ef-

ficient symmetric versions of threshold-based incomplete factorizations based on a fill-in parameter like *lfil* are described in [90] and [101].

2.4.3 Effect of ordering

As with complete factorizations, incomplete LU factorizations depend on the order in which the variables are eliminated. Thus the ordering of a matrix plays an important role in determining the quality of an ILU preconditioner. Pivoting strategies specifically designed for incomplete factorizations have been developed and perform very well for anisotropic PDE problems, but tend to be computationally expensive [51, 48]. These strategies pick the next variable to eliminate based on minimizing the norm of the matrix of dropped entries R_k (in equation (2.15)) at each step.

The remainder of the research in this area compares ordering strategies for complete factorizations (e.g., minimum degree, reverse Cuthill-McKee (RCM), nested dissection, red-black) applied to incomplete factorizations. For symmetric matrices, the common observation is that "local" orderings, such as the natural and bandwidth-reducing orderings, are best [60]. When a natural ordering is available, it should be used; other orderings may severely degrade the quality of the preconditioner. However, as the amount of fill-in is allowed to increase, the factorization approaches that of a direct solve, and direct solver orderings become beneficial.

For very nonsymmetric matrices that are difficult to solve, the observation is that minimum degree and RCM orderings are generally more superior than natural orderings [61, 19]. No mechanism is able to explain this, except that the reorderings sometimes avoid unstable triangular factors that can be produced when factoring very nonsymmetric matrices. There is also experimental evidence to support the idea that pivoting is more often required for more structurally nonsymmetric matrices [76]. Some of these aspects of reordering will be presented in more detail in Chapter 5.

2.4.4 Parallel and multilevel variants

Parallel and multilevel variants of ILU are essentially defined by a reordering of the linear system. Reorderings that expose parallelism such as the red-black reordering, however, are known to result in poor convergence rates for iterative methods. The rate can improve, however, as more fill-in is allowed, and red-black ordering can even become preferable over the natural ordering [126, 127]. Certain multilevel variants of ILU also use orderings that expose parallelism. The convergence rates of these methods are often better or more scalable than those using natural ordering, and results such as these are motivating current research in this area.

To begin, we note that if it is only required to parallelize the solves with the sparse triangular factors of an ILU factorization, then *level-scheduling* [2] is sufficient. Also, ILU is often used within other parallel preconditioners, for example, in block Jacobi preconditioning, with ILU factors used to approximately solve with the block systems. This is one of the parallel preconditioners available in the Aztec package [140]. Orderings and blockings that give good block Jacobi preconditioners have been investigated in [62, 113]. Overlapping blocks can also be used.

A simple reordering that exposes some parallelism for incomplete factorizations is the *domain decomposition reordering*. This ordering procedure begins by performing a nonoverlapped domain decomposition on the nodes in a PDE grid (or variables in the matrix adjacency graph). The interior nodes of a subdomain (those not coupled to nodes external to the subdomain) are ordered consecutively, subdomain after subdomain, followed by the interface nodes ordered at the end. If n subdomains are used, this ordering of the unknowns gives rise to matrices with the following structure:

$$\begin{pmatrix} B_1 & & & F_1 \\ & B_2 & & F_2 \\ & & \ddots & \vdots \\ & & B_n & F_n \\ E_1 & E_2 & \cdots & E_n & C \end{pmatrix}. \tag{2.20}$$

An ILU factorization algorithm for this matrix begins by performing an ILU factorization for each of the "local" B_i matrices *in parallel*. The remainder of the factorization will require some information from the local factors, and thus require some processor communication. This algorithm is known as "distributed ILU," [102] and was one of the original preconditioners in the P_SPARSLIB package [130]. This ordering is also used for domain decomposition Schur complement preconditioners (e.g., [13]). A version of these latter preconditioners called *approximate* block LU factorization will be discussed in Chapter 4.

Multicolor and independent set reorderings also reveal parallelism for ILU factorizations. When no fill-in is desired, a multicolor reordering is used. The the most common example of this is the red-black ordering for 5- and 7-point matrices, where half the variables can be eliminated in parallel, followed by the second half. If four colors are needed to color the graph of a matrix, and

the variables of like color are ordered together, then the reordered matrix has the structure

$$A = \begin{pmatrix} D_1 & A_{12} & A_{13} & A_{14} \\ A_{21} & D_2 & A_{23} & A_{24} \\ A_{31} & A_{32} & D_3 & A_{34} \\ A_{41} & A_{42} & A_{43} & D_4 \end{pmatrix}$$

where the D_i are diagonal, and the A_{ij} are general sparse blocks. All the variables corresponding to a block can be eliminated in parallel. In the triangular solve operations, all the variables in a block can be solved together as well. This approach is used in the BlockSolve95 package [89].

If fill-in is desired, then after the first elimination, the structure of the remaining block to be factored will change. In particular, the diagonal blocks will no longer be diagonal. For this reason, at each step, a block of variables that can be eliminated in parallel needs to be determined. This is accomplished by a maximal independent set reordering, i.e., by ordering first a set of variables that are not coupled. This approach is called ILU with multi-elimination (ILUM) [129].

The parallel granularity of ILUM is a row or small block of rows. To enlarge the granularity, the independent set reorderings can be replaced by domain decomposition reorderings. A parallel implementation of ILUM with domain decomposition used for the first step is described in [91]. This has been extended to use domain decomposition reordering at every step [73].

A block version of ILUM using blocks of size up to 2 by 2 by finding independent sets of *edges* has been reported to be more robust on convection-diffusion problems [134]. An algorithm to pair a node with a neighbor that has minimum degree gave the best results. Algorithms that chose edges in the independent set based on their edge weight were not as effective. A block version of ILUM that can use large sparse blocks has also recently been developed [133].

Multi-level versions of ILU such as ILUM are currently under vigorous investigation. Some variants of these methods, such as NGILU [23], use "hierarchical reorderings" of the PDE mesh points and report near grid-independent convergence of an iterative method used with this preconditioner. Connections between these methods with multigrid and other multi-level methods can be established [24, 110].

2.5 Sparse approximate inverse preconditioners

Rather than have a preconditioner approximate A, an alternative is to approximate A^{-1}. Preconditioners based on the latter are called "explicit" preconditioners, while those based on the former are called "implicit." While an approximation to A^{-1} does not require an additional inversion or solve operation, A^{-1} may be more difficult to approximate than A since usually it is dense. It is a philosophical question whether one approach is better than the other. In fairly abusive notation, an approximation to A^{-1} is often also called M, so that left-preconditioning is written

$$MAx = Mb.$$

The meaning of M should be apparent in this thesis.

The prototypical example of an implicit preconditioner is an ILU factorization, while the prototypical example of an explicit preconditioner is a sparse approximate inverse. Other approximate inverse preconditioners are polynomial preconditioners, and banded approximations to the inverse. ILU preconditioners were discussed in the previous section. In this section, we briefly introduce approximate inverse preconditioners.

One motivation to look at explicit preconditioners is that, as shown in Chapter 1, ILU factorizations often fail due to the instability of the triangular solve recurrences. In such cases, it may be attractive to approximate the inverse of the matrix directly. Approximate inverse preconditioners are also attractive from the parallel computing standpoint. The preconditioning operation is a sparse matrix-vector product, which has much more parallelism than a triangular solve operation. The construction of the approximate inverse is usually also parallelized easily. This section introduces some methods of approximating the matrix A^{-1} with a sparse matrix M by minimizing the Frobenius norm of a residual matrix such as $(I - AM)$. An important feature of this objective function is that it can be decoupled as the sum of squares of the 2-norms of the n individual columns, i.e.,

$$\|I - AM\|_F^2 = \sum_{j=1}^{n} \|e_j - Am_j\|_2^2 \tag{2.21}$$

in which e_j and m_j are the j-th columns of the identity matrix and of the matrix M, respectively. Thus, minimizing (2.21) is equivalent to minimizing the individual functions

$$\|e_j - Am_j\|_2, \quad j = 1, 2, \ldots, n. \tag{2.22}$$

Of course, if no restriction is placed on M, the exact inverse will be found. Thus a sparsity pattern (set of nonzero indices) $\mathcal{S}$ for M is prescribed, and (2.22) can be solved as a set of least-squares problems. If A is nonsingular, then the least-squares matrices have full rank.

If the normal equations are used to solve (2.21), then we need to solve the linear system

$$(A^T A M)_{ij} = A_{ij}^T, \quad (i,j) \in \mathcal{S}. \tag{2.23}$$

Explicitly, the independent linear system that needs to be solved for column j is

$$(A^T A m_j)_i = (A^T e_j)_i, \quad (i,j) \in \mathcal{S}. \tag{2.24}$$

Note that the objective function (2.21) will lead to a *right* approximate inverse. The function $\|I - MA\|_F = \|I - A^T M^T\|_F$ will give a *left* approximate inverse and is treated in exactly the same way. To be consistent in this thesis, we will use the form (2.21), which will be notationally clearer. The early development of approximate inverses, however, used the form for the left approximate inverse.

The major developments in sparse approximate inverses began with the least-squares and diagonal block method first defined by Benson [14]. Kolotilina and Yeremin [97] generalized the methods with weighted Frobenius norms. They also developed factorized approximate inverses [98] which guarantee the nonsingularity of the approximate inverse. Techniques to update the sparsity pattern $\mathcal{S}$ were developed by Cosgrove, Díaz and Griewank [50]. Later, this was shown to be effective by Grote and Huckle [80] and their paper popularized the method. Benzi et al. [18, 21] have developed factorized approximate inverses based on conjugation rather than optimization.

The next chapter proposes some new methods for constructing sparse approximate inverses. These methods attempt to reduce the cost of minimizing (2.22) by performing these minimizations *approximately* with an iterative method. In addition, the sparsity pattern $\mathcal{S}$ for the approximate inverse emerges automatically; there is no need to prescribe it beforehand.

Chapter 3

Sparse approximate inverse preconditioners

This chapter begins with a review of current techniques for approximating the inverse of a matrix with a sparse matrix. We then propose the use of iterative methods for determining the approximate inverse, first mentioning Newton's method, and then presenting iterative minimization methods for $\|I - AM\|_F$. In contrast to previous work, the key ideas here are that the minimizations are only done *approximately*, and that the iterative method naturally determines the sparsity pattern of the approximate inverse. Especially when A is relatively full, the new methods have the opportunity to be much more economical than current methods, since finding an exact minimizer M may not be necessary.

3.1 Review of current methods

3.1.1 Least-squares and diagonal block methods

The first approximations to the inverse using a norm of $(I - AM)$ were by Benson [14], in his M.S. thesis work with P. Frederickson. Techniques specific to circulant matrices were developed, and two techniques were extended to approximating the inverse of banded matrices A with banded matrices M of semibandwidth $2q+1$. The *least-squares* method [14] determines each column m_j of M independently by minimizing

$$\|e_j - Am_j\|_2$$

where e_j is the j-th unit coordinate vector. Benson solved these least-squares problems via the normal equations of size $2q+1$. The *diagonal block* method [14] constructs M such that the $2q+1$ central diagonals of $I - AM$ are zero. For each column m_j, independent linear systems involving diagonal blocks of A of size $2q+1$ are solved.

In general terms, the diagonal block method seeks M such that

$$(AM)_{ij} = \delta_{ij}, \quad (i,j) \in \mathcal{S} \tag{3.1}$$

where $\mathcal{S}$ is the set of nonzero indices in the sparsity pattern, and δ_{ij} is the Kronecker delta function. The n columns can be computed independently, as shown explicitly by

$$(Am_j)_i = (e_j)_i, \quad (i,j) \in \mathcal{S}, \quad j = 1,2,\ldots,n.$$

The systems are consistent if the principal submatrices of A are nonsingular. Kolotilina [94] has shown that for any $\mathcal{S}$, the matrix M computed this way is nonsingular if A is an H-matrix.

Another related method is the *truncation* method,

$$M_{ij} = \{A^{-1}\}_{ij}, \quad (i,j) \in \mathcal{S}, \tag{3.2}$$

i.e., the nonzero entries in M are exactly the corresponding entries in A^{-1}. There exist algorithms for determining the value of any entry of the inverse of a matrix, given its LU decomposition [138, 6]. Thus, this type of approximate inverse can be economical to compute for banded matrices, and have been used in block incomplete factorizations when the pivot blocks are banded.

In general, the least-squares and diagonal block methods seek an approximate inverse with a predetermined sparsity pattern, $\mathcal{S}$, with nnz nonzeros, and whose entries are determined through nnz equations. In the diagonal block method, each equation arises from controlling nnz entries in the product AM, in particular, the entries specified by $\mathcal{S}$, and ignoring the other entries (see (3.1)). In the least-squares method, each equation is from the normal equations for a minimization based on all the entries in AM. The diagonal block method is less accurate in this sense; note that some entries of A are not used in the computation if the pattern of M is not the same as the pattern of A. Also, $\mathcal{S}$ cannot be updated easily when constructing M (see Section 3.1.4). Apparently, for these reasons, it is not commonly used.

Benson's work was not published more widely until Benson and Frederickson generalized the sparsity pattern $\mathcal{S}$ to *q-local operators* [16]. In a grid consisting of nodes connected by edges, column i of the matrix of a q-local operator consists of nonzeros corresponding to node i and its q-th level nearest-neighbors. The matrix of a 0-local operator is a diagonal matrix. The graph of a symmetric matrix can be considered a 1-local operator; thus when $\mathcal{S}$ is the pattern of A, the 1-local

approximate inverse is being sought.

The use of approximate inverses in a parallel environment was not mentioned until Benson et al. [17]. Grote and Simon [81] implemented a least-squares method on the CM-5, where A and M were banded matrices.

Unfortunately, M determined by the least-squares method cannot be guaranteed to be nonsingular. Also, if A is symmetric, M is not necessarily symmetric. Ong [114] suggested the minimization of $\|AMA - A\|_F$, which produces a symmetric M when A is symmetric. This is true because the minimum is unique, and $\|AMA - A\|_F = \|AM^TA - A\|_F$. Other functionals that give a symmetric M are possible.

Ong also suggested a weighted diagonal block approximate inverse, by solving

$$(VAM)_{ij} = V_{ij}, \quad (i,j) \in \mathcal{S} \tag{3.3}$$

where V is a matrix, and noted that this gives the same result as the least-squares method when $V = A^T$ (although only symmetric A was considered).

3.1.2 Weighted Frobenius norms

Kolotilina and Yeremin [97] introduced the weighted Frobenius norm for the minimization

$$\min \|I - AM\|_W^2 = \min \mathrm{tr}\{(I - AM)W(I - AM)^T\}$$

where W is a positive definite, but not necessarily symmetric matrix. Minimization of this quadratic functional with respect to the entries in M yields the system of linear equations

$$\{A^T(W + W^T)AM\}_{ij} = \{A^T(W + W^T)\}_{ij}, \quad (i,j) \in \mathcal{S}$$

where $\mathcal{S}$ is again the prescribed sparsity pattern of M. If W is known explicitly, each column of M can be computed independently.

Remarkably, the least-squares, diagonal block, and truncation methods all fall under this generalization. If $W = I$, then we have the least-squares method using the normal equations. If $W = A^{-1}$ and A is symmetric, then we have the diagonal block method. If $W = A^{-T}A^{-1}$, we have the truncation method (3.2). Ong's weighted diagonal block method (3.3) was a similar generalization. Axelsson [11] has an excellent review of this work.

3.1.3 Factorized approximate inverses

To solve the problem of the possible singularity of the approximate inverse M, Kolotilina and Yeremin [98] consider factorized approximate inverses

$$A^{-1} \approx U_M L_M$$

in which L_M and U_M are approximate inverses of L_A and U_A, a set of non-unit triangular LU factors of A which are unknown. When A is SPD, only one factor is computed, and if the diagonal of this factor does not contain any zeros, the resulting preconditioner is also SPD. If L_M is computed in this case, it is an approximation to the lower triangular Cholesky factor of A.

Of course, L_M and U_M can be found with the previous methods once the factors L_A and U_A are computed. However, Kolotilina and Yeremin noticed the remarkable fact that when minimizing[1]

$$\|I - L_M L_A\|_F$$

using the normal equations, the factorization $A = L_A U_A$ does not need to be known.

Consider the symmetric case. Define $\mathcal{S}_L$ to be the sparsity pattern of the lower triangular factor (including the diagonal), and substitute $\mathcal{S}_L$ for $\mathcal{S}$, L_M for M and L_A for A in the normal equations for the left approximate inverse

$$(MAA^T)_{ij} = A^T_{ij}, \quad (i,j) \in \mathcal{S}$$

to get

$$\{L_M L_A L_A^T\}_{ij} = \{L_A^T\}_{ij}, \quad (i,j) \in \mathcal{S}_L$$

or

$$\{L_M A\}_{ij} = \{L_A^T\}_{ij}, \quad (i,j) \in \mathcal{S}_L.$$

Since $\{L_A^T\}_{ij}$ for $(i,j) \in \mathcal{S}_L$ is a diagonal matrix, the off-diagonal entries of L_A do not need to be known.

[1] The form for the left approximate inverse is used here for notational convenience.

Since the diagonal of L_A is also usually not known, Kolotilina and Yeremin solve instead

$$(\widetilde{L_M}A)_{ij} = \delta_{ij}, \quad (i,j) \in \mathcal{S}_L$$

and then scale $L_M = D^{-1}\widetilde{L_M}$ where D is the diagonal of $\widetilde{L_M}$. This produces a preconditioned matrix $L_M A L_M^T$ such that its diagonal contains all ones. The above equation looks like (3.1), and thus precautions such as diagonal compensation [7] must be taken to ensure that L_M will not be singular. A parallel implementation of this preconditioner has been made by Field [70].

The factorized approximate inverse can be easily extended to the nonsymmetric case by using weighted Frobenius norms or the weighted diagonal block method. The minimization of the functional $\|I - LAU\|_F$ to generate factorized approximate inverses will be explored in Section 4.3.

Factorized forms based on conjugation rather than the minimization of a Frobenius norm have been developed by Benzi et al. [18, 21]. In the nonsymmetric case, biconjugation produces two sets of A-biconjugate vectors W and Z, i.e.,

$$W^T A Z = D$$

where D is a diagonal matrix. The inverse

$$A^{-1} = Z D^{-1} W^T$$

is an explicit product of matrices. To make the biconjugation process economical but approximate, small entries in W and Z are dropped during their computation. The resulting W and Z are sparse, leading to a sparse approximate inverse preconditioner.

The matrices W and Z are produced by applying a biconjugation process to some initial W_0 and Z_0 which are nonsingular. If W_0 and Z_0 are the identity matrix, then the process produces W and Z that are triangular, and thus guaranteed to be nonsingular as long as the biconjugation process does not break down. In this case, the familiar LDU factorization is produced. The following is this version of the biconjugation algorithm without dropping. In the algorithm, w_j and z_j denote the j-th columns of W and Z, respectively, and a_j^T and c_j denote the j-th row and column of A, respectively.

ALGORITHM **3.1.1** *Inverse triangular factorization by biconjugation*

1. *Begin with* $[w_1, \ldots, w_n] = I$ *and* $[z_1, \ldots, z_n] = I$
2. $d_1 := a_{11}$
3. *For* $i = 2, 3, \ldots, n$ *do*
4. *For* $j = 1, 2, \ldots, i-1$ *do*
5. $\rho := a_j^T z_i$
6. $z_i := z_i - (\rho/d_j) z_j$
7. $\rho := c_j^T w_i$
8. $w_i := w_i - (\rho/d_j) w_j$
9. *EndDo*
10. $d_i := a_j^T z_i$ or $d_i := c_j^T w_i$
11. *EndDo*

This is the left-looking, delayed update variant, where each column of W or Z is completed computed before beginning to compute the next one. To make W and Z sparse, entries in z_j on line 6 and in w_j on line 8 are dropped if they are smaller than some tolerance. Dropping is usually not done based on the size of (ρ/d_j) alone. The incomplete algorithm is very similar to a form of incomplete Gram-Schmidt orthogonalization [135, 123].

If no dropping is used, the algorithm does not break down if all the leading principal submatrices are nonsingular. If dropping is used, then the algorithm does not break down if A is a symmetric H-matrix [18]. Orderings analogous to minimum degree can be used to improve the performance of this preconditioner, particularly on anisotropic problems [28].

3.1.4 Updating the sparsity pattern

Except for the biconjugation method, all the methods described above choose sparsity pattern $\mathcal{S}$ beforehand and keeps it constant during the computation of M. Although there are strategies to select $\mathcal{S}$ based on the cost of constructing and using M [100], a good *a priori* choice based on the structure of the inverse is very difficult. A wavelet basis has been proposed to make the approximation better for PDE problems [38].

For the least-squares method, Cosgrove, Díaz and Griewank [50] introduced the idea of augmenting $\mathcal{S}$ while M is being constructed. An initial approximate inverse is constructed with an initial sparsity pattern $\mathcal{S}$. Then, new nonzero locations are introduced into $\mathcal{S}$, and the new minimization problem is solved. This process is repeated until a maximum number of nonzeros has been reached, or $\|I - AM\|_1$ satisfies a lower bound. The 1-norm was used since it can be checked column-by-column.

A numerical test is used to determine which nonzero locations should be added to $\mathcal{S}$. A second issue is which locations to check, and in what order, since it is expensive to check all of them.

For the j-th column, the numerical test is based on the reduction in the optimal residual norm, $\min \|e_j - Am\|_2$ when the sparsity pattern of m is augmented. This reduction can be determined without computing the corresponding new solution. Denote $\mathcal{M}$ as the set of vectors with a given sparsity pattern, and $\mathcal{M}^*$ as the set of vectors with the same pattern plus an additional nonzero at position k. Also define

$$m_j = \arg\min_{m \in \mathcal{M}} \|e_j - Am\|_2, \qquad r = e_j - Am_j$$
$$m_j^* = \arg\min_{m \in \mathcal{M}^*} \|e_j - Am\|_2, \qquad r^* = e_j - Am_j^*.$$

Let $\hat{A}$ denote the columns of A that correspond to the indices in $\mathcal{M}$. The best improvement to the residual norm for column j can be measured by

$$\|r^*\|_2^2 - \|r\|_2^2 = -\frac{(r^T A e_k)^2}{\|PAe_k\|_2^2}$$

where P is the orthogonal projector onto the complement of the range of $\hat{A}$, i.e., the orthogonal projector onto the null space of $\hat{A}^T$ [50, 78]. Note that Ae_k is simply the k-th column of A.

Since PAe_k is rather expensive to compute for many k's, Cosgrove, Díaz and Griewank [50] use the lower bound

$$\frac{|r^T A e_k|}{\|PAe_k\|_2} \geq \frac{|r^T A e_k|}{\|Ae_k\|_2} \tag{3.4}$$

and consider locations k whose lower bound is above a certain tolerance.

Instead of finding the exact minimizer m_j^* for the augmented problem, Grote and Huckle [80] hold the current solution constant and find the scalar μ that minimizes

$$\|r - \mu Ae_k\|_2$$

for every possible additional nonzero at location k. The solution of this one-dimensional minimization is simply $\mu = -r^T Ae_k / \|Ae_j\|_2^2$, and the new residual r^* satisfies

$$\|r^*\|_2^2 - \|r\|_2^2 = -\frac{(r^T A e_k)^2}{\|Ae_k\|_2^2}.$$

The right-hand side is used to test new locations. By comparing this with the lower bound in (3.4),

we see that the approximation made by Cosgrove, Díaz and Griewank [50] gives the same numerical test. In later work, Gould and Scott [78] found a relatively cheap computation of $\|PAe_k\|_2$ and use the exact test (the left-hand side of (3.4)). A disadvantage of these techniques is that these tests are expensive to perform if there are many candidate entries to evaluate.

If the least-squares problems are solved via a QR factorization, then the solution m_j^* to the augmented least-squares problem should be found via updating the QR factorization. Huckle [86] discusses many possible ways of solving the least-squares problem that take advantage of the sparse structure of the least-squares matrix.

Notice that for most locations k, there will be no improvement to the residual norm, specifically, when the nonzero locations of r and Ae_k do not intersect (r is sparse since A and m_k are sparse). Grote and Huckle [80] determine which columns in A intersect with r to make the process more efficient. The best s locations (e.g., $s \leq 5$) are added to the pattern of the j-th column of S.

Although their work was earlier, Cosgrove, Díaz and Griewank [50] used a more complicated scheme that depends on the order in which locations are checked. Locations that introduce fewer new rows h in the least-squares subproblems are checked first. For each h, the set of all possible candidates is divided into three prioritized classes. The first location that satisfies their numerical test is added. See [50] for details.

It is possible to interpret the candidate locations as nodes on a grid when A is the result of a discretization. Consider approximating column j starting with a single entry at location j. The residual r corresponds to the first-level neighbors of node j, and the candidate locations consist of these plus the second-level neighbors. When candidates are actually added, the new residual contains nonzeros corresponding to the neighbors of these added candidates. The pattern of r grows from a point on the grid.

Two recent implementations of Grote and Huckle's method [80] are Deshpande et al. [53] and Barnard and Clay [12]. The former implements the method approximately, by only considering the first-level neighbors of a node as candidate fill-in positions. The latter uses dynamic load balancing and other optimizations in its parallel implementation.

There is no simple way to update the sparsity pattern if A is not available explicitly, as in the case of Kolotilina and Yeremin's factorized approximate inverses, where L_A and U_A are not known. It is also difficult to update the sparsity pattern in the diagonal block methods.

3.2 Newton iteration

As an alternative to directly minimizing the objective function (2.21), an approximate inverse may also be computed using an iterative process known as the method of Hotelling and Bodewig [85]. This method, which is modeled after Newton's method for solving $f(x) \equiv 1/x - a = 0$, has many similarities to our descent methods to be described later. Starting with an initial approximation M_0, this iteration has the form

$$M_{i+1} = M_i(2I - AM_i).$$

For convergence, it is required that the spectral radius of $I - AM_0$ be less than one, and if we choose an initial guess of the form $M_0 = \alpha A^T$ then convergence is achieved if

$$0 < \alpha < \frac{2}{\rho(AA^T)}.$$

In practice, we can follow Pan and Reif [117] and use

$$\alpha = \frac{1}{\|AA^T\|_1}$$

for the right approximate inverse. As the iterations progress, M becomes denser and denser, and a natural idea here is to perform the above iteration in sparse mode [104], i.e., drop some elements in M, or else the iterations become too expensive. In this case, however, the convergence properties of the Newton iteration are lost. Some comparative results for this method will be presented in Section 3.10.

3.3 Global matrix iterations

In this section we describe a "global" approach to minimizing (2.21), where we use a descent-type method, treating M as an unknown sparse matrix. The objective function (2.21) is a quadratic function on the space of $n \times n$ matrices, viewed as objects in $\mathcal{R}^{n^2}$. The actual inner product on the space of matrices with which the function (2.21) is associated is

$$\langle X, Y \rangle = \mathrm{tr}(Y^T X) \tag{3.5}$$

where $\mathrm{tr}(\cdot)$ is the trace function. One possible descent-type method we may use is steepest descent which we will describe later. In the following, we will call the array representation of an n^2 vector X the $n \times n$ matrix whose column vectors are the successive n-vectors of X.

In descent algorithms a new iterate M_{new} is defined by taking a step along a selected direction G, i.e.,

$$M_{new} = M + \alpha G$$

in which α is selected to minimize the objective function associated with M_{new}. This is achieved by taking

$$\alpha = \frac{\langle R, AG \rangle}{\langle AG, AG \rangle} = \frac{\mathrm{tr}(R^T AG)}{\mathrm{tr}\,((AG)^T AG)} \tag{3.6}$$

where $R = I - AM$ is the residual matrix. Note that the denominator may be computed as $\|AG\|_F^2$. After each of these descent steps is taken, the resulting matrix M will tend to become denser. It is therefore essential to apply some kind of numerical dropping, either to the new M or to the search direction G before taking the descent step. In the first case, the descent nature of the step is lost, i.e., it is no longer guaranteed that $F(M_{new}) \leq F(M)$, while in the second case, the fill-in in M is more difficult to control. We will discuss both these alternatives in Section 3.7.

The simplest choice for the descent direction G is to take it to be the residual matrix $R = I - AM$, where M is the new iterate. The corresponding descent algorithm is referred to as the minimal residual (MR) algorithm. In the simpler case where numerical dropping is applied to M, our global minimal residual algorithm will have the following form.

ALGORITHM **3.3.1** *Global minimal residual descent algorithm*

1. *Select an initial M*
2. *Until convergence do*
3. *Compute $G := I - AM$*
4. *Compute α by (3.6)*
5. *Compute $M := M + \alpha G$*
6. *Apply numerical dropping to M*
7. *EndDo*

Another popular choice is to take G to be the direction of steepest descent, i.e., the direction opposite to the gradient. Thinking in terms of n^2 vectors, the gradient of F can be viewed as an n^2 vector g such that

$$F(x + e) = F(x) + (g, e) + O(\|e\|^2)$$

where $(\cdot, \cdot)$ is the usual Euclidean inner product. If we represent all vectors as 2-dimensional $n \times n$

arrays, then the above relation is equivalent to

$$F(X+E) = F(X) + \langle G, E\rangle + O(\|E\|^2).$$

This allows us to determine the gradient as an operator on arrays, rather than n^2 vectors, as is done in the next proposition.

Proposition 3.3.1 *The array representation of the gradient of F with respect to M is the matrix*

$$G = -2A^T R$$

in which R is the residual matrix $R = I - AM$.

Proof. For any matrix E we have

$$\begin{aligned} F(M+E) - F(M) &= \mathrm{tr}(I - A(M+E))^T(I - A(M+E)) \\ &\qquad -\mathrm{tr}(I - A(M))^T(I - A(M)) \\ &= \mathrm{tr}\left[(R - AE)^T(R - AE) - R^T R\right] \\ &= -\mathrm{tr}\left[(AE)^T R + R^T AE - (AE)^T(AE)\right] \\ &= -2\mathrm{tr}(R^T AE) + \mathrm{tr}(AE)^T(AE) \\ &= -2\langle A^T R, E\rangle + \langle AE, AE\rangle. \end{aligned}$$

Thus, the differential of F applied to E is the inner product of $-2A^T R$ with E plus a second order term. The gradient is therefore simply $-2A^T R$. □

The steepest descent algorithm consists of simply replacing G in line 3 of the MR algorithm described above by $G = A^T R$. This algorithm can be very slow in some cases, since it is essentially a steepest descent-type algorithm applied to the normal equations.

In either global steepest descent or minimal residual, we need to form and store the G matrix explicitly. The scalars $\|AG\|_F^2$ and $\mathrm{tr}(R^T AG)$ can be computed from the successive columns of AG, which can be generated, used, and discarded. Therefore, we need not store the matrix AG.

3.4 Column-oriented iterations

This section describes column-oriented algorithms which consist of minimizing the individual objective functions (2.21). We perform this minimization by taking a sparse initial guess and solving approximately the n linear subproblems

$$Am_j = e_j, \quad j = 1, 2, \ldots, n \tag{3.7}$$

with a few steps of a nonsymmetric descent-type method, such as MR or untruncated GMRES. For this method to be efficient, the iterative method must work in sparse mode, i.e., m_j is stored and operated on as a sparse vector, and the Arnoldi basis in GMRES is kept in sparse format.

In the following MR algorithm, n_i iterations are used to solve (3.7) approximately for each column, giving an approximation to the j-th column of the inverse of A. Each initial m_j is taken from the columns of an initial guess, M_0. Again, we assume numerical dropping is applied to M. In the GMRES version of the algorithm, we never use restarting since n_i is typically very small. Also, a variant called FGMRES [125] which allows an arbitrary Arnoldi basis, is actually used in this case.

ALGORITHM **3.4.1** *Minimal residual iteration*

1. *Start: set* $M = M_0$
2. *For each column* $j = 1, \ldots, n$ *do*
3. *Define* $m_j = Me_j$
4. *For* $i = 1, \ldots, n_i$ *do*
5. $r_j := e_j - Am_j$
6. $\alpha_j := \frac{(r_j, Ar_j)}{(Ar_j, Ar_j)}$
7. $m_j := m_j + \alpha_j r_j$
8. *Apply numerical dropping to* m_j
9. *EndDo*
10. *EndDo*

Thus, the algorithm computes the current residual r_j and then minimizes the residual norm

$$\|e_j - Am_{j,new}\|_2$$

in the set $m_j + \alpha r_j$.

In the sparse implementation of MR and GMRES, the matrix-vector product, SAXPY, and dot product kernels now all entirely involve sparse vectors. The matrix-vector product is much more efficient if the sparse matrix is stored by columns since all the entries do not need to be traversed. Efficient codes for all these kernels may be constructed which utilize a full n-length work vector [57].

Columns from an initial guess M_0 for the approximate inverse are used as the initial guesses for the iterative solution of the linear subproblems. There are two obvious choices: $M_0 = \alpha I$ and $M_0 = \alpha A^T$. The scale factor α is chosen to minimize the spectral radius $\rho(I - \alpha AM)$. Denoting the initial guess as $M_0 = \alpha M$ and writing

$$\frac{\partial}{\partial \alpha}\|I - \alpha AM\|_F^2 = \frac{\partial}{\partial \alpha}\mathrm{tr}[(I - \alpha AM)^T(I - \alpha AM)] = 0$$

leads to

$$\alpha = \frac{\mathrm{tr}(AM)}{\mathrm{tr}(AM(AM)^T)}.$$

The transpose initial guess is more expensive to use because it is denser than the identity initial guess. However, for very indefinite systems, this guess immediately produces a symmetric positive definite preconditioned system, corresponding to the normal error equations. Depending on the structure of the inverse, a denser initial guess is often required to involve more of the matrix A in the computation. Interestingly, the cheaper the computation, the more it uses only "local" information, and the less able it may be to produce a good approximate inverse.

The choice of initial guess also depends to some degree on "self-preconditioning" which we describe next. Additional comments on the choice of initial guess will be presented there.

It should be noted that approximate minimization in another form was also proposed by Kolotilina, Nikishin, and Yeremin [96], also to reduce the possibly high cost of the least-squares subproblems. However, in their approach, the *normal equations* for the least-squares subproblems (2.24) are solved approximately with a few steps of the conjugate gradient method, i.e., the sparsity pattern still must be prescribed beforehand.

3.5 Self-preconditioning

The approximate solution of the linear subproblems (3.7) using an iterative method suffers from the same problems as solving the original problem if A is indefinite or poorly conditioned. However, the linear systems may be preconditioned with the columns that have already been computed. More precisely, each system (3.7) for approximating column j may be preconditioned with M_0' where the first $j-1$ columns of M_0' are the m_k that already have been computed, $1 \le k < j$, and the remaining columns are the initial guesses for the m_k, $j \le k \le n$.

This suggests that it is possible to define *outer* iterations that sweep over the matrix, as

well as *inner* iterations that compute each column. On each subsequent outer iteration, the initial guess for each column is the previous result for that column. This technique usually results in much faster convergence of the approximate inverse.

Unfortunately with this approach, the parallelism of constructing the columns of the approximate inverse simultaneously is lost. However, there is another variant of self-preconditioning that is easier to implement and more easily parallelizable. Simply, all the inner iterations are computed simultaneously and the results of all the columns are used as the self-preconditioner for the next outer iteration. Thus, the preconditioner for the inner iterations changes only after each outer iteration. The performance of this variant usually lies between full self-preconditioning and no self-preconditioning. A more reasonable compromise is to compute blocks of columns in parallel, and some (inner) self-preconditioning may be used.

Self-preconditioning is particularly valuable for very indefinite problems when combined with a scaled transpose initial guess; the initial preconditioned system AM_0 is positive definite, and the subsequent preconditioned systems somewhat maintain this property, even in the presence of numerical dropping. Self-preconditioning with a transpose initial guess, however, may produce worse results if the matrix A is very ill-conditioned. In this case, the initial worsening of the conditioning of the system is too severe, and the alternative scaled identity initial guess should be used instead. We have also found cases where self-preconditioning produces worse results, usually for positive definite problems; this is not surprising, since the minimizations would progress very well, only to be hindered by self-preconditioning with a poor approximate inverse in the early stages. Numerical evidence of these phenomena will be provided in Section 3.10.

Algorithm 3.5.1 implements the minimal residual iteration with self-preconditioning. In the algorithm, n_o outer iterations and n_i inner iterations are used. Again, $M = M_0$ initially. We have also indicated where numerical dropping might be applied.

ALGORITHM **3.5.1** *Self-preconditioned minimal residual iteration*

1. *Start:* $M = M_0$
2. *For outer* $= 1, 2, \ldots, n_o$ *do*
3. *For each column* $j = 1, \ldots, n$ *do*
4. *Define* $s := m_j = Me_j$
5. *For inner* $= 1, \ldots, n_i$ *do*
6. $r := e_j - As$
7. $z := Mr$
8. $q := Az$
9. $\alpha := \frac{(r,q)}{(q,q)}$
10. $s := s + \alpha z$

11. *Apply numerical dropping to s*
12. *EndDo*
13. *Update j-th column of M: $m_j := s$*
14. *EndDo*
15. *EndDo*

In a computer implementation, M is stored as n sparse vectors, each holding up to *lfil* entries. M is thus constructed in place.

3.6 Convergence behavior of self-preconditioned MR

Next we consider the convergence behavior of the algorithms for constructing a sparse approximate inverse. In particular, we analyze the column-oriented algorithm when self-preconditioning is used, but no numerical dropping is applied.

Consider each column being updated individually by exactly one step of the MR algorithm. Let M be the current approximate inverse at a given substep. The self-preconditioned MR iteration for computing the j-th column of the next approximate inverse is obtained by the following sequence of operations.

1. $r_j := e_j - Am_j = e_j - AMe_j$
2. $t_j := Mr_j$
3. $\alpha_j := \frac{(r_j, At_j)}{(At_j, At_j)}$
4. $m_j := m_j + \alpha_j t_j$

Note that α_j can be written as

$$\alpha_j = \frac{(r_j, AMr_j)}{(AMr_j, AMr_j)} = \frac{(r_j, Br_j)}{(Br_j, Br_j)}$$

where we define

$$B = AM$$

to be the preconditioned matrix at the given substep. We now drop the index j to simplify the notation. The new residual associated with the current column is given by

$$\begin{aligned} r^{new} &= r - \alpha At \\ &= r - \alpha AMr \\ &\equiv r - \alpha Br. \end{aligned}$$

We use the orthogonality of the new residual against AMr to obtain

$$\|r^{new}\|_2^2 = \|r\|_2^2 - \alpha^2 \|Br\|^2.$$

Replacing α by its value defined above we get

$$\|r^{new}\|_2^2 = \|r\|_2^2 \left[1 - \left(\frac{(Br,r)}{\|Br\|_2\|r\|_2}\right)^2\right].$$

Thus, at each inner iteration, the residual norm for the j-th column is reduced according to the formula

$$\|r^{new}\|_2 = \|r\|_2 \sin \angle(r, Br) \tag{3.8}$$

in which $\angle(u,v)$ denotes the acute angle between the vectors u and v. Assuming that each column converges, the preconditioned matrix B will converge to the identity. As a result of this, the angle $\angle(r,Br)$ will tend to $\angle(r,r) = 0$ and therefore the convergence ratio $\sin\angle(r,Br)$ will also tend to zero, showing superlinear convergence.

We now consider equation (3.8) more carefully in order to analyze more explicitly the convergence behavior. We will denote by R the residual matrix $R = I - AM$. We observe that

$$\begin{aligned} \sin\angle(r,Br) &= \min_{\alpha} \frac{\|r - \alpha\, Br\|_2}{\|r\|_2} \\ &\leq \frac{\|r - Br\|_2}{\|r\|_2} \equiv \frac{\|Rr\|_2}{\|r\|_2} \\ &\leq \|R\|_2. \end{aligned}$$

This results in the following statement.

Proposition 3.6.1 *Assume that the self-preconditioned MR algorithm is employed with one inner step per iteration and no numerical dropping. Then the 2-norm of each residual $e_j - Am_j$ of the j-th column is reduced by a factor of at least $\|I - AM\|_2$, where M is the approximate inverse before the current step, i.e.,*

$$\|r_j^{new}\|_2 \leq \|I - AM\|_2 \, \|r_j\|_2 \tag{3.9}$$

In addition, the Frobenius norm of the residual matrices $R_k = I - AM_k$ obtained after each outer iteration, satisfies

$$\|R_{k+1}\|_F \leq \|R_k\|_F^2. \tag{3.10}$$

As a result, when the algorithm converges, it does so quadratically.

Proof. Inequality (3.9) was proved above. To prove quadratic convergence, we first transform this inequality by using the fact that $\|X\|_2 \leq \|X\|_F$ to obtain

$$\|r_j^{new}\|_2 \leq \|R_{k,j}\|_F \|r_j\|_2.$$

Here the k index corresponds to the outer iteration and the j-index to the column. We note that the Frobenius norm is reduced for each of the inner steps corresponding to the columns, and therefore

$$\|R_{k,j}\|_F \leq \|R_k\|_F.$$

This yields

$$\|r_j^{new}\|_2^2 \leq \|R_k\|_F^2 \|r_j\|_2^2$$

which, upon summation over j gives

$$\|R_{k+1}\|_F \leq \|R_k\|_F^2.$$

This completes the proof. □

It is also easy to show a similar result for the following variations:

1. Global MR iterations;
2. MR with an arbitrary number of inner steps;
3. GMRES(m) for an arbitrary m.

These follow from the fact that the algorithms deliver an approximate matrix or column which has a smaller residual than what we obtain with one inner step MR.

We emphasize that quadratic convergence is guaranteed only at the limit and that the above theorem does not prove convergence. In the presence of numerical dropping, the proposition does not hold.

3.7 Numerical dropping strategies

There are many options for numerical dropping. So far, to ease the presentation, we have only discussed the case where dropping is performed on the solution vectors or matrices. Section 3.7.1 discusses this case in more detail, while Section 3.7.2 discusses the case where dropping is applied to the search directions. In the latter case, the descent property of the algorithms is maintained.

3.7.1 Dropping in the solution

When dropping is performed on the solution, we have options for

1. when dropping is performed, and
2. which elements are dropped.

In the previous algorithms, we have made the first point precise; however, there are other alternatives. For example, dropping may be performed only after M or each column of M is computed. Typically this option is too expensive, but as a compromise, dropping may be performed at the end of a few inner iterations, before M is updated, namely before step 13 in Algorithm 3.5.1. Interestingly, we found experimentally that this option is not always better.

In GMRES, the Krylov basis vectors are kept sparse by dropping elements just after the self-preconditioning step, before the multiplication by A.

To address which elements are dropped, we can utilize a dual threshold strategy based on a drop tolerance, *droptol*, and the maximum number of elements per column, *lfil*. By limiting the maximum number of elements per column, the maximum storage for the preconditioner is known beforehand.

The drop tolerance may be applied directly to the elements to be dropped: i.e., elements are dropped if their magnitude is smaller than *droptol*. However, we found that this strategy could cause *spoiling* of the minimization, i.e., the residual norm may increase after several steps, along with a deterioration of the quality of the preconditioner.

To try to determine a more optimal dropping strategy, we use a simple perturbation analysis. We denote by m_j the current column, and by $\hat{m}_j$ the perturbed column formed by adding the sparse column d in the process of numerical dropping. The new column and corresponding residual are therefore

$$\hat{m}_j = m_j + d, \quad \hat{r}_j = \boldsymbol{r}_j - Ad.$$

The square of the residual norm of the perturbed m_j is given by

$$\|\hat{r}_j\|_2^2 = \|r_j\|_2^2 - 2(d, A^T r_j) + \|Ad\|_2^2. \tag{3.11}$$

Recall that $-2A^T r_j$ is the gradient of the function (2.21). As is expected from standard results in optimization, if d is in the direction opposite to the gradient, and if the steplength small enough, we can achieve a decrease of the residual norm. Spoiling occurs when $(d, A^T r_j)$ is close to zero so that for practical sizes of $\|d\|_2$, $\|Ad\|_2^2$ becomes dominant, causing an increase in the residual norm.

Consider specifically the situation where only one element is dropped, and assume that all the columns Ae_i of A have been pre-scaled so that $\|Ae_i\|_2 = 1$. In this case, $d = m_{ij}e_i$ and the above equation becomes

$$\|\hat{r}_j\|_2^2 = \|r_j\|_2^2 - 2m_{ij}(e_i, A^T r_j) + m_{ij}^2. \tag{3.12}$$

A strategy could therefore be based on attempting to make the function

$$\|\hat{r}_j\|_2^2 - \|r_j\|_2^2 = -2m_{ij}(e_i, A^T r_j) + m_{ij}^2 \tag{3.13}$$

nonpositive, a condition which is easy to verify. This suggests selecting elements to drop in m_j only at indices i where the selection function (3.13) is zero or negative. However, note that this is not entirely rigorous since in practice a few elements are dropped at the same time. Thus we do not entirely perform dropping via numerical values alone. In a two-stage process, we first select a number of candidate elements to be dropped based only on the numerical size as determined by a certain tolerance. Among these, we drop all those that satisfy the condition

$$\rho_{ij} = -2m_{ij}(e_i, A^T r_j) + m_{ij}^2 < tol_2$$

or we can keep those *lfil* elements that have the largest ρ_{ij}.

Another alternative is based on attempting to achieve maximum reduction in the function (3.13). Ideally, we wish to have

$$m_{ij} = (e_i, A^T r_j)$$

since this will achieve the "optimal" reduction in (3.13)

$$\|\hat{r}_j\|_2^2 - \|r_j\|_2^2 = -m_{ij}^2.$$

This leads to the alternative strategy of dropping elements in positions i of m_j where $m_{ij} - (e_i, A^T r_j)$ are the smallest. We found, however, that this strategy produces poorer results than the previous one, and neither of these strategies completely eliminate spoiling.

3.7.2 Dropping in the search direction

Dropping may be performed on the search direction G in Algorithm 3.3.1, or equivalently in r_j and z in Algorithms 3.4.1 and 3.5.1 respectively. In these cases, the descent property of the algorithms is maintained, and the problem of spoiling is avoided.

Starting with a sparse initial guess, the allowed number of fill-ins is gradually increased at each iteration. For an MR-like algorithm, the search direction d is derived by dropping entries from the residual direction r. So that the sparsity pattern of the solution x is controlled, d is chosen to have the same sparsity pattern as x, plus one new entry, the largest entry in absolute value. No drop tolerance is used. Minimization is performed by choosing the step-length as

$$\alpha = \frac{(r, Ad)}{(Ad, Ad)}$$

and thus the residual norm for the new solution is guaranteed to be not more than the previous residual norm. In contrast to Algorithm 3.4.1, the residual may be updated with very little cost. The iterations may continue as long as the residual norm is larger than some threshold, or a set number of iterations may be used.

If A is indefinite, the normal equations residual direction $A^T r$ may be used as the search direction, or simply to determine the location of the new fill-in. It is interesting to note that the largest entry in $A^T r$ gives the greatest residual norm reduction in a one-dimensional minimization. When fill-in is allowed to increase gradually using this search direction, this technique becomes very similar to the adaptive selection scheme of [80]. The effect is also similar to self-preconditioning when the initial guess to the approximate inverse is a scaled transpose of A.

At the end of each iteration, it is possible to use a second stage that exchanges entries in the solution with new entries if this causes a reduction in the residual norm. This is required if the sparsity pattern in the approximate inverse needs to change as the approximations progress. We have found this to be necessary, particularly for very unstructured matrices, but have not yet found a strategy that is genuinely effective. As a result, approximations using numerical dropping in the solution are often better, even though the scheme just described has a stronger theoretical

justification, similar to that of [80]. This also shows that the adaptive scheme of [80] may benefit from such an exchange strategy.

Algorithm 3.7.1 implements a minimal residual-like algorithm with this numerical dropping strategy. The number of inner iterations is usually chosen to be *lfil* or somewhat larger.

ALGORITHM **3.7.1** *Self-preconditioned MR algorithm with dropping in search direction*

1. *Start:* $M = M_0$
2. *For each column* $j = 1, \ldots, n$ *do*
3. *Define* $m_j = Me_j$
4. $r_j := e_j - Am_j$
5. *For inner* $= 1, 2, \ldots, n_i$ *do*
6. $t := Mr_j$
7. *Choose* d *to be* t *with the same pattern as* m_j*;*
   If $nnz(m_j) <$ lfil *then add one entry which is the*
   largest remaining entry in absolute value
8. $q := Ad$
9. $\alpha := \frac{(r_j, q)}{(q, q)}$
10. $m_j := m_j + \alpha d$
11. $r_j := r_j - \alpha q$
12. *EndDo*
13. *EndDo*

If dropping is applied to the unpreconditioned residual, then economical use of this approximate inverse technique is not limited to approximating the solution to linear systems with sparse coefficient matrices or sparse right-hand sides. An approximation may be found, for example, to a factorized matrix, or a dense operator which may only be accessed with a matrix-vector product. Such a need may arise, for instance, when preconditioning row projection systems.

We must mention here that any adaptive strategy such as this one for choosing the sparsity pattern makes massive parallelization of the algorithm more difficult. If, for instance, each processor has the task of computing a few columns of the approximate inverse, it is not known beforehand which columns of A must be fetched into each processor.

3.8 Nonsingularity of the approximate inverse

It cannot be proved that the approximate-inverse M is nonsingular unless the approximation is accurate enough, typically to a level that is impractical to attain. This is stated in the following proposition.

Proposition 3.8.1 *Assume that A is nonsingular and that the residual of the approximate inverse M satisfies the relation*

$$\|I - AM\| < 1 \tag{3.14}$$

where $\|\cdot\|$ is any consistent matrix norm. Then M is nonsingular.

Proof. The result follows immediately from the equality

$$AM = I - (I - AM) \equiv I - N \tag{3.15}$$

and the well-known fact that if $\|N\| < 1$, then $I - N$ is nonsingular. □

We note that the result is true in particular for the Frobenius norm, which, although not an induced matrix norm, is consistent.

We should point out that the result does not tell us anything about the degree of sparsity of the resulting approximate inverse M. It may well be the case that in order to guarantee nonsingularity, we must have an M that is dense, or nearly dense. In fact, in the particular case where the norm in the proposition is the 1-norm, it has been proved by Cosgrove, Díaz and Griewank [50] that the approximate inverse may be *structurally dense*, in that it is always possible to find a sparse matrix A for which M will be dense if $\|I - AM\|_1 < 1$.

The drawback of using M that is possibly singular is the need to check the solution, or the actual residual norm at the end of the linear iterations. In practice, however, we have not noticed premature terminations due to a singular preconditioned system.

3.9 Cost of constructing the approximate inverse

The cost of computing the approximate inverse is relatively high. Let n be the dimension of the linear system, n_o be the number of outer iterations, and n_i be the number of inner iterations ($n_o = 1$ in Algorithm 3.7.1).

We approximate the cost by the number of sparse matrix-sparse vector multiplications in the sparse mode implementation of MR and GMRES. Profiling for a few problems shows that this operation accounts for about three-quarters of the time when self-preconditioning is used. The remaining time is used primarily by the sparse dot product and sparse SAXPY operations, and in the case of sparse mode GMRES, the additional work within this algorithm.

If Algorithm 3.5.1 is used, two sparse mode matrix-vector products are used, the first one for computing the residual; three are required if self-preconditioning is used. In Algorithm 3.7.1 the residual may be updated easily and stored, or recomputed as in Algorithm 3.5.1. Again, an additional product is required for self-preconditioning. The cost is simply nn_on_i times the number of these sparse mode matrix-vector multiplications. Each multiplication is inexpensive, but the actual cost depends on the number of nonzeros in the columns in M. Dropping in the search directions, however, is slightly more expensive because more iterations are typically used (e.g., one iteration for each fill-in).

In Newton iteration, two sparse matrix-sparse matrix products are required, although the convergence rate may be doubled with a form of Chebyshev acceleration [118]. Global iterations without self-preconditioning require three matrix-matrix products. These costs are comparable to the column-oriented algorithms.

3.10 Numerical experiments and observations

Experiments with the algorithms and options described in this chapter were performed with matrices from the Harwell-Boeing and FIDAP collections (see Section 1.2). The matrices were scaled so that the 2-norm of each column is unity. In each experiment, we report the number of GMRES(20) steps to reduce the initial residual of the right-preconditioned linear system by 10^{-5}. A zero initial guess was used, and the right-hand side was constructed so that the solution is a vector of all ones. A dagger (†) in the tables below indicates that there was no convergence in 500 iterations. In some tables we also show the value of the Frobenius norm (2.21). Even though this is the function that we minimize, we see that it is not always a reliable measure of GMRES convergence. All the results are shown as the outer iterations progress. In Algorithm 3.5.1 (dropping in solution vectors) one inner iteration was used unless otherwise indicated; in Algorithm 3.7.1 (dropping in residual vectors) one additional fill-in was allowed per iteration. Various codes in Fortran 77, C++, and Matlab were used, and run in 64-bit precision on Sun workstations and a Cray C90 supercomputer.

We begin with a comparison of Newton, "global" and column-oriented iterations. Our early numerical experiments showed that in practice, Newton iteration converges very slowly initially and is more adversely affected by numerical dropping. Global iterations were also worse than column-oriented iterations, perhaps because a single α defined by (3.6) is used, as opposed to one for each column in the column-oriented case. Table 3.1 gives some numerical results for the WEST0067

matrix; the number of GMRES iterations is given as the number of outer iterations increases. The MR iteration used self-preconditioning with a scaled transpose initial guess. Dropping based on numerical values in the intermediate solutions was performed on a column-by-column basis, although in the Newton and global iterations this restriction is not necessary. In the presence of dropping ($lfil = 10$), we did not find much larger matrices where Newton iteration gave convergent GMRES iterations. Scaling each iterate M_i by $1/\|AM_i\|_1$ did not alleviate the effects of dropping. The superior behavior of global iterations in the presence of dropping in Table 3.1 was not typical.

No dropping					
	1	2	3	4	5
Newton	†	414	158	100	41
Global	228	102	25	16	11
MR	130	35	13	10	6

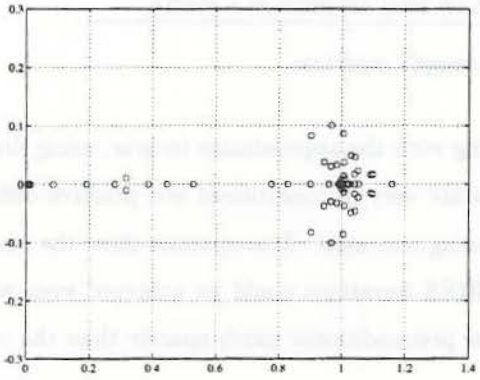
Dropping: $lfil = 10$, $droptol = 0.001$					
	1	2	3	4	5
Newton	463	†	435	†	457
Global	241	87	46	35	26
MR	281	120	86	61	43

Table 3.1: Comparison of Newton, global, and column MR iterations. Table shows the number of preconditioned GMRES(20) steps required for solving WEST0067 when 1 to 5 outer iterations are used to construct the preconditioner.

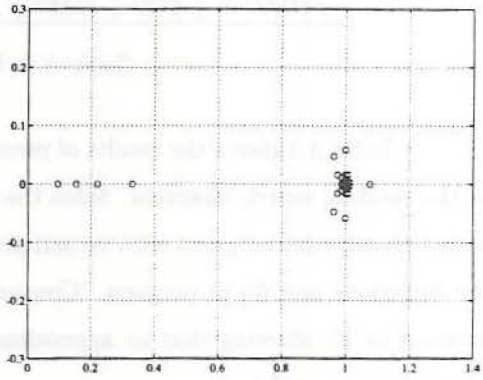
The eigenvalues of the preconditioned WEST0067 matrix are plotted in Fig. 3.1, both with and without dropping, using column-oriented MR iterations. As the iterations proceed, the eigenvalues of the preconditioned system become closer to 1. Numerical dropping has the effect of spreading out the eigenvalues. When dropping is severe and spoiling occurs, we have observed two phenomena: either dropping causes some eigenvalues to become negative, or some eigenvalues stay clustered around the origin.

Next we show results for all the positive definite matrices in the FIDAP collection. Table 3.2 lists some statistics about these matrices. The combination of ill-conditioning and indefiniteness of the other matrices was too challenging for our methods, and their results are not shown here.

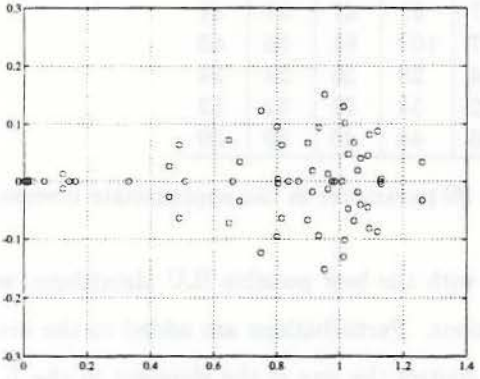
All the matrices are also symmetric, except for FIDAP07. None of the matrices could be solved with ILU(0) or ILUT [127], a threshold incomplete LU factorization, even with large amounts of fill-in. Our experience with these matrices is that they produce unstable L and U factors in incomplete factorizations.



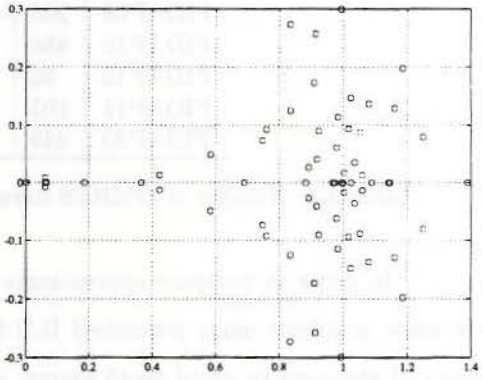
(a) no dropping, $n_o = 2$



(b) no dropping, $n_o = 4$



(c) $lfil = 10, n_o = 2$



(d) $lfil = 10, n_o = 4$

Figure 3.1: Eigenvalues of preconditioned system, WEST0067.

Matrix	n	nnz	
FIDAP03	1821	52685	Flow past a circular cylinder
FIDAP07	1633	54543	Natural convection in a square cavity
FIDAP09	3363	99471	Jet impingement in a narrow channel
FIDAP10	2410	54840	2D flow over multiple steps in a channel
FIDAP13	2568	75628	Axisymmetric flow through a poppet valve
FIDAP15	6867	98671	2D spin up of a liquid in an annulus
FIDAP33	1733	22189	2D radiation heat transfer in a cavity

Table 3.2: FIDAP example matrices.

Table 3.3 shows the results of preconditioning with the approximate inverse, using dropping in the residual search direction. Since the problems are very ill-conditioned but positive definite, a scaled identity initial guess with no self-preconditioning was used. The columns show the results as the iterations and fill-in progress. Convergent GMRES iterations could be achieved even with *lfil* as small as 10, showing that an approximate inverse preconditioner much sparser than the original matrix can be useful.

Matrix	10	20	30	40	50	60	70
FIDAP03	159	133	47	42	40	38	38
FIDAP07	33	23	18	17	14	14	14
FIDAP09	203	117	67	51	47	41	41
FIDAP10	438	191	107	107	81	66	63
FIDAP13	56	39	34	28	26	24	24
FIDAP15	103	83	62	54	53	52	52
FIDAP33	249	105	86	44	40	39	39

Table 3.3: Number of GMRES iterations vs. *lfil* parameter in the approximate inverse.

In order to compare approximate inverses with the best possible ILU algorithms, we solve the same problems using perturbed ILU factorizations. Perturbations are added to the inverse of diagonal elements to avoid small pivots, and thus control the size of the elements in the L and U factors. We use a two-level block ILU strategy called BILU(0)–SVD(α), that uses a modified singular value decomposition to invert the blocks [150]. When a block $A = U\Sigma V^T$ needs to be inverted, it is replaced by the perturbed inverse $M = V\bar{\Sigma}^{-1}U^T$, where $\bar{\Sigma}$ is Σ with its singular values thresholded by $\alpha\sigma_1$, a factor of the largest singular value.

Table 3.4 shows the results, using a block size of 4. For this set of problems, the results are comparable to approximate inverse preconditioning, but with less work to compute the preconditioner. None of the problems converged, however, for $\alpha = 0.1$, and there was not one α that

gave the best result for all problems. As we will see below, these perturbed ILU factorizations are unfortunately not applicable to very unstructured matrices.

Matrix	α	
	0.3	1.0
FIDAP03	†	170
FIDAP07	19	39
FIDAP09	28	72
FIDAP10	66	140
FIDAP13	20	40
FIDAP15	†	119
FIDAP33	†	149

Table 3.4: BILU(0)–SVD(α) preconditioner.

We now show our main results in Table 3.5 for several standard matrices in the Harwell-Boeing collection. All the problems are nonsymmetric and indefinite, except for SHERMAN1 which is symmetric, negative definite. In addition, SAYLR3 is singular. SHERMAN2 was reordered with reverse Cuthill-McKee to attempt to change the sparsity pattern of the inverse. Again, we show the number of GMRES iterations required for convergence against the number of outer iterations used to compute the approximate inverse. A scaled A^T was used as the initial guess for the approximate inverse. When columns in the initial guess contained more than *lfil* nonzeros, dropping was applied to the guess. Numerical dropping was applied to the intermediate vectors in the solution, retaining *lfil* nonzeros and using no drop tolerance.

For problems SHERMAN2, WEST0989, GRE1107 and NNC666, the results become worse as the outer iterations progress. This spoiling effect is due to the fact that the descent property is not maintained when dropping is applied to the intermediate solutions. This is not the case when dropping is applied to the search direction, as seen in Table 3.3.

Except for SAYLR3, the problems that could not be solved with ILU(0) also could not be solved with BILU(0)–SVD(α), nor with ILUTP, a variant of ILUT more suited to indefinite problems since it uses partial pivoting to avoid small pivots [123]. ILUTP also substitutes $(10^{-4} + \delta)$ times the norm of the row when it is forced to take a zero pivot, where δ is the drop tolerance. ILU factorization strategies simply do not apply in these cases.

We have shown the best results after a few trials with different parameters. The method is sensitive to the widely differing characteristics of general matrices, and apart from the comments we have already made for selecting an initial guess and whether or not to use self-preconditioning,

Matrix	n	ILU(0)	g/m	p/u	*lfil*	n_i	1	2	3	4	5
PORES2	1244	44	m	p	30	2	†	†	52	30	19
PORES3	532	38	m	u	10	1	†	†	421	150	112
SHERMAN1	1000	32	m	u	10	1	224	187	96	74	60
SHERMAN2	1080	8	m	p	50	1	†	†	147	46	136
SHERMAN3	5005	56	m	u	10	1	499	363	239	192	148
SHERMAN4	1104	22	m	u	10	1	87	69	43	42	41
SHERMAN5	3312	22	g	p	20	2	†	148	107	70	60
SAYLR3	1000	†	m	u	10	1	223	188	96	74	60
WEST0497	497	†	g	p	50	5	†	†	†	80	20
WEST0989	989	†	m	p	50	2	†	†	303	†	†
GRE1107	1107	†	m	p	50	2	421	†	†	†	†
GRE216B	216	†	m	p	5	1	3	3	3	3	3
NNC261	261	†	m	p	20	2	†	39	20	17	14
NNC666	666	†	m	p	50	2	†	†	427	147	173

g/m = GMRES or MR
p/u = self-preconditioned or unself-preconditioned

Table 3.5: Number of GMRES(20) iterations vs. n_o.

there is no general set of parameters that works best for constructing the approximate inverse. The following two tables illustrate some different behaviors that can be seen for three very different matrices. LAPL0324 is a standard symmetric positive definite 2-D Laplacian matrix of order 324. WEST0067 and PORES3 are both indefinite; WEST0067 has very little structure, while PORES3 has a symmetric pattern. Table 3.6 shows the number of GMRES(20) iterations and Table 3.7 shows the Frobenius norm of the residual matrix against the number of outer iterations that were used to compute the approximate inverse.

Matrix	*lfil*	init	p/u	1	2	3	4	5
WEST0067	none	A^T	p	130	35	13	10	6
	none	A^T	u	484	481	†	472	†
	none	I	p	†	†	†	†	†
	10	A^T	p	281	120	86	61	43
LAPL0324	none	A^T	p	466	200	50	21	12
	none	A^T	u	21	17	12	12	10
	none	I	p	16	15	11	11	9
	10	A^T	u	30	22	17	17	17
PORES3	none	A^T	p	†	†	†	†	†
	none	A^T	u	†	†	274	174	116
	10	A^T	u	†	†	421	150	112

Table 3.6: Number of GMRES(20) steps for 1 to 5 outer iterations used to construct the approximate inverse.

Matrix	*lfil*	init	p/u	1	2	3	4	5
WEST0067	none	A^T	p	4.43	3.21	2.40	1.87	0.95
	none	A^T	u	6.07	6.07	6.07	6.07	6.07
	none	I	p	8.17	8.17	8.17	8.17	8.17
	10	A^T	p	4.77	4.26	4.42	4.92	6.07
LAPL0324	none	A^T	p	7.91	5.69	4.25	3.12	2.23
	none	A^T	u	6.62	4.93	4.00	3.41	3.00
	none	I	p	5.34	4.21	3.53	3.08	2.75
	10	A^T	u	6.54	4.81	4.07	3.82	3.92
PORES3	none	A^T	p	10.78	9.30	8.25	7.66	7.16
	none	A^T	u	12.95	12.02	11.48	10.82	10.20
	10	A^T	u	12.94	12.02	11.48	10.82	10.23

Table 3.7: $\|I - AM\|_F$ for 1 to 5 outer iterations used to construct the approximate inverse.

Chapter 4

Applications of the approximate inverse

4.1 Improving a preconditioner

In the algorithms of the previous chapter, a matrix M was sought to make AM close to the identity matrix. To be more general, we can seek instead an approximation to some matrix B. Thus we consider

$$F(M) = \|B - AM\|_F^2 \tag{4.1}$$

in which B is some matrix to be defined. As before, this objective function decouples into the set of problems

$$\|b_j - Am_j\|_2, \quad j = 1, 2, \ldots, n. \tag{4.2}$$

Most of the techniques already described will admit this generalization. In our case, we treat (4.2) as a set of sparse linear systems

$$Am_j = b_j, \quad j = 1, 2, \ldots, n$$

which have sparse right-hand sides b_j, and for which sparse approximations to the solutions m_j are desired. We refer to techniques that solve linear systems with these constraints as *sparse approximate inverse techniques*. These techniques will be used in several applications in this chapter.

Another way of viewing the concept of approximately minimizing (4.1) is that of improving an existing preconditioner, B. Once we find a matrix M whose objective function (4.1) is small enough, then the preconditioner for the matrix A is defined by

$$P = MB^{-1}.$$

This implies that B is a matrix which is easy to invert, or rather, that solving systems with B should be inexpensive. At one extreme when $B = A$, the best M is the identity matrix, but solves with B are expensive. At the other extreme, we find our standard situation which corresponds to $B = I$,

and which is characterized by trivial B-solves but expensive to obtain M matrices. In between these two extremes there are a number of appealing compromises, perhaps the simplest being the block diagonal of A.

For a numerical example of improving a preconditioner, we use approximate inverses to improve the block-diagonal preconditioners for the ORSREG1, ORSIRR1 and ORSIRR2 matrices from the Harwell-Boeing collection (see Section 1.2). These matrices were generated from oil reservoir simulation problems. The experiments used dropping on numerical values with *lfil* = 10, and *droptol* = 0.001. In Table 4.1, *block size* is the block size of the block-diagonal preconditioner, and *block precon* is the number of GMRES iterations required for convergence when the block-diagonal preconditioner is used alone. The number of GMRES iterations is shown against the number of outer iterations used to improve the preconditioner.

Matrix	n	block size	block precon	1	2	3	4	5
ORSREG1	2205	21	117	49	59	95	73	71
ORSIRR1	1030	8	253	98	104	108	85	88
ORSIRR2	833	8	253	136	99	98	92	81

Table 4.1: Number of GMRES steps required for convergence when 1 to 5 iterations are used to improve a block-diagonal preconditioner.

4.2 Approximate pseudo-inverses

There are instances where A is extremely ill-conditioned and possibly very indefinite. In such cases, iterative minimization with the matrix A may be very slow. An alternative is to seek an approximate pseudo-inverse matrix, for example,

$$M \approx (A^T A)^{-1} A^T.$$

One could obtain the approximate inverse of $A^T A$ using one of the approaches described earlier. However, this approach will fail because as for A, we are likely to find a cluster of eigenvalues of $A^T A$ around the origin. The simplest remedy is simply to shift them away from the origin. Thus, we consider the algorithm:

ALGORITHM **4.2.1** *Approximate pseudo-inverse*

1. *Select a positive* α *and compute* $B = A^T A + \alpha I$
2. *Apply numerical dropping to* B
3. *Compute an approximate inverse* C *of* B
4. *Compute* $M = CA^T$

The choice of the scalar α impacts the performance in the following way. A large α will make step 3 very easy to perform, not only because the descent algorithms will converge quickly but also because for larger α, there are small elements in the actual inverse. On the other hand a large α will lead to a poor approximate inverse. Note that without dropping, and if the inversion in step 3 is done exactly, then the eigenvalues of MA are given by

$$\frac{\sigma_i}{\sigma_i + \alpha}$$

where the σ_i's are the squares of the singular values of A. Thus, the largest singular values are mapped close to unity whereas the smallest ones are roughly divided by α. For example, we can say that all σ_i's that are greater than or equal to α will be mapped into the interval $[1/2, 1]$. Similarly, all σ_i's satisfying

$$\sigma_i \geq \frac{\alpha^2}{1 - \alpha}$$

will be mapped into the interval $[\alpha, 1]$.

We show the results of a few numerical experiments with the WEST0067 matrix from a chemical engineering application in the Harwell-Boeing collection. One inner iteration was used, with a scaled identity initial guess to approximate the inverse of $A^T A + \alpha I$. Table 4.2 shows the results with no dropping. As clearly seen, the ultimate quality of the preconditioner decreases as α increases. However, the approximate inversion of $A^T A + \alpha I$ is much easier, as seen by the faster convergence of the residual norm. In Table 4.3, dropping is applied using *lfil* = 10.

4.3 Triangular factorized approximate inverses

4.3.1 Alternate L and U minimization

Approximate inverse techniques based on minimizing $\|I - AM\|_F$ cannot guarantee that the approximate inverse M is nonsingular unless M is accurate enough. Triangular factorized approximate inverses as described in Section 3.1.3, however, are trivially nonsingular, since each of the triangu-

Number of GMRES iterations

α	1	2	3	4	5
0.00	252	75	18	12	8
0.01	267	67	19	14	11
0.05	259	65	26	19	19
0.10	254	65	30	25	25

$\|I - (A^T A + \alpha I)C\|_F$

α	1	2	3	4	5
0.00	4.53	3.34	2.50	1.92	1.06
0.01	4.47	3.12	1.83	0.64	0.06
0.05	4.22	2.37	0.75	0.06	0.00
0.10	3.94	1.76	0.35	0.01	0.00

Table 4.2: WEST0067, Approximate pseudo-inverse, no dropping.

Number of GMRES iterations

α	1	2	3	4	5
0.00	302	261	340	†	†
0.05	305	320	436	†	†
0.10	280	209	377	481	†
0.25	225	148	†	387	477

$\|I - (A^T A + \alpha I)C\|_F$

α	1	2	3	4	5
0.00	4.68	6.30	7.59	8.20	8.71
0.05	4.43	6.01	7.05	7.76	8.12
0.10	4.18	5.68	6.57	7.34	7.91
0.25	3.48	4.21	4.93	5.31	5.57

Table 4.3: WEST0067, Approximate pseudo-inverse, $lfil = 10$.

lar factors are nonsingular. In addition, factorized forms of the approximate inverse are effectively denser than unfactorized forms: the product contains more nonzeros than the sum of the nonzeros in the factors. On the other hand, the two methods described in Section 3.1.3 have certain disadvantages: the technique proposed by Kolotilina and Yeremin [97] cannot update the sparsity pattern of the approximation dynamically, and the incomplete biconjugation algorithm of Benzi et al. [18, 21] cannot be parallelized easily.

In this section, we propose two alternative techniques for computing triangular factorized approximate inverses that do not have these disadvantages. The first method is straightforward, and is based on minimizing

$$\|I - LAU\|_F \tag{4.3}$$

so that $UL \approx A^{-1}$. Since U is really a *right* approximate inverse of $(LA)^{-1}$ and L is a *left* approximate inverse of $(AU)^{-1}$, in practice this factorized preconditioner should be applied in a split fashion, i.e., the system $Ax = b$ should be preconditioned as

$$LAUy = Lb, \quad x = Uy.$$

The function (4.3) is nonlinear in the entries in L and U. Thus, to minimize (4.3), we can *alternately* determine better minimizers L and U, by holding one factor constant while determining the other. Obviously, it is not necessary to determine the optimal minimizer for a certain sparsity pattern at each step. Let l_j and u_j denote the j-th column of L^T and U respectively. The following describes this alternating algorithm, using n_o "outer" iterations.

ALGORITHM **4.3.1** *Alternate L and U minimization of* $\|I - LAU\|_F$

1. *Select lower and upper triangular initial guesses* $L^{(0)}$ *and* $U^{(0)}$
2. *For* $k = 1, \ldots, n_o$ *do*
3. *For* $j = 1, \ldots, n$ *do*
4. *Solve* $\left(L^{(k-1)}A\right) u_j^{(k)} = e_j$ *approximately, using* $u_j^{(k-1)}$ *as initial guess*
5. *EndDo*
6. *For* $j = 1, \ldots, n$ *do*
7. *Solve* $\left((U^{(k-1)})^T A^T\right) l_j^{(k)} = e_j$ *approximately, using* $l_j^{(k-1)}$ *as initial guess*
8. *EndDo*
9. *EndDo*

Note that if A is SPD, then only the equations involving u_j are solved, with L replaced by U^T. Thus only half the computations are performed, and the resulting preconditioner UU^T is also SPD.

In lines 4 and 7, a small number of minimal residual iterations are used for the approximate solve. As before, we allow the sparsity pattern for these approximate solutions to emerge automatically, and dropping is used to keep the solutions sparse. (The sparsity patterns can also be set and enlarged dynamically, as in the methods described in Section 3.1.4). Also in lines 4 and 7, all the rows of $L^{(k+1)}$ and all the columns of $U^{(k+1)}$ can be computed in parallel.

It is also possible to exchange the order of the k and j loops in the algorithm, so that each row of L or each column of U is fully computed before continuing on to the next row or column. This algorithm has better data locality, but if L and U are immediately updated with the new row or column, then the new algorithm has very little parallelism.

Algorithm 4.3.1 can also be self-preconditioned like the previous algorithms. In line 4, the right self-preconditioner is either $U^{(k)}$ or U updated with the latest columns. Similarly, in line 7, the right self-preconditioner is $(L^{(k)})^T$ or L updated with the latest rows.

At this point, we mention how to perform one step of the approximate solves in lines 4 and 7. Consider seeking an approximate solution to $LAu_j = e_j$. The resulting approximation u_j should have zero entries at indices $j+1$ to n. Thus the dropping scheme must be sure to drop entries at these locations.

4.3.2 Approximate inverse factors based on bordering

The second method, which we call *bordered approximate inverse factors* [135] seeks triangular L and U factors such that

$$LAU \approx D$$

where D is some unknown diagonal matrix. To describe the method, let A_{j+1} be the $(j+1)$-st leading principal submatrix of A. Then by writing the relation $L_{j+1}A_{j+1}U_{j+1} = D_{j+1}$ as

$$\begin{pmatrix} L_j & 0 \\ l_j & 1 \end{pmatrix} \begin{pmatrix} A_j & -v_j \\ -w_j & \alpha_{j+1} \end{pmatrix} \begin{pmatrix} U_j & u_j \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} D_j & 0 \\ 0 & d_{j+1} \end{pmatrix} \tag{4.4}$$

it can be seen that u_j and l_j can be computed by solving

$$A_j u_j = v_j \tag{4.5}$$

$$A_j^T l_j^T = w_j^T \tag{4.6}$$

and by computing d_{j+1} as

$$d_{j+1} = \alpha_{j+1} - w_j u_j - l_j v_j + l_j A_j u_j.$$

This formula for d_{j+1} makes the diagonal of $(LAU - D)$ zero.

The matrices L and U can be computed independently of each other. More importantly, all rows of L and all columns of U can also be computed independently. In (4.4) we have assumed that $L_j A_j U_j = D_j$, although this is not necessary for the $(j+1)$-st step.

To solve (4.5) and (4.6) approximately, again we use a few minimal residual iterations, starting with a sparse initial guess. (Another sparse approximate inverse technique such as one described in Section 3.1.4 may also be used.) Note that if we actually attempt to minimize the weighted norms

$$\|L_j(v_j - Au_j)\|_2 \quad \text{and} \quad \|(w_j - l_j A)U_j\|_2$$

for $1 \leq j \leq n$, we will equivalently minimize $\|D - LAU\|_F$.

If self-preconditioning is used, then the system (4.5), for example, may be preconditioned as

$$D_j^{-1} L_j A_j U_j \bar{u}_j = D_j^{-1} L_j v_j, \quad u_j = U_j \bar{u}_j.$$

In this case, L_j and U_j or some formative version of them must be available before computing u_j and l_j.

An attractive feature of this method is that the preconditioner computed with any accuracy can be proven to exist if A is SPD. We only need to note that d_{j+1} computed in (4.3.2) is the $(j+1, j+1)$ element of $L_{j+1} A_{j+1} L_{j+1}^T$, which is positive because A is SPD [135]. There is no comparable result for other triangular factorized sparse approximate inverses.

4.3.3 Comparison with other factorized forms

It is interesting to compare the above two algorithms to the algorithm of Kolotilina and Yeremin [97]. The nonsymmetric version of this algorithm corresponds to the *weighted* diagonal block method described at the end of Section 3.1.1. Note that this means that the nonsymmetric version does not perform the least-squares minimization in the Frobenius norm. To determine an approximate inverse L with sparsity pattern $\mathcal{S}_L$ of the lower triangular factor of $A = L_A U_A$, the weighted diagonal block method is

$$\{L L_A U_A\}_{ij} = \{U_A\}_{ij}, \quad (i,j) \in \mathcal{S}_L$$

where the weight matrix is U_A so that $L_A U_A$ can be replaced by the known quantity A. Precautions must be taken when the weight matrix is not positive definite. Also, as in Section 3.1.3, the diagonal of U_A is not known, so the equation

$$\{\tilde{L}A\}_{ij} = \delta_{ij}, \quad (i,j) \in \mathcal{S}_L \tag{4.7}$$

is solved instead. To determine the U factor of the approximate inverse, we have the corresponding equation

$$\{A\tilde{U}\}_{ij} = \delta_{ij}, \quad (i,j) \in \mathcal{S}_U \tag{4.8}$$

where $\mathcal{S}_U$ is the chosen sparsity pattern for U. There are a variety of options for scaling $\tilde{L}$ and $\tilde{U}$ to produce L and U.

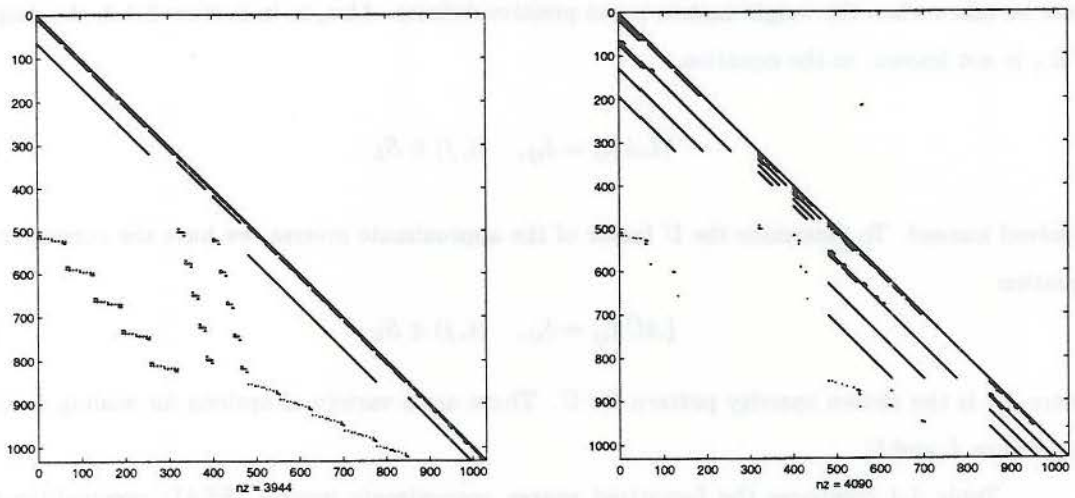
Table 4.4 compares the factorized sparse approximate inverse (FSAI) preconditioner of Kolotilina and Yeremin [97] with the bordered approximate inverse factors (BAIF) preconditioner. The latter used only 2 to 4 minimal residual steps to compute the preconditioner. The table shows the number of GMRES iterations to reduce the initial residual norm of the ORSIRR1 problem by 10^{-5}. The fill-in parameter for the bordered approximate inverse factors was chosen so that all the preconditioners use about the same storage.

Method	GMRES(60) iterations
FSAI	116
BAIF, 4 MR steps	76
BAIF, 3 MR steps	92
BAIF, 2 MR steps	106

Table 4.4: FSAI and BAIF preconditioner for ORSIRR1.

Figure 4.1 compares the sparsity pattern of the lower triangular factors of the approximate inverses for the FSAI and BAIF algorithms. The pattern of FSAI was chosen to be the pattern of the lower triangular part of the original matrix. The difference in the two patterns and the results in the table above shows that it is important to choose a proper sparsity pattern beforehand, or else use a dynamic scheme to select the sparsity pattern.

Let us compare the types of computations that are required to compute column j of U. In



(a) FSAI pattern (b) BAIF pattern

Figure 4.1: Pattern of the lower triangular approximate inverse factor for ORSIRR1.

the alternate minimization method, we need to approximately solve

$$(LA)u_j = e_j$$

with the restriction that u_j is zero at entries $j+1$ to n. The matrix L is an initial guess, possibly I, during the first outer loop of Algorithm 4.3.1. In the bordered approximate inverse factors method, we need to approximately solve

$$A_j u_j = v_j$$

where v_j is part of a column in the upper triangular part of A. We can also approximately minimize $\|L_j(v_j - A_j u_j)\|_2$ using some matrix L_j. In the method of Kolotilina and Yeremin, we need to solve

$$(A\tilde{u}_j)_i = (e_j)_i, \quad (i,j) \in \mathcal{S}_U.$$

All three algorithms are different.

4.4 ILUS using the approximate inverse technique

4.4.1 Factorization based on bordering

The approximate inverse technique can be used to compute ILU factorizations in sparse skyline format. This format stores the lower triangular factors L as sparse rows, and the upper triangular factor U as sparse columns. This allows the condition number of the incomplete L and U factors to be estimated during the factorization, and appropriate action may be taken as required. Like ILUT, fill-in is limited by both the drop-tolerance *droptol*, and the number of nonzeros *lfil* in each row of L or column of U. The latter parameter allows the maximum storage for the preconditioner to be known beforehand.

The preconditioner, which we call ILUS, defines the procedure for computing an incomplete factorization by threshold for nonsymmetric matrices in sparse skyline format. This is valuable if the original matrix is stored in skyline format, since conversion to row- or column-based data structures to compute an incomplete factorization is expensive and often requires a copy of the matrix. The regular skyline format, where all elements within the profile of the matrix are stored, is commonly used in finite element computations, particularly when direct methods are employed, see for example [84].

ILUS is an incomplete form of LDU Gaussian elimination based on bordering [115, 128]. Let A_{k+1} be the $(k+1)$-st leading principal submatrix of A and assume we have the decomposition $A_k = L_k D_k U_k$. Then we can compute the factorization of A_{k+1} using

$$\begin{pmatrix} A_k & v_k \\ w_k & \alpha_{k+1} \end{pmatrix} = \begin{pmatrix} L_k & 0 \\ y_k & 1 \end{pmatrix} \begin{pmatrix} D_k & 0 \\ 0 & d_{k+1} \end{pmatrix} \begin{pmatrix} U_k & z_k \\ 0 & 1 \end{pmatrix} \tag{4.9}$$

in which

$$z_k = D_k^{-1} L_k^{-1} v_k \tag{4.10}$$

$$y_k = w_k U_k^{-1} D_k^{-1} \tag{4.11}$$

$$d_{k+1} = \alpha_{k+1} - y_k D_k z_k. \tag{4.12}$$

Thus, we obtain each row and column of the factorization by solving two unit lower triangular systems and computing a scaled dot product. However, sparse approximate solutions to the triangular systems are required, since we do not want the preconditioner to be dense or expensive to compute.

In addition, a data structure that accesses the strictly lower-triangular part of A by rows and the strictly upper-triangular part of A by columns is required. The skyline or sparse skyline format described above is appropriate for this purpose. This format should also be used to store the resultant L and U factors. In the symmetric case, ILUS is equivalent to an incomplete Cholesky factorization; half the work can be saved, since the computation of y_k is not necessary.

A sparse approximate inverse technique is used to approximately solve (4.10) and (4.11) to give sparse approximate solutions. In general, the approximate inverse technique consisting of minimizing $\|e - Am\|_2$ with respect to a sparse vector m can be considered the sparse approximate solution to a sparse linear system with a sparse right-hand side, i.e., e is now a sparse vector. By using an iterative technique for the approximate solves, the sparsity pattern for the ILUS preconditioner emerges automatically.

It is also possible to compute sparse approximate solutions to the lower triangular systems (4.10) and (4.11) using a truncated Neumann series [45]. ILUS will also be mentioned again in Chapter 5 in an algorithm to compute L and U factors which are structurally symmetric.

4.4.2 Companion structure

In the approximate inverse technique, it is necessary to perform matrix-vector multiplications in sparse-sparse mode, by accumulating linear combinations of the columns of A. Unfortunately, the unit lower triangular matrices L_k and U_k^T are stored by rows, making them inconducive for this operation. The data structure for storing the triangular matrices must be augmented by a linked-list companion structure, which points to the entries in the matrix column by column. We describe this structure now.

Suppose the matrix L is generated row by row and stored in compressed sparse row (CSR) format [124], using the arrays `A(NNZ)`, `JA(NNZ)`, `IA(N+1)`, where `NNZ` is the number of nonzeros in the matrix, and `N` is the order of the matrix. (The sparse skyline format is this structure combined with the matrix U stored in compressed sparse column (CSC) format, with the diagonal stored separately; alternatively, a single `A` and `JA` may be used.) The linked-list companion structure requires three additional integer arrays: `JSTART(N)`, `LINK(NNZ)`, `JR(NNZ)`. The arrays `A`, `JA`, `LINK` and `JR` are parallel arrays, i.e., the i-th element of each array refer to the same nonzero entry. `JSTART` is an array of integer pointers into the parallel arrays, pointing to the first nonzero entry in each column. `JR` stores the row index of this entry, and `LINK` stores the pointer to the next entry in the column. A value of zero in `LINK` indicates that there is no next entry. When a new row is generated

in L, the new nonzeros are added to the beginning of the linked lists.

The companion structure nearly doubles the storage required for the preconditioner. At least some of this storage needs to be allocated later, if GMRES, for example, is to be used. If memory usage is critical, then the lower triangular matrices may be stored directly in column format, at the cost of memory reallocation when necessary.

4.4.3 Estimating stability

A key advantage of the ILUS factorization is the ease with which the stability of its intermediate L_k and U_k factors may be determined. Since we eventually solve with L and U separately, it is reasonable to estimate $\|L_k^{-1}\|$ and $\|U_k^{-1}\|$ separately. In fact, we do not actually need a very good estimate of the norm, only a rough indication of its size, and whether or not it is growing rapidly as the factorization is progressing. We have found that for the lower triangular factor, the infinity norm bound $\|L_k^{-1}e\|_\infty$, where e is a vector of all ones, is effective for this purpose. In addition, the solution and norm of $L_{k+1}^{-1}e$ may be updated easily: the last component of $L_{k+1}^{-1}e$ can be determined with one sparse SAXPY operation. Unfortunately, this cannot be done for the upper triangular factor. In this case, we estimate the infinity norm of its transpose. Other more complicated condition estimates are possible [22, 57, 77], but we have not found them to be necessary for our purpose.

To illustrate how easily ILUS detects instability, consider the FIDAP07 example problem. Figure 4.2 illustrates the growth in the condition norm bounds as the factorization progresses. In typical operation, the factorization would have been aborted when the bounds exceeded some level.

An interesting way to determine how badly L_k and U_k are conditioned is to examine the residual norm reduction in the approximate inverse iteration. If the residual was reduced by very little, this may indicate that the factors are poorly conditioned. However, this measure is not usually monotone with k like the norm bound above, and is thus difficult to utilize.

When instability has been detected, for example when a norm estimate exceeds some stable norm limit, the ILUS factorization code exits and indicates that the solver should switch to another preconditioner, or restart ILUS allowing more fill-in. This kind of behavior can save much computation time, especially when dealing with new and large matrices.

Instead of exiting, it is tempting to increase the allowed fill-in and continue with the current row. This must be accompanied by an increase in the norm limit, since increasing the number of allowed fill-ins does not guarantee that the norm estimate will decrease, nor that the norm limit will not again be exceeded very soon. We performed several experiments, and unfortunately, they

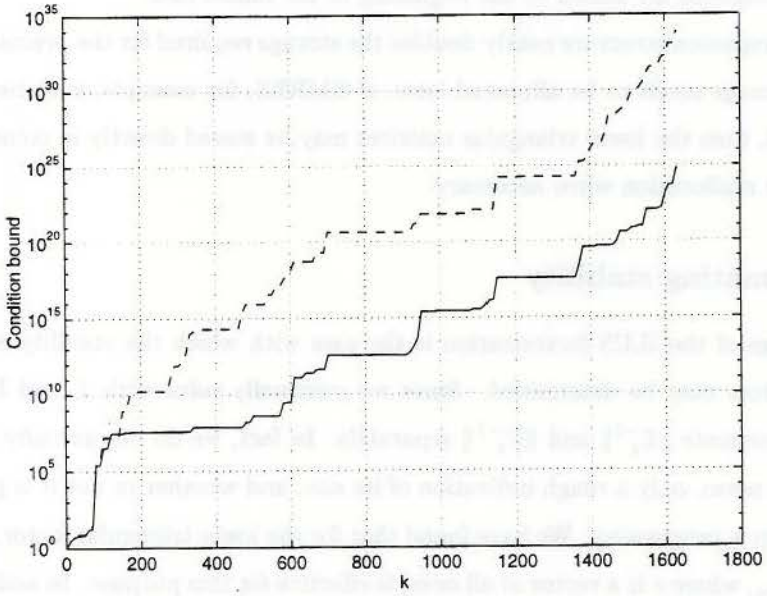


Figure 4.2: Growth in $\|L_k^{-1}e\|_\infty$ (solid line) and $\|U_k^{-T}e\|_\infty$ (dashed line) for FIDAP07.

showed that by the time even mild instability has been detected, the factorization is already too inaccurate to support continuing the factorization with more fill-in.

Another tempting step is to stop the current factorization and proceed with the remaining submatrix. This is a technique of blocking the matrix, while extracting an incomplete factorization for the diagonal blocks. We experimented with this idea and found that a major problem is that many w_k's and v_k's can be zero after the blocking. When combined with an α_{k+1} that is also zero, this leads to singular preconditioner. A complicated reordering would be necessary to solve this problem. Note that this problem may also occur without blocking, but this is far less likely.

It is also possible that a weighted dropping scheme for the elements in z_k be used, for example by using D_k^{-1} or $L_k^{-1}e$ as inverse weights. This allows the stability of the factors to be controlled in some sense. In one extreme, if all elements in z_k must be dropped, this corresponds to breaking the triangular solve recurrence, just as blocking does above.

The ILUS algorithm may be summarized as follows.

ALGORITHM **4.4.1** *ILUS*

1. *Set* $D_1 = a_{11}, L_1 = U_1 = 1$
2. *For* $k = 1, \ldots, n-1$ *do*
3. *Compute a sparse* $z_k \approx D_k^{-1} L_k^{-1} v_k$
4. *Compute a sparse* $y_k \approx w_k U_k^{-1} D_k^{-1}$
5. *Compute* $d_{k+1} := \alpha_{k+1} - y_k D_k z_k$
6. *Estimate* $\|L^{-1}\|$ *and* $\|U^{-1}\|$ *and exit if either exceeds some limit*
7. *EndDo*

4.4.4 Test results

We tested ILUS by using it as a preconditioner for GMRES for solving the linear systems arising from the driven cavity problem (see Section 1.2). Table 4.5 reports the number of GMRES iterations required for each linear system, and the computation time on one processor of a Cray C90 supercomputer in 64-bit arithmetic. The computation time is divided into the time to compute the ILUS preconditioner, and the time required by the GMRES iterations. The approximate inverse procedure was used to compute the ILUS factorization. This procedure also used GMRES, starting with the right-hand side of the linear systems as the initial guess. Three iterations were used, without dropping between steps, the parameter *lfil* was set at 40, and no drop tolerance was used.

For comparison with direct methods, at Reynolds number 1000, ILUS produced a preconditioner with 1,403,370 total nonzeros, while the frontal solver produced an upper triangular factor with 4,327,550 nonzeros, and a lower triangular factor with 4,424,959 nonzeros. Although the direct solver is much faster for these problems, the storage requirement for the iterative solver is much less, even including the companion structure. The advantage of iterative methods for problems of this size is storage, rather than time. For larger problems, iterative methods may also be advantageous in terms of time.

Figure 4.3 illustrates the increase in the condition norm bound for the L factor at Reynolds number 0, on a small 20 by 20 mesh ($n = 4562$, *lfil* $= 30$, all other parameters the same). For comparison, the growth of this bound is illustrated for the same, but *unscaled* matrix. The unscaled problem could not be solved with GMRES and this preconditioner.

4.5 Block tridiagonal incomplete factorizations

In Chapter 3, approximate inverses were used as preconditioners for the entire matrix. Approximate inverse solutions can be expensive to compute, however, for large and difficult problems. In addition,

Re.	GMRES iterations	CPU time (s) precon	solve	total
0	68	135.3	7.1	142.4
100	93	131.9	9.4	141.3
200	209	133.2	21.4	154.6
300	189	130.7	19.1	149.8
400	145	132.3	14.8	147.2
500	222	130.5	22.5	153.0
600	235	132.8	24.0	156.9
700	258	131.5	26.1	157.6
800	147	134.1	15.0	149.1
900	264	132.8	26.8	159.6
1000	391	135.4	39.9	175.4

Table 4.5: Test results for the driven cavity problems.

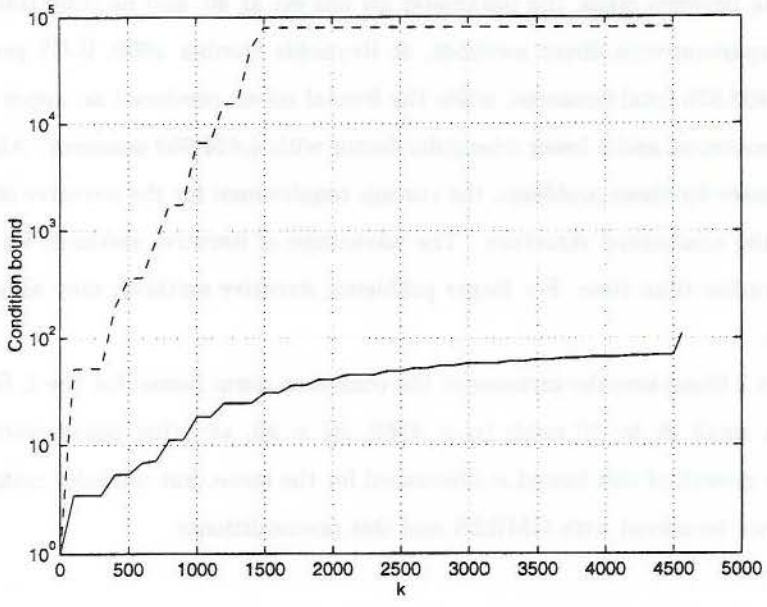


Figure 4.3: Growth in condition norm bounds at Re. 0, Scaled (solid line), Unscaled (dashed line).

the local nature of sparse approximate inverses suggests that their use could be more effective for smaller problems. In this and the remaining sections of this chapter, we examine matrices partitioned into blocks (i.e., smaller problems) as

$$A = \begin{pmatrix} B & F \\ E & C \end{pmatrix}. \tag{4.13}$$

Matrices of this form arise in many applications, such as in the incompressible Navier-Stokes equations described by (1.7), or from the domain decomposition reordering described by (2.20). Some segregated, or block-oriented techniques such as Uzawa's method were also described in Section 1.1.

Typically, the linear systems associated with the B matrix produced by domain decomposition reordering are easy to solve, being the result of restricting the original PDE problem into a set of independent and similar PDE problems on much smaller meshes. One of the motivations for this approach is parallelism. This approach ultimately requires solution methods for the Schur complement, $S = C - EB^{-1}F$.

If it is easy to solve with B, then one solution method is to solve the Schur complement system with the matrix $S = C - EB^{-1}F$ with an iterative method and preconditioner. In fact, if B is chosen to be diagonal, it may be advantageous to construct S exactly before applying a preconditioning technique to S [100].

If it is difficult to solve with B, then iterating with the matrix S will be expensive, since one must solve with it accurately. As an alternative, a fully-coupled type of preconditioner (i.e., involving all the variables) can be constructed, for example the factorization

$$\begin{pmatrix} B & 0 \\ E & M_S \end{pmatrix} \begin{pmatrix} I & B^{-1}F \\ 0 & I \end{pmatrix} \tag{4.14}$$

where M_S is an approximation to S. Approximate inverse techniques may be used for B^{-1} (or $B^{-1}F$) and M_S^{-1} when they are required. The former is essential if a sparse approximation to S is constructed. This type of framework was presented in [43] and [79].

The "globally-coupled local inverse" preconditioner [139] also uses the same approach, but chooses a specific block partitioning that is effective with approximate inverses. From a discretization of a PDE, the nodes corresponding to the equations in S are distributed uniformly in the domain. This has the effect of coupling the approximate inverses used to approximate B^{-1} to try to regain the low-frequency information in A. The distribution of the nodes in S is balanced with the local

extent of the approximate inverse for B. This is related to the early use of approximate inverses as multigrid smoothers [72, 15], i.e., approximate inverses at different levels.

A specific example of (4.14) is the incomplete factorization of block tridiagonal matrices, also known as "nested factorization" since it can be viewed as a recursive version of (4.14). These factorizations have been studied extensively in the past decade [5, 6, 10, 11, 49, 97, 99], but were not usually applied to general sparse matrices since general techniques for approximating the inverse of sparse pivot blocks were not available. The remainder of this section will examine these factorizations with general approximate inverse techniques.

Consider the matrix A in block tridiagonal form

$$A = \begin{pmatrix} B_1 & F_2 & & & \\ E_2 & B_2 & F_3 & & \\ & \ddots & \ddots & \ddots & \\ & & E_{m-1} & B_{m-1} & F_m \\ & & & E_m & B_m \end{pmatrix}$$

and its *complete* block LU factorization

$$(E + D)D^{-1}(D + F) \tag{4.15}$$

where E and F are the block strictly lower- and upper-triangular parts of A, and D is a block-diagonal matrix whose blocks D_i are defined by the recurrence

$$D_i = B_i - E_i D_{i-1}^{-1} F_i \tag{4.16}$$

for $i = 2, \ldots, m$ and with $D_1 = B_1$.

To make the factorization (4.15) *incomplete*, the exact inverse D_{i-1}^{-1} in (4.16) is replaced with an approximate inverse, i.e.,

$$D_i = B_i - E_i \cdot \text{apinv}(D_{i-1}) \cdot F_i. \tag{4.17}$$

Notice that the result D_i of this operation should somehow remain sparse. To study the possible options, it is useful to view "apinv" as a composite operation. This will also aid the design of modular software.

First, D_{i-1} may not be in a form suitable for computing an approximate inverse. For example, diagonal compensation or a sparsification of D_{i-1} may be necessary for the approximate inverse to exist or be economical to compute. Thus we can separate out a preprocessing step called "approx1"

$$D_i = B_i - E_i \cdot \text{apinv}(\text{approx1}(D_{i-1})) \cdot F_i. \tag{4.18}$$

Second, the matrix D_i above should be relatively sparse and not be expensive to form. This can be done in a variety of ways. In some cases, the approximate inverse is sparse or can be sparsified in such a way that $\text{apinv}(\text{approx1}(D_{i-1})) \cdot F_i$ is also sparse. To be general, we can write

$$D_i = B_i - E_i \cdot \text{approx2}(\text{apinv}(\text{approx1}(D_{i-1}))) \cdot F_i \tag{4.19}$$

where "approx2" sparsifies the approximate inverse if necessary.

Another alternative is to write

$$D_i = B_i - E_i \cdot \text{apinv2}(\text{approx1}(D_{i-1}), F_i) \tag{4.20}$$

where the "apinv2" function here gives a sparse approximation to $\text{approx1}(D_{i-1})^{-1} \cdot F_i$. (A function that operates with E_i can also be defined.)

The "apinv" function is a sparse approximate inverse, possibly in factored form. If the structures of all the B_i are the same, then it may be desirable that the sequence of approximate inverses in (4.17) have the same pattern. This may be done by choosing "approx1" to restrict the pattern of D_i to this structure, for example. The common example of this is when the B_i are tridiagonal, and the E_i and F_i are diagonal.

The operation "apinv2" may be the result of

$$\min_X \|\text{approx1}(D_{i-1})X - F_i\|_F$$

for some sparsity pattern for X. The "apinv2" function can also be a factorization, which can be economical if $\text{approx1}(D_{i-1})$ is narrowly banded.

The existence of block incomplete factorizations has been proven for M-, H-, and block H-matrices, with certain assumptions on the sparse approximations to the pivots blocks and their inverses. For block tridiagonal matrices (i.e., block factorizations of generalized SSOR type), existence has been proven for positive, nonsymmetric matrices, again with certain assumptions.

We illustrate the use of approximate inverses in these factorizations with the FIDAP19 example problem, the largest nonsymmetric matrix in the FIDAP collection ($n = 12005$, $nnz = 259879$). The problem is an axisymmetric 2D developing pipe flow, using the two-equation k-ϵ model for turbulence. A constant block size of 161 was used, the smallest block size that would yield a block tridiagonal system (the last block has size 91). Since the matrix arises from a finite element problem, a more careful selection of the partitioning may yield better results. In the worse case, a pivot block may be singular; this would cause difficulties for several approximate inverse techniques such as [98] if the sparsity pattern is not augmented. In our case, a minimal residual solution in the null space would be returned. Our experiments did not perform the "approx" modifications described above.

Since the matrix contains different equations and variables, the rows of the system were scaled by their 2-norms, and then their columns were scaled similarly. A Krylov subspace size for GMRES of 50 was used. For comparison, Table 4.6 first illustrates the solution with the BILU(0)–SVD(α) method (described in Section 3.10) with a block size of 5. The infinity-norm condition of the inverse of the block LU factors is estimated with $\|(LU)^{-1}e\|_\infty$, where e is the vector of all ones. This condition estimate decreases dramatically as the perturbation is increased.

α	condition estimate	GMRES steps
0.000	1.e46	†
0.001	7.e47	†
0.010	8.e26	†
0.050	3.e08	†
0.100	3.e05	†
0.500	129.	87
1.000	96.	337

Table 4.6: FIDAP19, BILU(0)–SVD(α).

Table 4.7 shows the condition estimate, number of GMRES steps to convergence, timings for setting up the preconditioner and the iterations, and the number of nonzeros in the preconditioner. The method BTIF denotes block tridiagonal incomplete factorization and may be used with several approximate inverse techniques. MR-s(*lfil*) and MR-r(*lfil*) denote the minimal residual algorithm using dropping in the solution and residual vectors, respectively, and LS is the least-squares solution using the sparsity pattern of the pivot block as the sparsity pattern of the approximate inverse. The MR methods used *lfil* of 10, and specifically, 3 outer and 1 inner iteration for MR-s, and *lfil* iterations

for MR-r. Self-preconditioning was used, and the initial guess for the approximate inverse was a scaled transpose of the original pivot block. LS used the DGELS routine in LAPACK to compute the least-squares solution. The experiments were carried out on one processor of a Sun Sparcstation 10. The code for constructing the incomplete block factorization is somewhat inefficient in two ways: it transposes the data structure of the pivot block and the inverse (to use column-oriented algorithms), and it counts the number of nonzeros in the sparse matrix-matrix multiplication before performing the actual multiplication.

	cond.	GMRES	CPU time (s)			nonzeros
	est.	steps	precon	solve	total	precon
BILU(0)–SVD(0.5)	129.	87	15.98	143.18	159.16	983 875
BTIF–MR-s(10)	119.	186	56.20	113.41	169.61	120 050
BTIF–MR-r(10)	92.	239	77.85	142.80	220.65	120 050
BTIF–MR-s(5)	382.	328	44.58	186.34	230.92	60 025
BTIF–MR-r(5)	93.	527	34.86	295.51	330.37	60 025
BTIF–LS	5.e95	†	290.02	†	†	453 605

Table 4.7: FIDAP19, block tridiagonal incomplete factorization.

The timings show that BTIF–MR-s(10) is comparable to BILU(0)–SVD(0.5) but uses much less memory. Although the actual number of nonzeros in the matrix is 259,879, there were 39,355 block nonzeros required in BILU(0), and therefore almost a million entries that needed to be stored. BILU(0) required more time in the iterations because the preconditioner was denser, and needed to operate with much smaller blocks. The MR methods produced approximate inverses that were sparser than the original pivot blocks. The LS method produces approximate inverses with the same number of nonzeros as the pivot blocks, and thus required greater storage and computation time. The solution was poor, however, possibly because the second, third, and fourth pivot blocks were poorly approximated.

4.6 Preconditioners for block-partitioned matrices

This chapter continues to explore some preconditioning options when the matrix is expressed in the block-partitioned form (4.13). The preconditioners are defined from some block-partitioned factorization. The iterative method acts on the fully-coupled system, but the preconditioning has some similarity to segregated methods. This approach only requires preconditioning or approximate solves with submatrices, where the submatrices correspond to a Schur complement, or some combination

of operators, such as reaction, diffusion, and convection. It is particularly advantageous to use the block-partitioned form if one knows enough about the submatrices to apply specialized preconditioners, for example operator-splitting and semi-discretization, as well as lower-order discretizations.

These block-partitioned preconditioners require the approximate inverse technique: the solutions need to be sparse because they form the rows or columns of the preconditioner, or are used in further computations. Dense solutions here will cause the construction or the application of the preconditioner to be too expensive.

Consider a sparse linear system

$$Au = b \tag{4.21}$$

which is put in the block form,

$$\begin{pmatrix} B & F \\ E & C \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix}. \tag{4.22}$$

For now the only condition we require on this partitioning is that B be nonsingular. We use extensively the following block LU factorization of A,

$$\begin{pmatrix} B & F \\ E & C \end{pmatrix} = \begin{pmatrix} B & 0 \\ E & S \end{pmatrix} \begin{pmatrix} I & B^{-1}F \\ 0 & I \end{pmatrix} \tag{4.23}$$

in which S is the Schur complement,

$$S = C - EB^{-1}F. \tag{4.24}$$

As is well-known, we can solve (4.23) by solving the reduced system,

$$Sy = g' \quad \text{with} \quad g' = g - EB^{-1}f \tag{4.25}$$

to compute y, and then back-substitute in the first block row of the system (4.22) to obtain x, i.e., compute x by

$$x = B^{-1}(f - Fy) .$$

The above block structure can be exploited in several different ways to define preconditioners for A. Thus, the preconditioners proposed here combine a standard block preconditioner with one of the preconditioners for S, to be described in Section 4.6.5. First, we outline a few such options.

4.6.1 Solving the preconditioned reduced system

A method that is often used is to solve the reduced system (4.25), possibly with the help of a certain preconditioner M_S for the Schur complement matrix S. Although this does not involve any of the block factorizations discussed above, it is indirectly related to it and to other well-known algorithms. For example, the Uzawa method, which is typically formulated on the full system, can be viewed as a Richardson (or fixed point) iteration applied to the reduced system. The matrix S need not be computed explicitly—instead, one can perform the matrix-vector product $w = Sv$ with the matrix S via the following sequence of operations:

1. Compute $t := Fv$;
2. Solve $Bu = t$;
3. Compute $w := Cv - Eu$.

If we wish to use a Krylov subspace technique such as GMRES on the preconditioned reduced system, we need to solve the systems in step 2, exactly, i.e., by a direct solver or an iterative solver requiring a high accuracy. This is because the S matrix is the coefficient matrix of the system to be solved, and it must be constant throughout the GMRES iteration. We have experimented with this approach and found that this is a serious limitation. Convergence is reached in a number of steps which is typically comparable with that obtained with methods based on the full matrix. However, each step costs much more, unless a direct solution technique is used, in which case the initial LU factorization may be very expensive. Alternatively, a highly accurate ILU factorization can be employed for B, to reduce the cost of the many systems that must be solved with it in the successive outer steps.

4.6.2 Approximate block-diagonal preconditioner

One of the simplest block preconditioners for a matrix A partitioned as in (4.22) is the block-diagonal matrix

$$M = \begin{pmatrix} B & 0 \\ 0 & M_C \end{pmatrix} \tag{4.26}$$

in which M_C is some preconditioning for the matrix C. If $C = 0$ as is the case for the incompressible Navier-Stokes equations, then we can define $M_C = I$ for example. An interesting particular case is when C is nonsingular and $M_C = C$. This corresponds to a block Jacobi iteration. In this case, we

have

$$I - M^{-1}A = \begin{pmatrix} 0 & B^{-1}F \\ C^{-1}E & 0 \end{pmatrix}$$

the eigenvalues of which are the square roots of the eigenvalues of the matrix $C^{-1}EB^{-1}F$. Convergence will be fast if all these eigenvalues are small. Careful reordering and partitioning of the matrix will improve the performance of the preconditioner in various cases, see for example [1, 62].

4.6.3 Approximate block LU factorization

The block factorization (4.23) suggests using preconditioners based on the block LU factorization

$$M = LU$$

in which

$$L = \begin{pmatrix} B & 0 \\ E & M_S \end{pmatrix} \quad \text{and} \quad U = \begin{pmatrix} I & B^{-1}F \\ 0 & I \end{pmatrix}$$

to precondition A. Here M_S is some preconditioner to the Schur complement matrix S. If we had a sparse approximation $\tilde{S}$ to the Schur complement S we could compute a preconditioning matrix M_S to $\tilde{S}$, for example, in the form of an approximate LU factorization. We must point out here that any preconditioner for S will induce a preconditioner for A. As was discussed in Section 4.6.1 a notable disadvantage of an approach based on solving the reduced system (4.25) by an iterative process is that the action of S on a vector must be computed very accurately in the Krylov acceleration part. In an approach based on the larger system (4.22) this is not necessary. In fact any iterative process can be used for solving with M_S and B provided we use a flexible variant of GMRES such as FGMRES [125].

Systems involving B may be solved in many ways, depending on their difficulty and what we know about B. If B is known to be well-conditioned, then triangular solves with incomplete LU factors may be sufficient. For more difficult B matrices, the incomplete factors may be used as a preconditioner for an inner iterative process for B. Further, if the incomplete factors are unstable, an approximate inverse for B may be used, either directly or as a preconditioner. If B is an operator, an approximation to it may be used; its factors may again be used either directly or as a preconditioner. This kind of flexibility is typical of what is available for using iterative methods on block-partitioned matrices.

An important observation is that if we solve exactly with B then the error in this block ILU

factorization lies entirely in the (2,2) block since,

$$A = LU + \begin{pmatrix} 0 & 0 \\ 0 & S - M_S \end{pmatrix}. \tag{4.27}$$

One can raise the question as to whether this approach is any better than one based on solving the reduced system (4.25) preconditioned with M_S. It is known that in fact the two approaches are mathematically equivalent if we start with the proper initial guesses. Specifically, the initial guess should make the x-part of the residual vector equal to 0 for the original system (4.22), i.e., the initial guess is

$$u_0 = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \quad \text{with} \quad x_0 = B^{-1}(f - Fy_0) \ .$$

This result, due to Eisenstat and reported in [93], immediately follows from (4.27) which shows that the preconditioned matrix has the particular form,

$$(LU)^{-1}A = \begin{pmatrix} I & 0 \\ 0 & M_S^{-1}S \end{pmatrix}. \tag{4.28}$$

Thus, if the initial residual has its x-component equal to zero then all iterates will be vectors with y components only, and a GMRES iteration on the system will reduce to a GMRES iteration with the matrix $M_S^{-1}S$ involving only the y variable.

There are many possible options for choosing the matrix M_S. Among these we consider the following ones.

1. $M_S = I$ – no preconditioning on S.
2. $M_S = C$ – precondition with the C matrix if it is nonsingular. Alternatively we can precondition with an ILU factorization of C.
3. $M_S \approx S$ – construct a sparse approximation to S and use it as a preconditioner. In general, we only need to approximate the action of S on a vector.

The following algorithm applies one preconditioning step to $\begin{pmatrix} f \\ g \end{pmatrix}$ to get $\begin{pmatrix} x \\ y \end{pmatrix}$.

ALGORITHM **4.6.1** *Approximate block LU preconditioning*

1. $x := B^{-1}f$
2. $y := M_S^{-1}(g - Ex)$
3. $x := x - B^{-1}Fy$

We have experimented with a number of options for solving systems with M_S in step 2 of the algorithm above. For example, M_S may be approximated with $\tilde{S} = C - EY$, where $Y \approx B^{-1}F$ is computed by the approximate inverse technique. If this approximation is used, it is possible to also replace $B^{-1}F$ by Y in step 3.

4.6.4 Approximate block Gauss-Seidel

By ignoring the U factor of the approximate block LU factorization, we are led to a form of block Gauss-Seidel preconditioning, defined by $M = L$, i.e.,

$$M = \begin{pmatrix} B & 0 \\ E & M_S \end{pmatrix}. \tag{4.29}$$

The same remarks on the ways to solve systems with B and ways to define the preconditioning matrix M_S apply here. The algorithm for this preconditioner is the same as Algorithm 4.6.1 without step 3.

4.6.5 Sparse solutions with the Schur complement

In the previous subsections, sparse approximate solutions with the Schur complement $S = C - EB^{-1}F$ were required in the preconditioning for block-partitioned matrices. We will briefly describe three approaches in this section: (1) approximating S, (2) approximating S^{-1}, and (3) exploiting a partial approximate inverse of A.

To approximate S with a sparse matrix, we can use

$$\tilde{S} = C - EY, \quad Y \approx B^{-1}F, \tag{4.30}$$

where Y is computed by the approximate inverse technique, possibly preconditioned with whatever we are using to solve with B. Since Y is sparse, $\tilde{S}$ computed this way is also sparse. Moreover, since S is usually relatively dense, solving with $\tilde{S}$ is an economical approach. Typically, a zero initial guess is used for Y. We remark that it is usually too expensive to form Y by solving $B^{-1}F$ approximately and then dropping small elements, since it is rather costly to search for elements to drop. We also note that we can generate $\tilde{S}$ column-by-column, and if necessary, compute a factorization of $\tilde{S}$ on a column-by-column basis as well. The linear systems with $\tilde{S}$ can be solved in any fashion, including with an iterative process with or without preconditioning.

Another method is to compute an approximation to S^{-1} using the idea of induced preconditioning. Since S^{-1} is the (2,2) block of

$$\begin{pmatrix} B & F \\ E & C \end{pmatrix}^{-1} = \begin{pmatrix} B^{-1} + B^{-1}FS^{-1}EB^{-1} & -B^{-1}FS^{-1} \\ -S^{-1}EB^{-1} & S^{-1} \end{pmatrix} \tag{4.31}$$

we can compute a sparse approximation to it by using the approximate inverse technique applied to the last block column of A and then throwing away the upper block. In practice, the upper part of each column may be discarded before computing the next column. In our experiments, since the approximate inverse algorithm is applied to A, an indefinite matrix in most of the problems, the normal equations search direction $A^T r$ is used in the algorithm, with a scaled identity initial guess for the inverse.

A drawback of the above approach is that the top submatrix of the last block column is discarded, and that the resulting approximation of S^{-1} may actually contain very few nonzeros. A related technique is to compute the partial approximate inverse of A in the last block row. This technique does not give an approximation to S^{-1}, but defines a simple preconditioning method itself, called the *partial approximate inverse* method. Writing the inverse of A in the form,

$$\begin{pmatrix} B & F \\ E & C \end{pmatrix}^{-1} \approx \begin{pmatrix} M_1 \\ M_2 \end{pmatrix} \tag{4.32}$$

we can then get an approximate solution to $A\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix}$ with

$$\begin{aligned} y &= M_2 \begin{pmatrix} f \\ g \end{pmatrix} \\ x &= B^{-1}(f - Fy). \end{aligned} \tag{4.33}$$

It is not necessary to solve accurately with B. Again, the normal equations search direction is used for the approximate inverse algorithm in the numerical experiments.

4.7 Test results for block-partitioned preconditioners

We will use the following names to denote the methods that we tested. We test NOPRE, ILUT, and ILUTP for comparison purposes.

NOPRE No preconditioner.

ILUT(*nfil*) and **ILUTP**(*nfil*) Incomplete LU factorization with threshold of *nfil* nonzeros per row in each of the L and U factors. These preconditioners were described in Section 2.4.2.

PAR(*lfil*) Partial approximate inverse preconditioner, using *lfil* nonzeros per row in M_2.

ABJ Approximate block Jacobi preconditioner. This preconditioner only applies when $C \neq 0$.

ABLU(*lfil*) Approximate block LU factorization preconditioner. The approximation (4.30) to S with *lfil* nonzeros per column of Y was used.

ABLU_y(*lfil*) Same as above, but using Y whenever $B^{-1}F$ needs to be applied in step 3 of Algorithm 4.6.1.

ABLU_s(*lfil*) Approximate block LU factorization preconditioner, using (4.31) to approximate S^{-1} with *lfil* nonzeros per column when approximating the last block column of the inverse of A.

ABGS(*lfil*) Approximate block Gauss-Seidel preconditioner. The approximation (4.30) to S with *lfil* nonzeros per column of Y was used.

The storage requirements for each preconditioner are given in Table 4.8. The ILU preconditioners require considerably more storage than the approximate block-partitioned factorizations, since their storage depends on n rather than n_C. Because the approximation to S^{-1} discards the upper block, the storage for it is less than $lfil \times n_C$. The storage required for $\tilde{S}$ is more difficult to estimate since it is at least the product of two sparse matrices. It is generally less than $2 \times lfil \times n_C$; Table 4.16 gives the exact number of nonzeros in $\tilde{S}$ for the FIDAP problems.

	Matrices	Matrix locations
ILUT(*nfil*)	L, U	$2 \times nfil \times n$
PAR(*lfil*)	M_2	$lfil \times n_C$
ABJ	none	none
ABLU(*lfil*)	$\tilde{S}$	less than $2 \times lfil \times n_C$
ABLU_y(*lfil*)	$\tilde{S}, Y$	less than $3 \times lfil \times n_C$
ABLU_s(*lfil*)	approx S^{-1}	less than $lfil \times n_C$
ABGS(*lfil*)	$\tilde{S}$	less than $2 \times lfil \times n_C$

Table 4.8: Storage requirements for each preconditioner.

The first set of test problems is a finite difference Laplace equation with Dirichlet boundary conditions. Three different sized grids were used. The matrices were reordered using a domain

decomposition reordering with 4 subdomains. In the following tables, n is the order of the matrix, nnz is the number of nonzero entries, n_B is the order of the B submatrix, and n_C is the order of the C submatrix.

Grid	n	nnz	n_B	n_C
32 by 32	961	4681	900	61
48 by 48	2209	10857	2116	93
64 by 64	3969	19593	3844	125

Table 4.9: Laplacian test problems.

The second set of test matrices were extracted from the example incompressible Navier-Stokes problems in the FIDAP [67] package. All problems with zero C submatrix were tested. The matrices are reordered so that the continuity equations are ordered last. The scaling of many of the matrices is poor, since each matrix contains different types of equations. Thus, we scale each row to have unit 2-norm, and then scale each column the same way. The problems are all originally nonsymmetric except 4, 12, 14 and 32.

Matrix	n	nnz	n_B	n_C	
FIDAP04	1601	31850	1151	450	Hamel flow
FIDAP06	1651	49063	1180	471	Die swell
FIDAP12	3973	79078	2839	1134	Stokes flow
FIDAP14	3251	65875	2351	920	Isothermal seepage
FIDAP20	2203	67830	1603	600	Surface disturbance attenuation
FIDAP23	1409	42761	1008	401	Fountain flow
FIDAP24	2283	47901	1635	648	Forward roll coating
FIDAP26	2163	74465	1706	457	Driven thermal convection
FIDAP28	2603	77031	1853	750	Two merging liquids
FIDAP31	3909	91223	3279	630	Dilute species deposition
FIDAP32	1159	11047	863	296	Radiation heat transfer
FIDAP36	3079	53099	2575	504	Chemical vapor deposition
FIDAP40	7740	456189	5916	1824	3D Die swell

Table 4.10: FIDAP example matrices.

The third set of test problems is from the driven cavity problem. We will show our results for problems with Reynolds number 0, 500, and 1000. All matrices arise from a mesh of 20 by 20 elements, leading to matrices of size $n = 4,562$ and having nnz =138,187 nonzero entries. These matrices have 3,363 velocity unknowns, and 1,199 pressure unknowns. The matrices are scaled the same way as for the FIDAP matrices—the problems are otherwise very difficult to solve.

Linear systems were constructed so that the solution is a vector of all ones. A zero initial guess for right-preconditioned FGMRES [125] restarted every 20 iterations was used to solve the systems. The tables show the number of iterations required to reduce the residual norm by 10^{-7}. The iterations were stopped when 300 matrix-vector multiplications were reached, indicated by a dagger (†). The codes were written in Fortran 77 and run in single precision on a Cray C90 supercomputer.

For comparison purposes, we first perform some preliminary tests using the domain decomposition reordering on the driven cavity problems. This technique is also useful if B is highly indefinite and produces an unstable LU factorization; by limiting the size of the factorization, the instability cannot grow beyond a point for which the factorization is not useful.

In Table 4.11 we show some results of ILUT(40) on the driven cavity problem with different matrix reorderings. We used the original unblocked ordering where the degrees of freedom of the elements are ordered together, the blocked ordering where the continuity equations are ordered last, and a domain decomposition reordering found using a simple automatic recursive dissection procedure with four subdomains. This latter ordering found 3680 nodes internal to the subdomains, and 882 interface nodes.

Re.	Unblocked	Blocked	DD ordered
0	24	48	60
500	27	†	51
1000	78	†	51

Table 4.11: Effect of ordering on iteration count for ILUT-preconditioned driven cavity problems.

The poorer quality of the incomplete factorization for the driven cavity problems in block-partitioned form is due to the poor ordering rather than instability of the L and U factors. For the problem with Reynolds number 0, the unblocked format produces 745,187 nonzeros in the strictly lower-triangular part during the incomplete factorization (which is then dropped down to less than $n \times nfil = 182{,}480$ nonzeros) while the block-partitioned format produces 2,195,688 nonzeros, almost three times more.

The factorization for the domain decomposition reordered matrices encounters many zero pivots when it reaches the (2,2) block. These latter orderings do not necessarily cause ILUT to fill-in zeros on the diagonal. Nevertheless, the substitution of a small pivot described above seems to be effective here. The domain decomposition reordering also reduces the amount of fill-in because of the shape of the matrix (a downward pointing arrow). Combined with its tendency to limit the growth of instability, the results show this reordering is advantageous even on serial computers.

In Table 4.12 we compare the difficulty of solving the B and $\tilde{S}$ subsystems for the blocked and domain decomposition reorderings of the driven cavity problems. $\tilde{S}$ was computed as $\tilde{S} = C - EY$, where Y was computed using the approximate inverse technique with *lfil* of 30. Here we used ILUT(30) and only solved the linear systems to a tolerance of 10^{-5}. Solves with these submatrices in the block-partitioned preconditioners usually need to be much less accurate. In most of the experiments that follow, we used unpreconditioned iterations to a tolerance of 10^{-1} or 100 matrix-vector multiplications to solve with B and $\tilde{S}$. Other methods would be necessary depending on the difficulty of the problems. The table gives an idea of how difficult it is to solve with B and $\tilde{S}$, and again shows the advantage of using domain decomposition reorderings for hard problems.

	Blocked		DD ordered	
Re.	B	$\tilde{S}$	B	$\tilde{S}$
0	6	3	9	†
500	180	4	7	26
1000	†	†	7	45

Table 4.12: Iteration counts for solving with B and $\tilde{S}$ with different orderings of A.

In Tables 4.13 and 4.14 we present the results for the Laplacian problem with three different grid sizes, using no preconditioning, approximate block-diagonal, partial approximate inverse, approximate block LU, and approximate block Gauss-Seidel preconditioners. Note that in Table 4.14, an *lfil* of zero for the approximate block LU and Gauss-Seidel preconditioners respectively indicate the preconditioners

$$M = \begin{pmatrix} B & 0 \\ E & C \end{pmatrix} \begin{pmatrix} I & B^{-1}F \\ 0 & I \end{pmatrix} \quad \text{and} \quad M = \begin{pmatrix} B & 0 \\ E & C \end{pmatrix}. \tag{4.34}$$

Grid	NOPRE	ABJ	PAR			
			5	10	15	20
32 by 32	135	33	21	18	16	15
48 by 48	367	50	29	21	19	17
64 by 64	532	57	36	33	25	20

Table 4.13: Iterations to convergence for the Laplacian problem.

Now we present test results for the FIDAP problems. For the block-partitioned factorization preconditioners, unpreconditioned GMRES, restarted every 20 iterations, was used to approximately

Grid	ABLU					ABGS				
	0	5	10	15	20	0	5	10	15	20
32 by 32	23	17	15	15	15	15	17	15	15	15
48 by 48	17	18	16	15	15	18	19	19	18	18
64 by 64	19	20	18	18	17	20	23	21	20	20

Table 4.14: Iterations to convergence for the Laplacian problem.

solve the inner systems involving B and $\tilde{S}$ by reducing the initial residual norm by a factor of 0.1, or using up to 100 matrix-vector multiplications. Solves with the matrix B are usually not too difficult because for most problems, B is positive definite. A zero initial guess was used for these solves. The results with various options for the preconditioners are shown in Table 4.15. The best preconditioner appears to be ABLU_y; using Y for $B^{-1}F$ is better than solving a system with B very inaccurately. The number of nonzeros in $\tilde{S}$ is small, as illustrated by Table 4.16 for two values of *lfil*.

	PAR	ABLU_s	ABLU		ABLU_y		ABGS	
Matrix	20	5	20	40	20	40	20	40
FIDAP04	77	78	†	91	227	100	†	126
FIDAP06	†	†	†	†	†	†	†	†
FIDAP12	61	72	53	36	48	34	61	41
FIDAP14	†	†	75	35	83	40	72	48
FIDAP20	141	†	†	60	246	90	†	178
FIDAP23	†	90	†	†	†	269	†	†
FIDAP24	†	†	†	93	†	91	207	136
FIDAP26	†	117	†	91	104	63	209	163
FIDAP28	289	†	214	67	105	67	144	104
FIDAP31	101	49	†	†	122	74	120	162
FIDAP32	136	†	38	21	39	29	63	37
FIDAP36	54	70	80	52	47	36	69	54
FIDAP40	205	†	†	96	84	75	119	102

Table 4.15: Iterations to convergence for the FIDAP problems.

The driven cavity problems are much more challenging because the B block is no longer positive definite, and in fact, acquires larger and larger negative eigenvalues as the Reynolds number increases. For these problems, the unpreconditioned GMRES iterations with B performed done to a tolerance of 10^{-3} or a maximum of 100 matrix-vector multiplications. Again, ABLU_y appears to be the best preconditioner. The results are shown in Table 4.17.

For comparison with incomplete factorization preconditioners, we illustrate the difficulty of solving the FIDAP problems with ILUTP. We progressively allowed more fill-in until the problem

	lfil	
Matrix	20	40
FIDAP04	13377	20796
FIDAP06	15077	23533
FIDAP12	30851	48940
FIDAP14	33144	49932
FIDAP20	21759	33119
FIDAP23	12463	19084
FIDAP24	18966	29010
FIDAP26	13395	21468
FIDAP28	25181	38716
FIDAP31	16551	24452
FIDAP32	6775	11390
FIDAP36	13621	21063
FIDAP40	49729	93330

Table 4.16: Number of nonzeros in $\tilde{S}$.

	ABLU		**ABLU_y**		**ABGS**	
Re.	20	40	20	40	20	40
0	62	42	59	42	84	58
500	†	†	182	92	130	103
1000	†	†	164	118	†	†

Table 4.17: Iterations to convergence for the driven cavity problems.

could be solved, incrementing *nfil* in multiples of 10, with no drop tolerance. The results are shown in Table 4.18. For these types of problems, it is typical that very large amounts of fill-in must be used for the factorizations to be successful. An iterative solution was not attempted if the LU condition lower bound was greater than 10^{30}. If a zero pivot must be used, ILUT and ILUTP attempt to complete the factorization by using a small value proportional to the norm of the row. The matrices were taken in their original banded ordering, where the degrees of freedom of a node or element are numbered together.

Matrix	*nfil*
FIDAP04	20
FIDAP06	50
FIDAP12	70
FIDAP14	> 100
FIDAP20	30
FIDAP23	10
FIDAP24	10
FIDAP26	> 100
FIDAP28	20
FIDAP31	10
FIDAP32	> 100
FIDAP36	10
FIDAP40	10

Table 4.18: *nfil* required to solve FIDAP problems with ILUTP.

A distributed parallel implementation of the approximate block LU preconditioner is described in Saad and Sosonkina [131], which also reports some additional results.

In summary, indefinite problems such as these arising from the incompressible Navier-Stokes equations may be very tough for ILU preconditioners. We can solve more problems with the block approach than with a standard ILU factorization. In addition, this is typically achieved with a far smaller memory requirement than ILUT or a direct solver. We are essentially using domain decomposition: the smaller matrices obtained from the block partitioning can be preconditioned with a standard ILUT approach, while the larger matrices use a block ILU method, and the glue between the two is the preconditioning of the Schur complement.

Chapter 5

Variants of incomplete LU preconditioners

Incomplete LU factorization preconditioners are among the most reliable preconditioners in a general setting. Their failure rate, however, is still too high for them to be useful as black-box library software for general matrices. This chapter attempts to understand the failure modes of ILU preconditioners, particularly for general nonsymmetric and indefinite matrices. Through examples from actual problems, this chapter shows how these problems evince themselves, how these problems can be detected, and how these problems can sometimes be circumvented by certain variants of ILU preconditioners.

5.1 Difficulties in ILU factorizations

The incomplete LU factorization preconditioners were originally developed for M-matrices, for which properties such as existence and a form of stability can be proved [106] (see also [105, 146]). However, ILU preconditioners have been successfully applied in much more general situations. In the general symmetric case, diagonal perturbations of the matrix are required to help guarantee the existence of a symmetric factorization [92, 105, 109]. These perturbations may be applied before the factorization, or during the factorization when a small or negative pivot is encountered. In the nonsymmetric case, there may be another problem: the incomplete factors L and U may be much worsely conditioned than the original matrix A. A coupled effect is that the long recurrences associated with solving with these factors are unstable [32, 65]. This was shown in Figure 1.2 in Section 1.1. A remedy is also to use diagonal perturbations, this time to make the factors diagonally dominant [109, 143, 66], but the perturbations in this case may need to be very large.

ILU preconditioners have also been applied successfully to indefinite matrices, i.e., matrices with indefinite symmetric parts. However, the problems described above can be more severe and more probable:

1. *Inaccuracy due to very small pivots.* Pivots can be arbitrarily small, and often lead to unstable and therefore inaccurate factorizations, i.e., the size of the elements in the factors can grow

uncontrollably, and the factorization becomes inaccurate. By accuracy, we mean the closeness of LU to A. Some pivots, however, particularly near the end of a factorization, may not be used in the factorization, and small values of these pivots have no effect on the stability of the factorization.

2. *Unstable triangular solves.* The incomplete factors of an indefinite matrix are often far from being diagonally dominant, which makes unstable triangular solves more likely. A sign of unstable triangular solves is when $\|L^{-1}\|$ and $\|U^{-1}\|$ are extremely large while the off-diagonal entries of L and U are reasonably bounded. If there are very small pivots, then the triangular solves will be unstable. However, this problem also occurs without the presence of very small pivots.

In *complete* LU factorizations, the main difficulty is the first one: small pivots leading to unstable and inaccurate factorizations. Large elements in the factors directly impact the backward error. The common remedy here is to use a pivoting scheme so that the size of the elements in the factors can be bounded. The second problem of unstable triangular solves is rare for complete factorizations, and thus the problem seems to be related to the effect of dropping nonzeros in incomplete factorizations.

In contrast, for incomplete factorizations, the first problem is much less severe. The growth of the elements in the factors depends on how often each element is updated. In incomplete factorizations, each element is updated far fewer times than in complete factorizations. As long as extremely small pivots are avoided, the growth of the elements in the incomplete factors is not a problem. However, triangular solves can be unstable even though a factorization is stable.

There are two other problems that exist for incomplete factorizations which have not yet been mentioned:

3. *Inaccuracy due to dropping.* Factorizations are made incomplete by dropping nonzeros to make the factorization more economical to store, compute, and solve with. Each nonzero that is dropped contributes to the "error" in the factorization, i.e., contributes to E in the relation $A = LU + E$. However, this error is not a very serious problem as long as accuracy can be improved by allowing more fill-in or using a different dropping scheme or sparsity pattern. If the inaccuracy is not due to dropping, but is due instead to small pivots and an unstable factorization, for example, then simply increasing the allowed fill-in will generally not help.

4. *Zero pivots.* The pivots of an incomplete factorization can be arbitrarily small, even zero. The

most common cause of zero pivots is an irregular structure or ordering of the matrix, one that has a null column above or null row to the left of a zero diagonal element. This is referred to as a *structurally* zero pivot. When a matrix has zeros on the diagonal, this problem can be common unless a careful ordering is used. Zero pivots can also be caused *numerically*, i.e., when a nonzero diagonal element becomes zero. Numerically zero pivots can be caused by very small pivots which cause a row to be "swamped out" by an extremely large factor of the pivotal row.

The above four problems will often occur together, and one problem may mask another. For example, a factorization that is initially inaccurate due to dropping can produce small pivots; these small pivots in turn can make the factorization unstable and even more inaccurate; the inaccuracy may lead to a small pivot which induces a numerically zero pivot. When the factorization fails on the zero pivot, none of the preceding problems may have been noticed.

The four problems may also interact in complex ways that are difficult to predict. For example, by allowing more fill-in to improve the accuracy, the new factorization may happen to have smaller pivots; this in turn may cause the triangular solves to be unstable.

To try to understand what can happen in an incomplete factorization, a number of statistics can be monitored. These statistics, shown in Table 5.1, can be monitored during the course of a factorization, or after the factorization has been computed.

Statistic	Meaning
condest	$\|(LU)^{-1}e\|_\infty, \quad e = (1,\ldots,1)^T$
1/pivot	size of reciprocal of the smallest pivot
max(L+U)	size of largest element in L and U factors

Table 5.1: Statistics that can be used to evaluate an incomplete factorization.

Probably the most useful statistic is *"condest,"* which measures the stability of the triangular solves. It simply measures $\|(LU)^{-1}e\|_\infty$ where e is the vector of all ones. Note that this statistic is also a lower bound for $\|(LU)^{-1}\|_\infty$ and indicates a relation between unstable triangular solves and poorly conditioned L and U factors. We refer to this statistic as the condition estimate of $(LU)^{-1}$. Indeed, we have already introduced this statistic in earlier parts of this thesis.

The second statistic is needed to help interpret this condition estimate. The condition estimate will certainly be poor if there are very small pivots. Thus when *condest* is very large, it should be compared to the size of the reciprocal of the smallest pivot. If these two quantities are

about the same size, then we assume that $\|(LU)^{-1}\|_\infty$ is large due to at least one very small pivot. If *condest* is much larger than *1/pivot* (e.g., *condest* greater than the square of *1/pivot*) then we assume that the recurrences associated with the triangular solves are unstable.

The third statistic is the size of the largest element in the L and U factors. A large value of this statistic in relation to the size of the elements in A indicates an unstable and thus inaccurate factorization. We will see in the numerical experiments that for incomplete factorizations, *max(L+U)* is never large unless *1/pivot* is large. In addition, when *max(L+U)* is large, it is usually about the same size as *1/pivot* (assuming that the maximum entries in A are $O(1)$). Occasionally, we will find very small pivots, but *max(L+U)* remains small. This occurs when the small pivot is not used in the factorization.

Usually, these statistics are only meaningful when their values are very large, e.g., on the order of 10^{15}. Extremely large values, particularly of the condition estimate, can be used to predict when the ILU preconditioner will fail. When all three statistics are reasonably small, and an ILU preconditioner does not help an iterative method converge, it is our experience that the cause of failure is inaccuracy due to dropping.

The chart in Figure 5.1 summarizes some of these statements. Note that there are no cases of small *condest* and large *1/pivot*. Also, although there may be cases when *1/pivot* is very large, as long as *condest* is much larger, we will still label this as an unstable triangular solve.

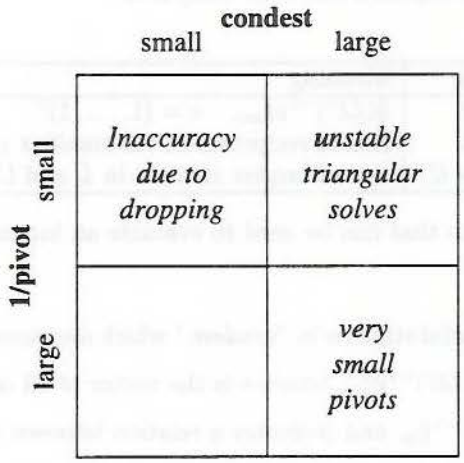


Figure 5.1: How to interpret the statistics.

Recall that the "error" in an incomplete factorization LU of a matrix A is the term E in

$$A = LU + E. \tag{5.1}$$

By only dropping small nonzero entries in L and U, the size of the entries in E can be kept small. This is important because, for symmetric linear systems, the size of E is very strongly related to the convergence rate of an ILU-preconditioned iteration [60].

For nonsymmetric and for indefinite problems, however, the performance is much less predictable. The factorization error is important, but just as important is the stability of the triangular solves, i.e., the norm of the *preconditioned* error $L^{-1}EU^{-1}$ in the preconditioned version of (5.1)

$$L^{-1}AU^{-1} = I + L^{-1}EU^{-1}.$$

When A is indefinite or has a large nonsymmetric part, then L^{-1} and U^{-1} may have very large norms, causing $\|L^{-1}EU^{-1}\|$ to be very large.

For indefinite matrices, the behavior of ILU preconditioners that drop small nonzero entries predicted by the matrix "structure" and methods that drop based on matrix "values" can be very different. Although numerical threshold-based ILU is generally more accurate than level-based ILU, their differences in behavior with respect to other factors must be considered. For example, by systematically retaining the largest elements in L and U in threshold-based ILU, the factorization is more prone to unstable triangular solves, because the off-diagonal elements are generally larger. The largest $(LU)^{-1}$ condition estimates that we see are those produced by threshold-based ILU rather than level-based ILU.

An even more serious problem is the erroneously large entries that may have been computed via a small and inaccurate pivot. In threshold-based ILU, these large entries are propagated during the factorization due to their size. This does not happen when a level-of-fill rule is used.

Practical implementations of threshold-based ILU, such as ILUT [127], include an additional parameter besides the drop tolerance *droptol*, called *lfil*. This is the maximum number of nonzeros allowed in each row of L and U, and allows the storage requirements of the preconditioner to be known beforehand. However, by limiting the fill-in in each row but not each column, a very nonsymmetric preconditioner may be produced. Figure 5.2 illustrates the pattern of a pair of L and U factors together, for a matrix that has a symmetric pattern. The vertical striping in the figure is characteristic of the problem. A consequence is that columns with small elements may never receive

fill-in, and never create fill-in onto a possibly small or zero diagonal element. The bordered form of factorization called ILUS described in Section 4.4 and mentioned again in Section 5.3 circumvents this problem.

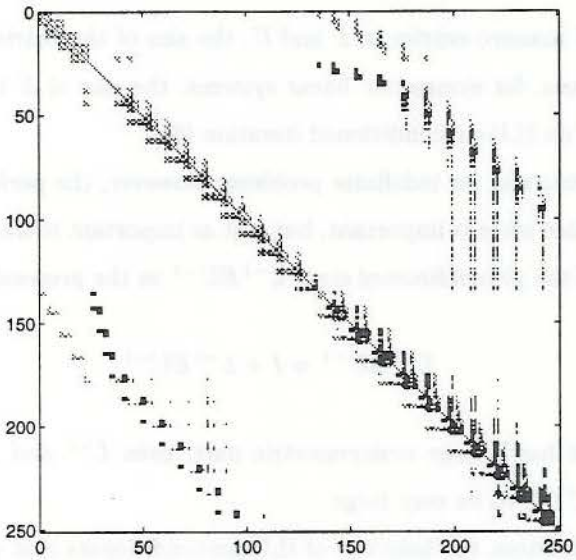


Figure 5.2: Poor ILUT pattern (of $L + U$).

A common cause of this problem is the unequal scaling of the rows or columns of a matrix when there are different types of equations and variables. Thus, when threshold-based ILU preconditioners are used, it is often recommended that matrices are first scaled so that each column has unit 2-norm, and then scaled again so that each row has unit 2-norm. However there are side-effects to this scaling: it may improve the conditioning of the matrix, but it may increase the degree of non-normality of the matrix.

5.2 Pivoting for incomplete factorizations

In *complete* factorizations, pivoting is required for nonsymmetric and indefinite matrices to prevent excessive growth of the entries of the factors. As mentioned in Section 5.1, this type of instability is not a serious problem for incomplete factorizations, and thus pivoting has not generally been used. However, a mechanism to help avoid zero and very small pivots is still needed. Probably the most popular mechanism is to replace these small pivots by a larger value, a technique that we discuss in Section 5.4. This technique, however, may cause too much inaccuracy in the preconditioner,

particularly if many replacements need to be made. An alternative which is particularly suitable for very unstructured matrices with many structurally zero pivots is to use *pivoting*.

The simplest way to incorporate pivoting in an incomplete factorization computed row-wise is to perform column (partial) pivoting. This is because no column data structures are available for the searching required for row pivoting. Referring back to Figure 2.5, in row i of the factorization, after z is computed by (2.18), the column with the largest entry in magnitude in z is exchanged with column i. The entry z_1, which will be the pivot for that row, is the largest entry in z.

In the implementation of column pivoting, no actual column exchanges are made, and the new row indices are determined through permutation vectors. The permutation vectors are updated with each column exchange. This variant of incomplete factorization combined with the dropping strategy of ILUT is called ILUTP. See [123] for more details.

Unlike the case with complete factorizations, pivoting for incomplete factorizations cannot guarantee that a nonzero pivot can always be found, i.e., z may be all zero and failures due to zero pivots can still occur. In fact, a poor pivoting sequence can occasionally trap a factorization into a zero pivot, even if the factorization would have succeeded without pivoting.

A tolerance parameter *permtol* can be included to determine whether or not to perform a permutation. The largest nondiagonal element a_{ij} that satisfies $permtol \times |a_{ij}| > |a_{ii}|$ is permuted into the diagonal position. This type of parameter is used in sparse direct factorization codes to balance stability with the preservation of sparsity.

In *block* incomplete factorizations (BILU), where each entry in Algorithm 2.4.1 is actually a small dense block, a form of pivoting is also occurring. When the inverse of u_{kk} is taken in line 4 of the algorithm, it is assumed to be computed with pivoting if necessary. Thus, one way to deal with zero diagonal entries that might lead to zero pivots is to use *blocking:* guarantee each zero diagonal entry is within a small nonsingular block. ILUTP can be used to simulate this type of pivoting by only searching for pivots within the current block.

The idea of *blocking* is similar to the idea of diagonal pivoting for complete factorizations of symmetric indefinite matrices [34, 58], where 2 by 2 pivot blocks are allowed. Here, permutations are allowed to find a 2 by 2 pivot block that is well conditioned.

5.3 Preserving symmetric structure for threshold-based ILU

For matrices with symmetric structure, threshold-based ILU will not generally produce L and U factors that are symmetric to each other, particularly when the *lfil* parameter is used. However, the symmetric structure can be preserved with an incomplete form of LDU Gaussian elimination based on bordering, namely, ILUS described in Section 4.4. The approximate inverse technique used in that section, however, is not absolutely necessary. Referring to (4.9), recall that the the equations that need to be solved in ILUS are

$$z_k = D_k^{-1} L_k^{-1} v_k \tag{5.2}$$

$$y_k = w_k U_k^{-1} D_k^{-1}. \tag{5.3}$$

Thus, we obtain each row and column of the factorization by approximately solving two lower triangular systems and computing a scaled dot product.

The lower triangular systems (5.2) and (5.3) are solved the same way as system (2.17) was solved, i.e., with numerical dropping. However, in this case, the lower triangular matrices are only available by rows, not by columns. Like before, a companion data structure that gives access to the columns is needed.

In order for z_k and y_k to have the same sparsity pattern, the systems (5.2) and (5.3) are solved simultaneously. Corresponding entries in y_k and z_k are both kept or both dropped, to try to maximize the absolute value of $y_k D_k z_k$. Half of the book-keeping for the sparse computations can be saved because of the symmetric pattern.

Besides preserving symmetric structure, there are several additional advantages to this form of factorization. First, fill-in onto the diagonal is guaranteed as long as all v_k and w_k are nonzero. Second, since L_k and U_k are available after step k, a running condition estimate $\|(L_k U_k)^{-1}\|_\infty$ can be monitored. Third, this form of factorization is amenable to matrices already in skyline format.

5.4 Stabilized ILU

One possibility to affront the problem of small pivots is simply to replace them by larger values. Algorithmically, the new pivots should be chosen large enough to ensure that they do not create extremely large off-diagonal elements. Small pivots can lead to unstable and inaccurate factorizations, and unstable triangular solves. Thus we call such a technique a *stabilized* incomplete factorization.

Stabilization essentially amounts to the factorization of a better conditioned matrix. The relation between accuracy and stability, however, is a delicate one. The trade-off is always between stable factorizations and solves, and a factorization that is accurate or close enough to the original matrix A. It is clear that if the matrix is diagonally dominant, or well-conditioned, stabilization is not necessary, and any modification to the original matrix will cause the factorization to be inaccurate. On the other hand, some matrices will give factorizations that are unstable and therefore inaccurate without stabilization. Stabilization will help here, but too large a stabilization (e.g., too large a diagonal shift) will again cause the factorization to be inaccurate. It is obvious that a successful balance between these two may not always be found, in which case some other technique must be brought into play.

For ILU(0) applied to SPD matrices, Kershaw [92] suggested replacing negative or zero pivots with small positive values, and continuing with the factorization. For threshold-based ILU, Munksgaard [109] proposed the same kind of modification, making the pivot element comparable to the sum of the magnitudes of the off-diagonal elements in a row. Manteuffel [105] proposed the factorization of a shifted matrix $A + \alpha I$, and when A is symmetric, proved that there exists a scalar $\alpha > 0$ such that the factorization for any sparsity pattern exists. Robert [122] later extended this result for real positive definite A. Even before incomplete factorizations were used widely, Jennings and Malik [87] augmented the entries on the diagonal of a sparsified matrix to guarantee it is positive definite for a complete factorization.

Besides guaranteeing existence, Manteuffel [105] noticed that the shift α that gave the best convergence of the iterative method was not the smallest one that makes the factorization exist, but one slightly larger. The shift should make the pivots large enough so that the matrix is not too poorly conditioned. Van der Vorst [143] found the same result for nonsymmetric matrices, and suggested modifications to the diagonal to make the resulting factors diagonally dominant. He called this a "stabilized" incomplete factorization because it improved the conditioning of the resulting L and U factors. Van der Vorst was perhaps the first to notice the possible poor conditioning of L and U (and thus L^{-1} and U^{-1}) for incomplete factorizations of nonsymmetric matrices.

In general, a major difficulty is the determination of the threshold value for the pivots, or the shift α. For irregularly structured symmetric matrices, Saad [128] gave a heuristic formula for the shift to help ensure that each pivot will be greater than some small positive value. Numerical experiments for positive definite matrices show that convergence improves sharply as α is increased toward the optimal α, and then deteriorates slowly [105, 128].

Shifted or stabilized factorizations are not to be confused with modified ILU (MILU) factorizations [82] where the row-sum criteria

$$Ae = LUe, \quad e = (1, 1, \cdots, 1)^T$$

is satisfied by modifying the diagonal of L or U. For M-matrices, the modification actually decreases the size of the pivots, making the factorization less stable. Perturbed factorizations add a small value to the diagonal opposite in direction to the modification and help guarantee a bound on the largest eigenvalue of the preconditioned system; see [11, Ch. 10] for a review. These methods apply to elliptic problems in one variable, where they lower the order of the spectral condition number of the preconditioned matrix to $O(h)$, where h is the discretization size.

Relaxed ILU (RILU) [8, 9] parameterizes the fraction of the modification to perform, giving it the same effect as the perturbation. Negative relaxation factors in RILU have a stabilizing effect for M-matrices. They were used for multigrid smoothing by Wittum [148] in his ILU_β method. In this method, the diagonal is augmented with β times the sum of the magnitudes of the dropped elements. ILU_0 corresponds to the regular, unmodified factorization, ILU_{-1} corresponds to MILU, and ILU_1 corresponds to the modification of Jennings and Malik [87]. For elliptic problems that are not M-matrices, modification may also have a stabilizing effect if it increases the value on the diagonal. Elman [66] used this as part of his criteria to modify certain rows and not others, in his stabilized factorization based on RILU.

Recently, the method of diagonal compensation [7] has been developed for preconditioning positive definite matrices with incomplete factorizations. Essentially, the SPD matrix is modified into an M-matrix, for example, by dropping positive off-diagonal elements and adding them to the diagonal. An ILU factorization computed on this M-matrix (which must exist) is often a good preconditioner for the original matrix. This can be viewed as another form of stabilization.

Figure 5.3 shows the typical behavior of the incomplete factors of $A + \alpha I$ as the scalar shift parameter α is increased. The test problem was the convection-diffusion equation

$$-\nabla^2 u + 2P_1 u_x + 2P_2 u_y = f \tag{5.4}$$

on the unit square with Dirichlet boundary conditions, discretized with central differences for the first derivative. The convection coefficients were $P_1 = P_2 = 10$. The error in the preconditioner LU

for the matrix A is measured by

$$\left\| \frac{A(LU)^{-1}e}{\|A(LU)^{-1}e\|_2} - \frac{e}{\|e\|_2} \right\|_2$$

where e is the vector of all ones. The scaling of the terms is necessary when $\|A(LU)^{-1}e\|_2$ is very large, to try to make the measure independent of the measure for instability. The error decreases to an optimal point and then increases. The instability measured by *condest* ($\|(LU)^{-1}e\|_\infty$) decreases monotonely. The point that gives the best convergence corresponds to the point with minimal error.

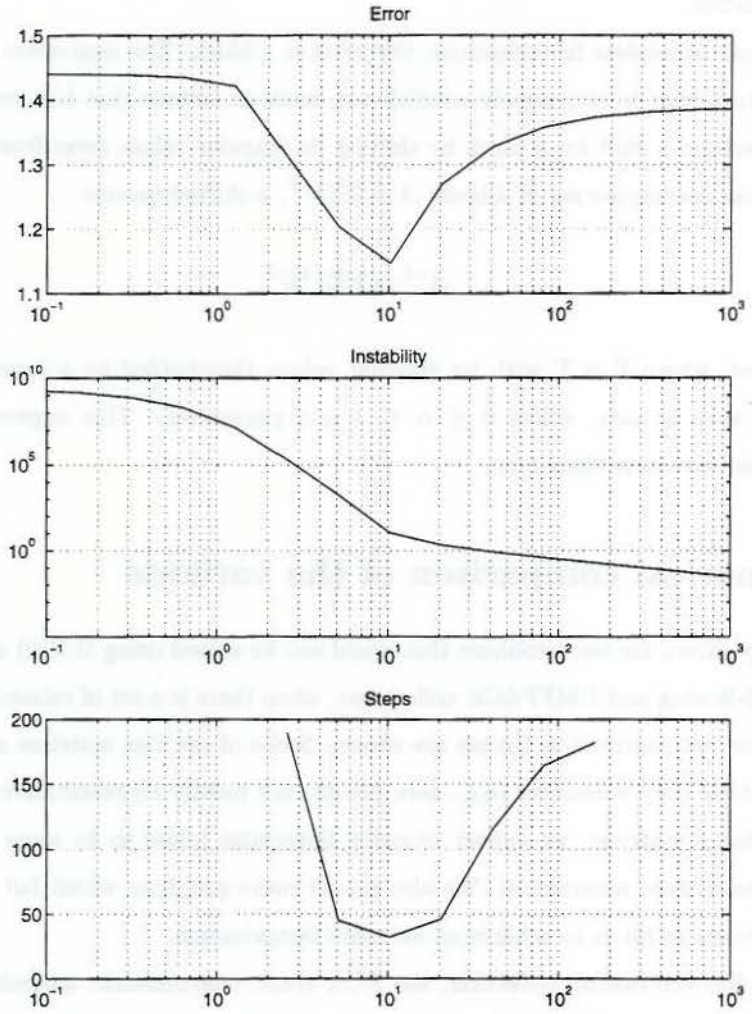


Figure 5.3: Error and instability of the ILU factors, and number of GMRES steps when solving a convection-diffusion equation. The horizontal axis is the size of the shift α.

When we focus on nonsymmetric and indefinite problems, negative pivots are acceptable and even expected. However, shifts such as $A + \alpha I$ are inadequate because they may shift the eigenvalues of A arbitrarily close to the origin; shifts of α may *decrease* the magnitude of the pivot. To avoid this, one can use a different shift for each row, computing them dynamically, during the factorization. The sign of the shift depends on the sign of the pivot.

The best shift, as discovered experimentally by Manteuffel [105] and Van der Vorst [143] earlier, is usually significantly larger than what is required to only avoid very small pivots. The larger shift has the effect of making the problem much better conditioned. This is an important effect of stabilization.

For block incomplete factorizations, the pivot is a block. The equivalent of a small pivot in this case is a block that is very poorly conditioned, with an inverse that has very large entries. It is possible to perform a shift for a block by shifting its singular values away from zero [150]. Given the singular value decomposition of a block $A = U\Sigma V^T$, a shifted inverse

$$A^{-1} \approx V\bar{\Sigma}^{-1}U^T \tag{5.5}$$

can be produced, where $\bar{\Sigma}$ is Σ with its singular values thresholded by a function of the largest singular value, such as $\alpha\sigma_1$, where $0 \leq \alpha \leq 1$ is a parameter. This approximate inverse has condition number no worse than $1/\alpha$.

5.5 Numerical comparison of the variants

Results are only shown for test problems that could not be solved using ILU(0) as a preconditioner. For the Harwell-Boeing and UMFPACK collections, when there is a set of related matrices, only the results for one or two matrices in the set are shown. Some of the test matrices are small. However, the difficulties that they encounter (e.g., zero pivots) are mostly representative of those for larger matrices. For large matrices, we expect unstable triangular solves to be more severe, since there will be longer associated recurrences. We also expect more problems which fail to converge due to insufficient amounts of fill-in to achieve an accurate factorization.

In the Harwell-Boeing collection, the RUA (real, unsymmetric, assembled) matrices were tested with ILU(0) preconditioning. Of the 97 problems, 35 were successfully solved, 56 failed due to zero pivots, and 5 did not converge. There were a large number of failures due to zero pivots because of the large number of very unstructured matrices in the collection.

Besides poor orderings that give structurally zero pivots, poor orderings can also give singular leading principal submatrices. We have found this to be common in the FIDAP test matrices. Leading principal submatrices that are singular often end with zero diagonal elements (suggesting that the equation that sets the absolute pressure was at the end). Matrices with this ordering cannot be factored exactly, but approximate factorizations are often useful. Nevertheless, singular leading principal blocks run the risk of producing very small or zero pivots, especially when the amount of fill-in is increased. (The direct solver in FIDAP does not perform pivoting, but replaces zero and small pivots with the "clipping constant," which has default value 10^{-8}.)

The test matrices are listed in Table 5.2, along with descriptions, their sizes, and their number of nonzero entries. All the matrices were scaled so that their columns have unit two-norms, and then scaled again so that their rows have unit two-norms. The importance of scaling was discussed at the end of Section 5.1. Scaling also normalizes the statistics presented in that section.

In the numerical tests in the following subsections, the iterative method used to solve these problems was right-preconditioned GMRES restarted every 50 steps. When no right-hand side was provided, a vector of all ones was used. The iterations began with a zero initial guess and were stopped when the exact residual norm was reduced by 8 orders of magnitude, or when 500 steps were taken. The latter case is indicated by a dagger (†) in the following tables.

5.5.1 Experiments with level-based ILU

We begin by showing how the statistics presented in Table 5.1 can be used to determine what difficulties are arising when an incomplete factorization fails. Table 5.3 lists problems that could not be solved using ILU(0) as a preconditioner, along with their values of the statistics, and the causes of failure as classified by Figure 5.1 and the comments in Section 5.1. We considered *condest* to be large when it was larger than 10^{10}, and used this value to classify the failures. (The value of the statistics is "Inf" when a zero pivot was encountered, and the cause of failure is understood to be a zero pivot.)

Note that *condest* can be very large; a value of 10^{96} obviously indicates a factorization that is useless. Also, the factorization is always stable unless there is a very small pivot (i.e., *max(L+U)* is never large unless *1/pivot* is large). When *max(L+U)* is large, it is usually about the same size as *1/pivot*. There are no cases where *max(L+U)* is large but *condest* is small.

About half of the failures in Table 5.3 were due to structurally zero pivots (FIDAP024 was the only case of a numerically zero pivot). These mostly correspond to very unstructured matrices,

Selected matrices from the Harwell-Boeing collection			
Matrix	n	nnz	description
BP0	822	3276	Basis matrix from the simplex method
BP1000	822	4841	Basis matrix from the simplex method
FS7603	760	5976	Chemical kinetics, 38 species
GEMAT11	4929	33185	Optimal power flow problem
GRE1107	1107	5664	Simulation of computer systems
IMPCOLD	425	1339	Chemical engineering model
LNS3937	3937	25407	Compressible Navier-Stokes
NNC1374	1374	8606	Nuclear reactor core model
ORANI678	2529	90158	Economic model of Australia
PSMIGR1	3140	543162	Demography application
SHL400	663	1712	Basis matrix from the simplex method
STR600	363	3279	Basis matrix from the simplex method
WEST0381	381	2157	Chemical engineering plant model
WEST2021	2021	7353	Chemical engineering plant model

Selected matrices from the UMFPACK collection			
Matrix	n	nnz	description
goodwin	7320	324784	CFD finite element matrix (Goodwin)
lhr01	1477	18592	Chemical process simulation (Mallya)
radfr1	1048	13299	Chemical process separation (Zitney)
shyy41	4720	20042	Fully-coupled Navier-Stokes (Shyy)
BBMAT	38744	1771722	N-S model of airfoil, ARC2D (Simon)

Selected matrices from the SPARSKIT collection			
Matrix	n	nnz	description
PULLIAM1	17028	400896	Euler model of airfoil, ARC2D, $M = 0.8$ (Pulliam)
UTM5940	5940	83842	Tokamak simulation (Brown)
WATSON5	1853	10847	Circuit simulation (Watson)
WIGTO966	3864	238252	Finite volume model of fluid flow (Wigton)
FIDAP006	1651	49533	Die-swell problem
FIDAP014	3251	66775	Isothermal seepage flow
FIDAP024	2283	48737	Unsymmetric forward roll coating
FIDAP032	1159	11343	Radiation heat transfer, open channel
FIDAPM02	537	19241	3-D steady Couette flow
FIDAPM03	2532	50380	Flow past a cylinder in freestream, Re = 40
FIDAPM07	2065	53533	Natural convection in a square enclosure
FIDAPM08	3876	103076	Developing flow, vertical channel
FIDAPM09	4683	95053	Jet impingment cooling, Re = 100
FIDAPM10	3046	53842	2-D flow over multiple steps in a channel
FIDAPM13	3549	71975	Axisymmetric poppet valve
FIDAPM15	9287	98519	Spin up of a liquid in an annulus
FIDAPM33	2353	23765	Radiation heat transfer in a square cavity

Table 5.2: Test matrices. These problems could not be solved using ILU(0) as a preconditioner.

Matrix	*max(L+U)*	*1/pivot*	*condest*	reason for failure
BP0	Inf	Inf	Inf	
BP1000	Inf	Inf	Inf	
FS7603	3.19e+02	9.53e+02	9.07e+03	inaccuracy
GEMAT11	Inf	Inf	Inf	
GRE1107	2.20e+06	2.81e+06	1.85e+96	unstable solve
IMPCOLD	Inf	Inf	Inf	
LNS3937	4.17e+11	6.36e+11	3.82e+13	small pivot
NNC1374	4.58e+08	5.27e+08	2.38e+10	small pivot
ORANI678	Inf	Inf	Inf	
PSMIGR1	Inf	Inf	Inf	
SHL400	Inf	Inf	Inf	
STR600	Inf	Inf	Inf	
WEST0381	Inf	Inf	Inf	
WEST2021	Inf	Inf	Inf	
goodwin	5.82e+05	3.63e+04	1.47e+06	inaccuracy
lhr01	Inf	Inf	Inf	
radfr1	Inf	Inf	Inf	
shyy41	Inf	Inf	Inf	
BBMAT	2.39e+06	2.00e+06	6.32e+52	unstable solve
PULLIAM1	Inf	Inf	Inf	
UTM5940	1.01e+03	2.21e+03	1.68e+04	inaccuracy
WATSON5	1.89e+00	5.63e+14	6.63e+15	small pivot
WIGTO966	3.42e+04	1.20e+04	2.11e+12	unstable solve
FIDAP006	1.46e+01	1.61e+01	4.91e+04	inaccuracy
FIDAP014	4.02e+03	9.98e+03	2.26e+20	unstable solve
FIDAP024	Inf	Inf	Inf	(numerically zero pivot)
FIDAP032	Inf	Inf	Inf	
FIDAPM02	1.36e+02	3.80e+02	2.34e+04	inaccuracy
FIDAPM03	Inf	Inf	Inf	
FIDAPM07	2.81e+03	7.29e+03	5.60e+13	unstable solve
FIDAPM08	1.52e+01	2.81e+01	1.03e+03	inaccuracy
FIDAPM09	1.21e+05	2.91e+05	1.38e+22	unstable solve
FIDAPM10	4.68e+27	7.05e+27	2.84e+31	small pivot
FIDAPM13	2.42e+27	3.47e+27	2.96e+27	small pivot
FIDAPM15	Inf	Inf	Inf	
FIDAPM33	Inf	Inf	Inf	

Table 5.3: Problems that could not be solved with ILU(0), and corresponding statistics and possible reasons for failure.

such as problem "lhr01" whose nonzero pattern is shown in Figure 5.4. Reordering and partial pivoting will be used to try to remedy the problem of structurally zero pivots. Failures due to small pivots or unstable triangular solves can be avoided to some extent by using pivoting, as discussed in Section 5.2, or by using a stabilization as discussed in Section 5.4.

Six failures were classified as due to "inaccuracy" due to dropping. For these problems, we checked whether or not allowing more fill-in would help solve these problems. The results are shown in Table 5.4. Only UTM5940 could not be solved with ILU with level as high as 2. The values of the statistics did not increase dramatically, i.e., no other effects seemed to come into play. In the case of FIDAP006, there was one very small pivot (10^{-15}) at the end of the factorization, but all other pivots were greater than 10^{-2} in magnitude. Increasing the fill-in for the other problems (with failures not classified as "inaccuracy") will not generally help, unless large amounts of fill-in is used.

Matrix	method	*max(L+U)*	*1/pivot*	*condest*	steps
FS7603	ILU(1)	2.36e+03	1.56e+03	1.59e+06	84
goodwin	ILU(2)	1.26e+05	9.63e+04	2.91e+06	417
UTM5940	ILU(2)	3.63e+02	7.12e+02	8.61e+04	†
FIDAP006	ILU(1)	2.22e+01	2.81e+14	9.52e+14	49
FIDAPM02	ILU(1)	3.85e+01	1.04e+03	5.42e+02	19
FIDAPM08	ILU(1)	1.91e+01	1.67e+01	3.99e+02	178

Table 5.4: Increasing the level-of-fill for problems classified as failed due to inaccuracy from dropping. The table shows the statistics and the number of GMRES steps for convergence.

5.5.2 Experiments with threshold-based ILU with pivoting

To remedy the problem of structurally zero pivots, we use partial pivoting. Table 5.5 shows the results using ILUTP. We used a permutation tolerance *permtol* of 1, meaning that whenever an off-diagonal element is larger than the diagonal element, a permutation occurs. Fill-in was controlled using *lfil* set to 30, i.e., 30 nonzeros in each row of L and U was allowed. This relatively large value of *lfil* helps ensure that nonzero pivots can be found.

The results show that there are only two cases where a nonzero pivot could not be found, whereas there were 19 cases of zero pivots with ILU(0). In the 17 cases that pivoting helped, all problems except three could now be solved. This suggests that ILU(0) had failed on problems due to structurally zero pivots, which were otherwise fairly easy to solve.

Pivoting can also to some extent help avoid very small pivots and enhance the stability of

the triangular solves. To illustrate this, in Table 5.6 we show the same experiment with ILUTP as above, but use a smaller *permtol* of 0.01. (We do not use a *permtol* of 0 since this result with no pivoting is extremely poor, i.e., ILUT performs very poorly on this test set without pivoting due to the problems discussed in Section 5.1.) There are four more failures, and the results here are poorer. There are three interesting cases: GEMAT11, WIGTO966, and FIDAPM03. These problems failed due to unstable triangular solves with *permtol* of 0.01, but were solved successfully when *permtol* of 1 was used.

There are cases, with both values of *permtol*, where ILUTP encountered a zero pivot while no zero pivots were encountered with ILU(0). Thus it is not rare for ILUTP to produce a poor pivoting sequence.

We emphasize that the matrices were scaled as described at the end of Section 5.1. There were many zero pivots and extremely large values of *condest* when scaling was not used.

5.5.3 Experiments with ILUS and reordering

ILU in skyline form (ILUS) was tested on the FIDAP matrices, since these matrices have symmetric structure. However, it is difficult to perform pivoting on matrices stored in bordered form. Thus some sort of preordering must be used instead.

For the FIDAP matrices, there is an obvious reordering that that may give a good factorization. In the original matrices, the unknowns were ordered element by element, with the continuity equations ordered last for each element. A better ordering is to order the continuity equations at the end of all other equations for all elements. This ordering gives a zero block in the lower right-hand corner of the matrix, and we call this a *block* reordering (the matrix is a 2 by 2 block matrix). This ordering ensures that there are no structurally zero pivots.

Table 5.7 shows the results of ILUS using this reordering. The version of ILUS used in this section used Algorithm 2.4.2 with dropping, rather than the approximate inverse technique. The fill-in was controlled to be not more than the fill-in for ILU(0). For comparison, we show in Table 5.8 the results of ILU(0) and ILUT, all with comparable amounts of fill-in. ILUS was the most reliable preconditioner. When the block reordering is used, none of the preconditioners encountered zero pivots, and all values of *1/pivot* are less than 10^8 (not shown). The results without this reordering are very poor for ILUS; for ILUT, all the failures shown for the original ordering were due to zero pivots.

If we increase the amount of fill-in but do not use this reordering, ILU(1), for example, can

Matrix	*max(L+U)*	*1/pivot*	*condest*	steps	reason for failure
BP0	1.45e+02	2.44e+02	1.13e+04	3	
BP1000	2.26e+01	1.30e+03	5.31e+03	13	
FS7603	7.89e+02	8.63e+02	1.07e+10	†	inaccuracy
GEMAT11	4.99e+02	1.09e+03	8.20e+04	25	
GRE1107	9.63e+00	2.97e+01	1.18e+04	†	inaccuracy
IMPCOLD	2.25e+00	7.81e+00	3.16e+02	2	
LNS3937	1.72e+09	1.14e+09	2.55e+19	†	unstable solve
NNC1374	1.49e+09	1.67e+10	5.19e+172	†	unstable solve
ORANI678	6.37e+00	1.70e+01	7.66e+01	8	
PSMIGR1	7.58e+00	2.81e+01	3.98e+03	9	
SHL400	3.28e+01	2.56e+03	5.83e+05	3	
STR600	3.60e+01	4.95e+01	4.96e+03	4	
WEST0381	2.26e+01	3.41e+01	2.04e+02	11	
WEST2021	1.27e+05	2.19e+05	1.17e+07	8	
goodwin	1.55e+02	2.09e+07	3.37e+70	†	unstable solve
lhr01	Inf	Inf	Inf	†	zero pivot
radfr1	1.83e+01	1.90e+01	2.29e+04	25	
shyy41	2.07e+01	1.06e+37	1.35e+37	†	small pivot
BBMAT	6.25e+16	1.49e+09	1.83e+177	†	unstable solve
PULLIAM1	1.02e+03	2.26e+16	1.16e+209	†	unstable solve
UTM5940	7.63e+01	1.80e+05	3.81e+32	†	unstable solve
WATSON5	6.98e+02	1.69e+05	7.21e+04	8	
WIGTO966	4.84e+00	3.91e+00	9.98e+03	247	
FIDAP006	8.27e+00	3.70e+02	8.83e+03	26	
FIDAP014	9.10e+02	2.77e+04	2.05e+64	†	unstable solve
FIDAP024	4.29e+00	4.84e+00	1.06e+03	20	
FIDAP032	2.27e+00	7.43e+01	3.21e+03	6	
FIDAPM02	3.87e+01	5.34e+01	6.04e+04	141	
FIDAPM03	8.21e+00	8.97e+00	4.32e+02	30	
FIDAPM07	1.14e+03	5.76e+04	3.24e+28	†	unstable solve
FIDAPM08	3.27e+00	1.58e+01	8.39e+05	24	
FIDAPM09	Inf	Inf	Inf	†	zero pivot
FIDAPM10	4.23e+00	1.78e+02	3.63e+04	25	
FIDAPM13	9.87e+01	8.22e+02	3.37e+06	†	inaccuracy
FIDAPM15	4.93e+00	9.53e+00	1.06e+07	60	
FIDAPM33	8.83e+00	1.22e+01	1.35e+04	4	

Table 5.5: Results for ILUTP (*lfil*=30, *permtol*=1.00) for problems that failed with ILU(0). The statistics are shown along with the number of GMRES steps required for convergence, or the possible reason for failure.

Matrix	*max(L+U)*	*1/pivot*	*condest*	steps	reason for failure
BP0	1.45e+02	2.44e+02	1.13e+04	3	
BP1000	6.12e+02	1.40e+03	2.29e+05	32	
FS7603	1.76e+03	1.87e+03	2.32e+07	195	
GEMAT11	9.15e+03	2.85e+04	7.49e+14	†	unstable solve
GRE1107	1.17e+03	1.88e+06	1.78e+34	†	unstable solve
IMPCOLD	5.92e+02	1.08e+02	4.53e+04	9	
LNS3937	Inf	Inf	Inf	†	zero pivot
NNC1374	5.71e+10	9.59e+09	5.75e+250	†	unstable solve
ORANI678	Inf	Inf	Inf	†	zero pivot
PSMIGR1	6.11e+00	4.71e+02	3.95e+03	9	
SHL400	3.28e+01	2.56e+03	5.83e+05	3	
STR600	2.54e+02	9.99e+01	5.11e+03	5	
WEST0381	2.55e+03	2.49e+02	2.48e+05	32	
WEST2021	1.93e+07	1.93e+07	3.86e+06	15	
goodwin	1.64e+04	1.47e+04	3.88e+42	†	unstable solve
lhr01	Inf	Inf	Inf	†	zero pivot
radfr1	9.10e+04	2.56e+02	2.44e+06	80	
shyy41	9.98e+01	3.55e+36	4.94e+42	†	small pivot
BBMAT	7.60e+19	2.60e+10	1.37e+474	†	unstable solve
PULLIAM1	6.39e+05	1.04e+12	3.80e+374	†	unstable solve
UTM5940	4.14e+02	1.87e+03	2.02e+07	†	inaccuracy
WATSON5	1.38e+03	5.65e+04	3.32e+04	8	
WIGTO966	2.25e+03	2.58e+02	2.23e+12	†	unstable solve
FIDAP006	1.26e+02	5.11e+13	1.69e+14	30	
FIDAP014	Inf	Inf	Inf	†	zero pivot
FIDAP024	8.97e+01	1.73e+02	4.34e+05	98	
FIDAP032	2.72e+00	1.18e+01	3.21e+03	6	
FIDAPM02	3.85e+01	1.04e+03	2.97e+02	13	
FIDAPM03	1.83e+02	1.52e+02	4.82e+17	†	unstable solve
FIDAPM07	4.49e+03	2.99e+05	5.14e+35	†	unstable solve
FIDAPM08	2.84e+01	1.62e+01	2.27e+06	28	
FIDAPM09	1.04e+03	1.27e+22	1.51e+314	†	unstable solve
FIDAPM10	2.29e+01	4.27e+01	1.82e+04	17	
FIDAPM13	1.39e+02	4.09e+02	2.81e+25	†	unstable solve
FIDAPM15	Inf	Inf	Inf	†	zero pivot
FIDAPM33	6.17e+02	1.33e+02	3.73e+18	3	

Table 5.6: Results for ILUTP (*lfil*=30, *permtol*=0.01) for problems that failed with ILU(0). The statistics are shown along with the number of GMRES steps required for convergence, or the possible reason for failure. The results are slightly worse than when *permtol*=1.00.

only help solve 3 FIDAP problems. However, if the block reordering is used, ILU(1) helps solve *all* the problems (not shown). Also as fill-in is increased, the results of ILUT and ILUS become very similar (not shown).

Matrix	*max(L+U)*	*1/pivot*	*condest*	steps
FIDAP006	4.19e+01	6.31e+01	0.56e+04	251
FIDAP014	2.40e+03	2.27e+06	0.30e+09	†
FIDAP024	4.25e+00	9.74e+00	0.17e+03	168
FIDAP032	2.95e+00	6.76e+00	0.47e+02	†
FIDAPM02	1.10e+01	5.32e+02	0.93e+03	102
FIDAPM03	1.79e+01	2.47e+01	0.47e+03	57
FIDAPM07	2.34e+04	7.17e+05	0.16e+06	†
FIDAPM08	4.94e+00	9.43e+00	0.95e+03	262
FIDAPM09	2.48e+02	5.99e+02	0.53e+04	†
FIDAPM10	1.08e+01	1.57e+01	0.33e+03	140
FIDAPM13	3.76e+02	5.63e+02	0.29e+05	79
FIDAPM15	1.34e+01	3.25e+01	0.34e+05	†
FIDAPM33	9.46e+00	2.50e+01	0.83e+03	24

Table 5.7: ILUT in bordered form with fill-in comparable to ILU(0). The matrices were reordered with continuity equations last (block reordering).

5.5.4 Experiments with stabilized ILU

Stabilization can be an effective option when *condest* is large, or there are small pivots. We tested the problems in Table 5.2 with a stabilized version of ILUT. Pivots whose absolute value were smaller than a parameter *thresh* were replaced by *thresh* with the original sign of the pivot. This worked very well for the FIDAP matrices. For the very unstructured matrices, this strategy did not help. Pivoting is a better solution for this latter class of matrices.

Table 5.9 shows the result for the FIDAP matrices of ILUT with *lfil* parameter 30, and *thresh* set to 0.5 (i.e., 1/pivtol ≤ 2), a relatively large value. As mentioned, this shift is usually much larger than what is suggested in the literature to plainly avoid very small pivots. The larger shift has the effect of making the problem much better conditioned. Without the shift, we could only solve problems FIDAP006, FIDAPM02, and FIDAPM08.

In Table 5.10(a) we perform a parameter study of the effect of changing *thresh* for the WIGTO966 problem. Pointwise ILU(0) was used, with GMRES(100) and a tolerance of 10^{-6}. As *thresh* is increased, *condest* decreases. This was always true in our experiments. However, the best

Matrix	ILU(0) reord	ILUT orig	ILUT reord	ILUS reord
FIDAP006	†	345	†	251
FIDAP014	†	†	†	†
FIDAP024	†	†	†	168
FIDAP032	245	†	274	†
FIDAPM02	219	49	†	102
FIDAPM03	119	†	153	57
FIDAPM07	†	†	†	†
FIDAPM08	†	189	†	262
FIDAPM09	†	†	†	†
FIDAPM10	236	†	225	140
FIDAPM13	150	†	448	79
FIDAPM15	†	†	†	†
FIDAPM33	25	†	38	24
total successes	6	3	5	8

Table 5.8: Number of steps for convergence with the use of various preconditionings, all with comparable fill-in; original ordering (orig) and block reordering (reord).

Matrix	*max(L+U)*	*condest*	steps
FIDAP006	2.30e+00	4.47e+02	46
FIDAP014	1.15e+00	1.74e+01	†
FIDAP024	2.85e+00	1.14e+02	33
FIDAP032	1.98e+00	1.98e+02	30
FIDAPM02	1.02e+00	5.60e+01	85
FIDAPM03	1.06e+01	2.35e+02	88
FIDAPM07	1.50e+00	2.12e+02	474
FIDAPM08	2.96e+00	3.88e+02	247
FIDAPM09	1.47e+00	9.26e+17	†
FIDAPM10	1.52e+01	1.14e+03	50
FIDAPM13	2.36e+00	4.57e+02	486
FIDAPM15	9.06e+00	6.81e+02	†
FIDAPM33	1.35e+01	5.20e+03	24

Table 5.9: Results for stabilized ILUT, $lfil = 30$, $thresh = 0.5$.

threshold balances the accuracy of the factorization and the stability of the triangular solves.

thresh	*condest*	steps
0.	2.19e+17	†
0.001	4.63e+17	†
0.002	1.75e+09	†
0.003	1.04e+06	73
0.004	9.84e+03	90
0.005	7.42e+03	84
0.01	1.85e+03	90
0.1	2.64e+02	†

(a) Pointwise ILU(0)

thresh	*condest*	steps
0.	1.51e+08	†
0.001	7.22e+09	†
0.01	5.30e+05	72
0.1	7.24e+04	43
0.5	1.16e+03	177
1.0	4.25e+02	†

(b) Block ILU(0)

Table 5.10: Stabilized ILU(0), (a) pointwise, and (b) block versions, for the WIGTO966 problem.

The WIGTO966 matrix comes from an Euler model of an airfoil with four degrees of freedom at each grid point. Thus we can use block ILU with a block size of 4, and illustrate the use of a block shift (5.5). Table 5.10(b) shows the results. Here, *thresh* is the ratio of the largest singular value to the smallest in (5.5). Our experiments with other problems generally show that when shifting is successful, it does not matter if a pointwise or block shift is used.

5.5.5 Harder problems

There are several problems in Table 5.2 for which we have not yet presented a successful solution method. Consider first the matrix "lhr01." ILU(0) and ILUTP both encounter zero pivots when trying to approximately factor this matrix. Figure 5.4 shows its nonzero pattern, and Figure 5.5 is a close-up of the top-left 100×300 block. For rows 25 to 60, there are not many choices for good pivots, when column pivoting is used. However, there may be many good choices of pivots if row pivoting is used. Thus we used a column-wise ILU algorithm with row pivoting (actually, we only computed the ILU factors of the transposed matrix data structure), and no zero pivots were encountered. Table 5.11 shows the problem was solved in 134 steps. By not applying these algorithms blindly, for example, by looking at the structure of the matrix in this case, we were able to make ILU work.

Another problem for which we had difficulty was GRE1107. ILUTP(30) with *permtol* 1.0 suggests that the difficulty is inaccuracy, and we start from there. We increased *lfil* to 50, but the

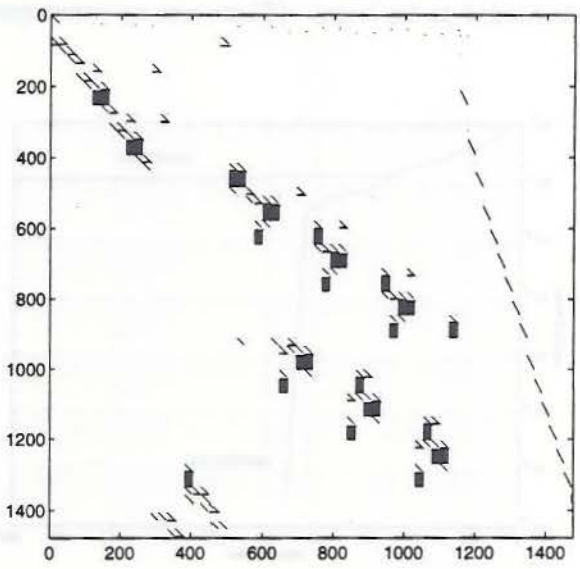


Figure 5.4: Nonzero pattern for "lhr01."

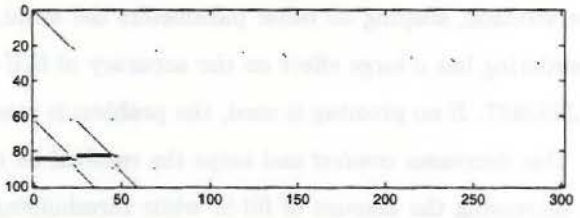


Figure 5.5: Nonzero pattern for "lhr01," top-left 100×300 block.

GMRES solver was still stagnating. By looking at the convergence history, Figure 5.6, convergence is steady until GMRES restarts at step 50. However, GMRES will converge in 94 steps if we do not restart (we tried GMRES(100)). In this case, we were able to make ILU work by being aware that the Krylov subspace basis needed to be larger. (There was no convergence with ILUTP(30) and GMRES(100).)

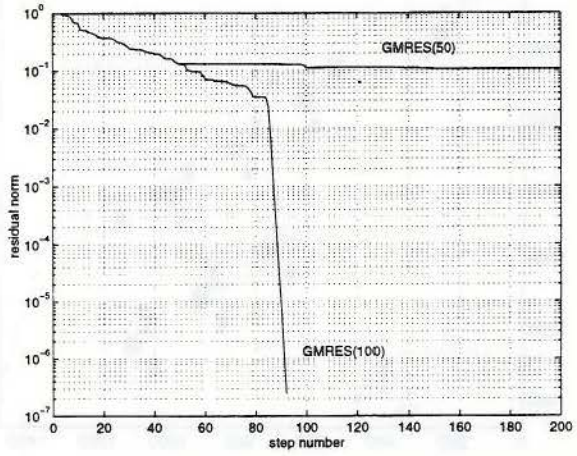


Figure 5.6: Convergence history for GRE1107.

We also could not solve UTM5940, this time due to inaccuracy in ILU(0) or unstable solves in ILUTP. For this problem, we know that zero pivots are not encountered with the original ordering, so we tried ILUT(30) without pivoting. The failure in this case could be classified as inaccuracy due to dropping, i.e., it was the pivoting that made the solves unstable in this case. Next we tried increasing *lfil* to 50, and the solution was found in 399 steps. The contributor of this matrix, Peter Brown, had found that reordering this matrix with reverse Cuthill-McKee (RCM) ordering makes ILU more effective. The solution, keeping all other parameters the same, was found in 37 steps in this case. In general, reordering has a large effect on the accuracy of ILU preconditioners [60, 61].

Consider now LNS3937. If no pivoting is used, the problem is small pivots. We thresholded the pivots for ILU(30). This decreases *condest* and helps the residual be reduced further, but there is still no convergence. Increasing the amount of fill-in while thresholding the pivots also does not help achieve convergence.

Our experience with NNC1374 is somewhat different. If ILUT with or without pivoting is used, the solves are unstable, probably due to the problems discussed in Section 5.1. Only very large values of the *thresh* stabilization parameter can reduce *condest* a significant amount. Thus

we go back to ILU(0), which had failed due to small pivots. We try thresholding the pivots in this case, but this did not help. By increasing the amount of fill-in at the same time, the solves became unstable.

There are several problems, such as the above two, that are very difficult to solve by using ILU preconditioners. A third problem, "shyy41," is even difficult to solve with *direct* solvers: the factorization is stable, but the triangular solves are very unstable. This matrix contains an independent diagonal block that is a 5-point matrix with a zero diagonal. Solves with the factors of this block are very unstable.

Matrix	method	*max(L+U)*	*1/pivot*	*condest*	steps
lhr01(t)	ILUTP(30)	4.58e+03	9.68e+09	3.44e+11	134
GRE1107(f)	ILUT(50)	1.57e+01	3.40e+01	1.43e+04	94
UTM5940	ILUT(50)	2.23e+03	7.20e+02	5.24e+06	399
UTM5940(r)	ILUT(50)	1.37e+02	7.06e+02	5.56e+06	37

Table 5.11: Solution of some harder problems. Notes: (t) transposed data structure and ILUTP with *permtol*=1; (f) full GMRES was used; (r) reordered with RCM.

5.5.6 Block ILU preconditioners

Many linear systems from engineering applications arise from the discretization of coupled partial differential equations. A blocking in these systems may be imposed by ordering together the equations and unknowns at a single grid point. Experimental tests suggest it is advantageous for preconditionings to exploit this block structure in a matrix. In *block* incomplete factorizations (BILU), each entry in Algorithm 2.4.1 is actually a small dense block. Dropping of the blocks can be based on the block level (BILUK) or the Frobenius norm of the block (BILUT).

PULLIAM1 and BBMAT are two matrices with the above block structure. We will briefly compare BILUK and BILUT, and show the effect of increasing the block size. Figure 5.7 shows *condest* for BILUT(*lfil*) applied to PULLIAM1, with blocksize 8 (*lfil* now refers to the number of blocks in a block row). The shape of this graph is typical: as fill-in is increased, the triangular solves become more unstable, until the factorization approaches that of a direct solve. For low amounts of fill-in, there are not enough nonzeros to make very large values of *condest*. (See also Figure 1.2.) BILUK(0) corresponds to *lfil* of approximately 4. BILUT is successful in this case for *lfil* approximately 23. Van der Vorst [144] briefly investigated the effect that increasing fill-in has on

stability in the nonsymmetric case. His conclusion also was that increasing the accuracy does not seem to help, unless of course, the accuracy approaches that of a direct solve.

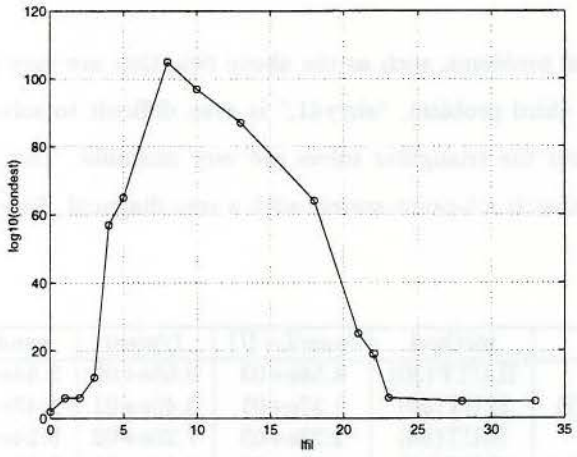


Figure 5.7: Condition estimate for BILUT(*lfil*) for PULLIAM1. The blocksize was 8.

Table 5.12(a) shows *condest* for BILUK applied to the PULLIAM1 problem. The values are much smaller. This suggests that threshold-based incomplete factorizations are much more prone to unstable triangular solves than level-based ones. An alternative when an threshold-based ILU is unstable is to use a level-based factorization. Table 5.12(b) shows the number of GMRES steps required to solve the PULLIAM1 problem. Table 5.13(a) shows the number of GMRES steps required to solve the BBMAT problem, along with timings on a Cray-C90 computer. Note that block size 16 is fastest, even though the factorization requires 50 percent more storage than block size 8 (due to some explicit storage of zeros; see Table 5.13(b)), partially due to better vectorization. The storage for a direct solver is approximately 36.0 million nonzero entries.

5.6 Summary

It is clear that the *blind* application of incomplete factorizations will be unsuccessful for many problems. However, by being attentive to the characteristics of a problem and the difficulties it encounters, incomplete factorizations can be made effective. For example, problem "lhr01" encountered structurally zero pivots even when column pivoting was used. After looking at the structure of the matrix, we were able to solve the problem by using row pivoting. Poor orderings and scalings are other characteristics of matrices that can cause difficulties.

block size	BILUK level 0	1	2
4	9.74e+16	4.89e+11	4.94e+13
8	2.64e+10	4.25e+09	1.03e+09
16	4.51e+09	2.18e+09	1.90e+09

(a) condest

block size	BILUK level 0	1	2
4	†	†	†
8	148	61	42
16	147	39	40

(b) GMRES steps

Table 5.12: BILUK preconditioning for the PULLIAM1 problem. GMRES(100) was used to reduce the residual norm by 10^{-6}.

block size	GMRES steps	CPU time (s) precon	solve	total
4	†	291.1	†	†
8	56	61.1	162.6	223.8
16	51	28.1	70.9	99.0

(a) Timings on one processor of a Cray-C90

block size	BILUK level 0	1	2
1	1.8	5.7	11.4
4	3.3	8.5	13.0
8	4.2	9.7	14.2
16	7.6	14.2	21.9

(b) Number of scalar nonzeros for BILUK, in millions

Table 5.13: BILUK preconditioning for the BBMAT problem.

We presented several statistics that can help determine the causes of failure of incomplete factorizations. These statistics measure obvious quantities: the stability of the triangular solves, the smallest pivots, and the stability of the factorization.

The occurrence of zero pivots is very common in very unstructured problems. Partial pivoting is very effective in remedying this problem. Thresholding small and zero pivots is a less effective solution when the matrix is very unstructured. However, large values of the threshold (i.e., of the stabilization, or perturbation to the pivot) has another, more important effect: it makes the L and U factors better conditioned. This is an important effect, even if the problem can be solved with ILU(0).

The most difficult problems to solve were those with unstable triangular solves that were not caused by very small pivots. The $(LU)^{-1}$ condition estimate can be reduced by thresholding the pivots, but very large thresholds are required. This destroys the accuracy of the factorization, and usually, increasing the fill-in is done in vain. For these problems, the last resort seems to be to use very large amounts of fill-in [41], for example, as we did for the PULLIAM1 and BBMAT problems.

General-purpose software for incomplete factorizations should include options for pivoting and perturbing pivots. The latter should be a particularly simple addition to any incomplete factorization code. The difficulty is determining when these options should be used, and the values for their parameters. The statistics introduced in this chapter can be used to determine what difficulties are occurring, and to guide the selection of parameters or variants of incomplete factorizations. The hope is that for a class of problems, one can find a strategy or set of parameters that is effective for all problems in that class.

Chapter 6

Object-oriented implementation

6.1 Motivation

General software for preconditioning is seriously lagging behind methods being published in the literature. Part of the reason is that many methods do not have general applicability: they are not robust on general problems, or they are specialized and need specific information (e.g., general direction of flow in a fluids simulation) that cannot be provided in a general setting.

Another reason, one that we will deal with in this chapter, is that specific linear systems need specific matrix and preconditioner data structures in order to be solved efficiently; i.e., there need to be multiple implementations of a preconditioner with specialized data structures. For example, in some finite element applications, diagonal blocks have a particular but fixed sparse structure. A block SSOR preconditioner that needs to invert these diagonal blocks should use an algorithm suited to this structure. A block SSOR code that treats these diagonal blocks in a general way is not ideal for this problem.

When we encounter linear systems from different applications, we need to determine suitable preconditioning strategies for their iterative solution. Rather than code preconditioners individually to take advantage of the structure in each application, it is better to have a framework for software reuse. Also, a wide range of preconditionings should be available so that we can choose a method that matches the difficulty of the problem and the computer resources available.

This chapter presents a framework to support preconditioning with various, possibly user-defined, data structures for matrices that are partitioned into blocks. The main idea is to define data structures (called *block types*) for the blocks, and an upper layer of software which uses these blocks transparently of their data structure. Thus various preconditioners, such as block relaxations and block incomplete factorizations, only need to be defined once, and will work with any block type. These preconditioners are called *global preconditioners* for reasons that will soon become apparent. The code for these preconditioners is almost as readable as the code for their pointwise counterparts. New global preconditioners can be added in the same fashion.

Global preconditioners need methods (called *local preconditioners*) to approximately or ex-

actly invert pivot blocks, or solve systems whose coefficient matrices are diagonal blocks. For example, a block stored in a sparse format might be inverted exactly, or an approximate inverse might be computed. Our design permits a variety of these inversion or solution techniques to be defined for each block type.

The transparency of the block types and local preconditioners can be implemented through polymorphism in an object-oriented language. Our framework, called BPKIT,[1] currently implements block incomplete factorization and block relaxation global preconditioners, a dense and a sparse block type, and a variety of local preconditioners for both block types. Users of BPKIT will either use the block types that are available, or add block types and local preconditioners that are appropriate for their applications. Users may also define new global preconditioners that take advantage of the existing block types and local preconditioners. Thus BPKIT is not intended to be complete library software; rather it is a framework under which software can be specialized from relatively generic components.

It is appropriate to make some comments about why we use *block* preconditioning. Many linear systems from engineering applications arise from the discretization of coupled partial differential equations. The blocking in these systems may be imposed by ordering together the equations and unknowns at a single grid point, or those of a subdomain. In the first case, the blocks are usually dense; in the latter case, they are usually sparse. Experimental tests suggest it is very advantageous for preconditionings to exploit this block structure in a matrix [43, 69, 89, 95]. The relative robustness of block preconditioning comes partly from being able to solve accurately for the strong coupling within these blocks. From a computational point of view, these block matrix techniques can be more efficient on cached and hierarchical memory architectures because of better data locality. In the dense block case, block matrix data structures also require less storage. Block data structures are also amenable to graph-based reorderings and block scalings.

When approximations are also used for the diagonal or pivot blocks (i.e., approximations with local preconditioners are used), these techniques are specifically called *two-level* preconditioners [97], and offer a middle-ground between accuracy and simpler computations. Beginning with [141] in 1976 and then [6] and [49] more than a decade ago, these preconditioners have been motivated and analyzed in the case of block tridiagonal incomplete factorizations combined with several types of approximate inverses, and have recently reached a certain maturity. Most implementations of these methods, however, are not flexible: they are often coded for a particular block size and inversion

[1] The BPKIT software is available at http://www.cs.umn.edu/~chow/bpkit.html.

technique, and further, they are almost always coded for dense blocks.

The software framework presented here derives its flexibility from the use of an object-oriented language. We chose to use C++ [137] in real, 64-bit arithmetic. Other object-oriented languages are also appropriate. The framework is computationally efficient, since all operations involving blocks are performed with code that employs fundamental types, or with optimized Fortran 77 libraries such as the Level 3 BLAS [54], LAPACK [52], and the sparse BLAS toolkit [36]. By the same token, users implementing block types and local preconditioners may do so in practically any language, as long as the language can be linked with C++ by their compilers. BPKIT also has an interface for Fortran 77 users.

Other C++ efforts in the numerical solution of linear equations include LAPACK++ [56] for dense systems, and Diffpack [31], ISIS++ [47], SparseLib++ and IML++ [55] for sparse systems. It is also possible to use an object-oriented style in other languages [63, 103, 136].

6.2 Interfaces for block preconditioning

We have attempted to be general when defining interfaces (to allow for extensions of functionality), and we have attempted to accept precedents where we overlap with related software (particularly in the interface with iterative methods).

6.2.1 Block matrices

A matrix that is partitioned into blocks is called a *block matrix*. Although with BPKIT any storage scheme may be used to store the blocks that are not zero, the locations of these blocks within the block matrix must still be defined. The block matrix class (data type) that is available in BPKIT, called `BlockMat`, contains a pointer to each block in the block matrix. The pointers for each row of blocks (block row) are stored contiguously, with additional pointers to the first pointer for each block row. This is the analogy to the compressed sparse row data structure [124] for pointwise matrices; pointers point to blocks instead of scalar entries. The global preconditioners in BPKIT assume that the `BlockMat` class is being used. It is possible for users to design new block matrix classes and to code new global preconditioners for their problems, and still use the block types and local preconditioners in BPKIT.

For the block matrix data structure described above, BPKIT provides conversion routines to that data structure from the Harwell-Boeing format [59]. There is one conversion routine for each

block type (e.g., one routine will convert a Harwell-Boeing matrix into a block matrix whose blocks are dense). However, these routines are provided for illustration purposes only. In practice, a user's matrix that is already in block form (i.e., the nonzero entries in each block are stored contiguously) can usually be easily converted by the user directly into the `BlockMat` form.

To be general, the conversion routines allow two levels of blocking. In many problems, particularly linear systems arising from the discretization of coupled partial differential equations, the blockings may be imposed by ordering together the equations and unknowns at a single grid point and those of a subdomain. The latter blocking produces *coarse*-grain blocks, and the smaller, nested blocks are called *fine*-grain blocks. Figure 1.1 in Chapter 1 showed a block matrix of dimension 24 with coarse blocks of dimension 6 and fine blocks of dimension 2.

The blocks in BPKIT are the coarse blocks. Information about the fine blocks should also be provided to the conversion routines because it may be desirable to store blocks such that the coarse blocks themselves have block structure. For example, the variable block row (VBR) [124] storage scheme can store coarse blocks with dense fine blocks in reduced space. Optimized matrix-vector product and triangular solve kernels for the VBR and other block data structures are provided in the sparse BLAS toolkit [36, 121]. No local preconditioners or block operations, however, are defined for fine blocks (i.e., there are not two levels of local preconditioners).

It is apparent that the use of very small coarse blocks will degrade computing performance due to the overhead of procedure calls. Larger blocks can give better computational efficiency and convergence rate in preconditioned iterative methods, and computations with large dense blocks can be vectorized. In this chapter, we will rarely have need to mention fine blocks; thus, when we refer to "blocks" with no distinction, we normally mean coarse blocks.

To be concrete, we give an example of how a conversion routine is called when a block matrix is defined. The statement

```
BlockMat B("HBfile", 6, DENSE);
```

defines `B` to be a square block matrix where the blocks have dimension 6, and the blocks are stored in a format indicated by `DENSE` (which is of a C++ enumerated type). The other block type that is implemented is `CSR`, which stores blocks in the compressed sparse row format. The matrix is read from the file `HBfile`, which must be encoded in the standard Harwell-Boeing format [59]. (The dimension of the matrix does not need to be specified in the declaration since it is stored within the file.) To specify a *variable* block partitioning (with blocks with different sizes), other interfaces are available which use vectors to define the coarse and fine partitionings.

6.2.2 Specifying the preconditioning

A preconditioning for a block matrix is specified by choosing

1. a *global preconditioner*, and
2. a *local preconditioner* for each diagonal or pivot block to exactly or approximately invert the block or solve the corresponding set of equations.

For example, to fully define the conventional block Jacobi preconditioning, one must specify the global preconditioner to be block Jacobi and the local preconditioner to be LU factorization.

In addition, the block size of the matrix has a role in determining the effect of the preconditioning. At one extreme, if the block size is one, then the preconditioning is entirely determined by the global preconditioner. At the other extreme, if there is only one block, then the preconditioning is entirely determined by the local preconditioner. The block size parameterizes the effect and cost between the selected local and global preconditioners. The best method is likely to be somewhere between the two extremes.

For example, suppose symmetric successive overrelaxation (SSOR) is used as the global preconditioner, and complete LU factorization is used as the local preconditioner. For linear systems that are not too difficult to solve, SSOR may be used with a small block size. For more challenging systems, larger block sizes may be used, giving a better approximation to the original matrix. In the extreme, the matrix may be treated as a single block, and the method is equivalent to LU factorization.

A global preconditioner `M` is specified with a very simple form of declaration. In the case of block SSOR, the declaration is

```
BSSOR M;
```

Two functions are used to specify the local preconditioner and to provide parameters to the global preconditioner:

```
M.localprecon(LP_LU);       // LU factorization for the blocks
M.setup(B, 0.5, 3);         // BSSOR(omega=0.5, iterations=3)
```

Here `B` is the block matrix defined as in Section 6.2.1. The `setup` function provides the real data to the preconditioner, and performs all the computations necessary for setting up the global preconditioner, for example, the computation of the LU factors in this case. Therefore, `localprecon`

must be called before `setup`. The `setup` function must be called again if the local preconditioner is changed. In these interfaces, the same local preconditioner is specified for all the diagonal blocks. In general, however, the local preconditioners are not required to be the same. In some applications, different variables (e.g., velocity and pressure variables in a fluids simulation) may be blocked together. It may then make sense to write a specialized global preconditioner with an interface that allows different local preconditioners to be specified for each block.

Global preconditioners

The global preconditioners that we have implemented in BPKIT are listed in Table 6.1, along with the arguments of the `setup` function, and any default argument values.

	`setup` arguments
`BJacobi`	none
`BSOR`	omega=1.0, iterations=1
`BSSOR`	omega=1.0, iterations=1
`BILUK`	level
`BTIF`	none

Table 6.1: Global preconditioners.

`BJacobi`, `BSOR` and `BSSOR` are block versions of the diagonal, successive overrelaxation, and symmetric successive overrelaxation preconditioners. `BILUK` is a block version of level-based incomplete LU (ILU) factorization. `BTIF` is an incomplete factorization for block tridiagonal matrices.

Local preconditioners

Local preconditioners are either *explicit* or *implicit* depending on whether (approximate) inverses of blocks are explicitly formed. An example of an implicit local preconditioner is LU factorization.

The global preconditioners that involve incomplete factorization require the inverses of pivot blocks. For large block sizes, the use of approximate or exact *dense* inverses usually requires large amounts of storage and computation. Thus *sparse* approximate inverses should be used in these cases. Implicit local preconditioners produce inverses that are usually dense, and are therefore usually not computationally useful for block incomplete factorizations. This use of implicit local preconditioners is disallowed within BPKIT. (This rule is also applied when small block sizes are used, since dense exact inverses are usually most efficient here. Also, if an exact factorization is sought, it is usually

most efficient to use an LU factorization on the whole matrix.) The global preconditioners that involve block relaxation may use either explicit or implicit local preconditioners, but usually the implicit ones are used. Explicit local preconditioners can be appropriate for block relaxation when the blocks are small.

Local preconditioners are also differentiated by the type of the blocks on which they operate. Not all local preconditioners exist for all block types; incomplete factorization, for example, is only meaningful for sparse types. Thus, a local preconditioner must be chosen that matches the type of the block.

BPKIT requires the user to be aware of the restrictions in the above two paragraphs when selecting a local preconditioner. Due to the dynamic binding of C++ virtual functions, violations of these restrictions will only be detected at run-time.

Table 6.2 lists the local preconditioners that we have implemented, along with their `localprecon` arguments, their block types, and whether the local preconditioner is explicit or implicit. In contrast to the `setup` function, `localprecon` takes no default arguments. We have included an explicit exact inverse local preconditioner for the `CSR` format for comparison purposes (it would be inefficient to use it in block tridiagonal incomplete factorizations, for example).

	`localprecon` arguments	Block type	Expl./Impl.
LP_LU	none	DENSE	implicit
LP_INVERSE	none	DENSE	explicit
LP_SVD	alpha1, alpha2	DENSE	explicit
LP_LU	none	CSR	implicit
LP_INVERSE	none	CSR	explicit
LP_RILUK	level, omega	CSR	implicit
LP_ILUT	lfil, threshold	CSR	implicit
LP_APINV_TRUNC	semibw	CSR	explicit
LP_APINV_BANDED	semibw	CSR	explicit
LP_APINV0	none	CSR	explicit
LP_APINVS	lfil	CSR	explicit
LP_DIAG	none	CSR	explicit
LP_TRIDIAG	none	CSR	implicit
LP_SOR	omega, iterations	CSR	implicit
LP_SSOR	omega, iterations	CSR	implicit
LP_GMRES	restart, tolerance	CSR	implicit

Table 6.2: Local preconditioners.

LP_LU is an LU factorization with pivoting. LP_INVERSE is an exact inverse computed via LU factorization with pivoting. LP_RILUK is level-based relaxed incomplete LU factorization. LP_ILUT is a threshold-based ILU with control over the number of fill-ins [127], which may be better for indefinite blocks. The local preconditions prefixed with LP_APINV are approximate inverse techniques.

LP_DIAG is a diagonal approximation to the inverse, using the diagonal of the original block, and LP_TRIDIAG is a tridiagonal implicit approximation, ignoring all elements outside the tridiagonal band of the original block. LP_SVD uses the singular value decomposition $X = U\Sigma V^T$ to produce a dense approximate inverse $X^{-1} \approx V\bar{\Sigma}^{-1}U^T$, where $\bar{\Sigma}$ is Σ with its singular values thresholded by $\alpha_1\sigma_1 + \alpha_2$, a constant α_2 plus a factor α_1 of the largest singular value σ_1. This may produce a more stable incomplete factorization if there are many blocks to be inverted that are close to being singular [150]. LP_SOR, LP_SSOR and LP_GMRES are iterative methods used as local preconditioners.

6.2.3 Interface with iterative methods

An object-oriented preconditioned iterative method requires that matrix and preconditioner objects define a small number of operations. In BPKIT, these operations are defined polymorphically, and are listed in Table 6.3.

For left and right preconditionings, the functions `apply` and `applyt` may be used to apply the preconditioning operator (M^{-1}, or its transpose) on a vector. *Split* (also called *two-sided,* or *symmetric*) preconditionings use `applyl` and `applyr` to apply the left and right parts of the split preconditioner, respectively. For an incomplete factorization $A \approx LU$, `applyl` is the L^{-1} operation, and `applyr` is the U^{-1} operation. To anticipate all possible functionality, the `applyc` function defines a combined matrix-preconditioner operator to be used, for example, to implement the Eisenstat trick [64]. If the Eisenstat trick is used with flexible preconditionings (described at the end of this section), the right preconditioner `apply` also needs to be used.

Two functions not listed here are matrix member functions that return the row and column dimensions of the matrix, which are useful for the iterative method code to help preallocate any work-space that is needed.

Not all the operations in Table 6.3 may be defined for all matrix and preconditioner objects, and many iterative methods do not require all these operations. The GMRES iterative method, for example, does not require the transposed operations, and the relaxation preconditioners usually do not define the split operations. This is a case where we violate an object-oriented programming paradigm, and give the parent classes all the specializations of their children (e.g., a specific precon-

ditioner may not define `applyl` although the generic preconditioner does). This will be seen again in Section 6.3.2.

Matrix operations	
`mult`	matrix-vector product
`trans_mult`	transposed matrix-vector product
Preconditioner operations	
`apply`	apply preconditioner
`applyt`	apply transposed preconditioner
`applyl`	apply left part of a split preconditioner
`applylt`	above, transposed
`applyr`	apply right part of a split preconditioner
`applyrt`	above, transposed
`applyc`	apply a combined matrix-preconditioner operator
`applyct`	above, transposed

Table 6.3: Operations required by iterative methods.

The argument lists for the functions in Table 6.3 use fundamental data types so that iterative solver codes are not forced to adopt any particular data structure for vectors. The interfaces use blocks of vectors to support iterative methods that use multiple right-hand sides. The implementation of these operations use Level 3 BLAS when possible. All the interfaces have the form:

```
void mult(int nr, int nc, const double *u, int ldu, double* v, int ldv) const;
```

where `nr` and `nc` are the row and column dimensions of the (input) blocks of vectors, `u` and `v` are arrays containing the values of the input and output vectors, respectively, and `ldu` and `ldv` are the leading dimensions of these respective arrays. The preconditioner operations are not defined as `const` functions, in case the preconditioner objects need to change their state as the iterations progress (and spectral information is revealed, for example).

When a non-constant operator is used in the preconditioning, a flexible iterative method such as FGMRES [125] must be used. In BPKIT, this arises whenever GMRES is used as a local preconditioner. Users may wish to write advanced preconditioners that work *with* the iterative methods, and which change, for example, when there is a lack of convergence. This is a simple way of enhancing the robustness of iterative methods. In this case, the iterative method should be written as a class function whose class also provides information about convergence history and possibly approximate spectral information [149].

6.2.4 Fortran 77 interface

Many scientific computing users are unfamiliar with C++. It is usually possible, however, to provide an interface which is callable from any other language. BPKIT provides an object-oriented type of Fortran 77 interface. Objects can be created, and pointers to them are passed through functions as Fortran 77 integers. Consider the following code excerpt (most of the parameters are not important to this description):

```
call blockmatrix(bmat, n, a, ja, ia, num_block_rows, partit, btype)
call preconditioner(precon, bmat, BJacobi, 0.d0, 0.d0, LP_LU, 0.d0, 0.d0)
call flexgmres(bmat, sol, rhs, precon, 20, 600, 1.d-8)
```

The call to `blockmatrix` above creates a block matrix from the compressed sparse row data structure, given a number of arguments. This "wrapper" function is actually written in C++, but all its arguments are available to a Fortran 77 program. The integer `bmat` is actually a pointer to a block matrix object in C++. The Fortran 77 program is not meant to interpret this variable, but to pass it to other functions, such as `preconditioner` which defines a block preconditioner with a number of arguments, or `flexgmres` which solves a linear system using flexible GMRES. Similarly, `precon` is a pointer to a preconditioner object. The constant parameters `BJacobi` and `LP_LU` are used to specify a block Jacobi preconditioner, using LU factorization to solve with the diagonal blocks.

The matrix-vector product and preconditioner operations of Table 6.3 also have "wrapper" functions. This makes it possible to use BPKIT from an iterative solver written in Fortran 77. This was also another motivation to use fundamental types to specify vectors in the interface for operations such as `mult` (see Section 6.2.3).

Calling Fortran 77 from C++ is also possible, and this is done in BPKIT when it calls underlying libraries such as the BLAS. BPKIT illustrates how we were able to mix the use of different languages.

6.3 Local matrix objects

A block matrix may contain blocks of more than one type. The best choice for the types of the blocks depends mostly on the structure of the matrix, but may also depend on the proposed algorithms and the computer architecture. For example, if a matrix has been reordered so that its diagonal blocks are all diagonal, then a diagonal storage scheme for the diagonal blocks is best. Inversion of

these blocks would automatically use the appropriate algorithm. (The diagonal block type and the local preconditioners for it would have to be added by the user.)

To handle different block types the same way, instances of each type are implemented as C++ polymorphic objects (i.e., a set of related objects whose functions can be called without knowing the exact type of the object). The block types are derived from a *local matrix* class called `LocalMat`, a class that defines the common interface for all the block types. The global preconditioners refer to `LocalMat` objects. When `LocalMat` functions are called, the appropriate code is executed, depending on the actual type of the `LocalMat` object (e.g., `DENSE` or `CSR`).

In addition, each block type has a variety of local preconditioners. The explicitness or implicitness of local preconditioners need to be transparent, since, for example, either can be used in block SSOR. Thus both types of preconditioners are derived from the same base class. In particular, local preconditioners for a given block type are derived from the base class which is that block type (e.g., the `LP_SVD` local preconditioner for the `DENSE` type is derived from the `DENSE` block type). This gives the user the flexibility to treat explicit local preconditioners as regular blocks.

Implicit local preconditioners are not derived separately because logically they are related to explicit local preconditioners. All block operations that apply to explicit preconditioners also apply to local preconditioners; however, many of these operations are inefficient for local preconditioners, and their use has been disallowed to prevent improper usage. Implicit preconditioners cannot be derived separately from explicit preconditioners because of their similarity from the point of view of global preconditioners. The `LocalMat` hierarchy is illustrated in Figure 6.1, showing the derivation of block types and the subsequent derivation of local preconditioners.

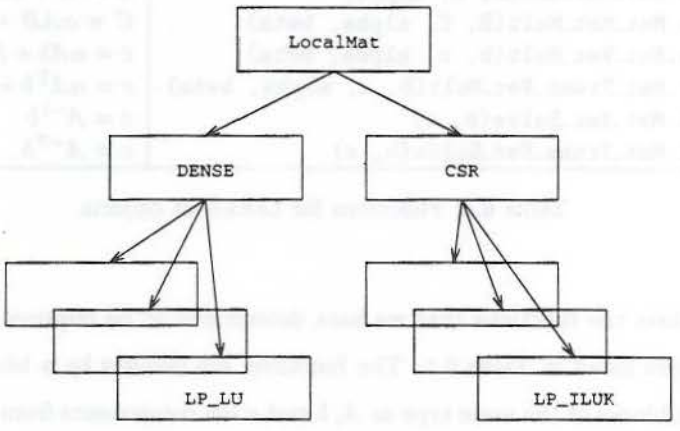


Figure 6.1: `LocalMat` hierarchy.

These `LocalMat` classes form the "kernel" of BPKIT, and allow global preconditioners to be implemented without knowledge of the type of blocks or local preconditioners that are used. Users may also add to the kernel by deriving their own specific classes.

The challenge of designing the `LocalMat` class was to determine what operations are required to implement block preconditioners and to give these operations semantics that allow an efficient implementation for all possible block types. The operations are implemented as C++ virtual functions. The following subsections describe these operations.

6.3.1 Allocating storage

An important difference between dense and sparse blocks is that the storage requirement for sparse blocks is not always known beforehand. Thus, in order to treat dense and sparse blocks the same way, storage is allocated for a block when it is required. As an optimization, if it is known that dense blocks are used (e.g., conversion of a sparse matrix to a block matrix with dense blocks), storage may be allocated beforehand by the user. Functions are provided to set the data pointers of the block objects. Thus it is possible to allocate contiguous storage for an *array* of dense blocks.

6.3.2 Local matrix functions

`B = A.CreateEmpty()`	$B = [\ ]$
`A.SetToZero(dim1,dim2)`	$A = 0$
`A.MatCopy(B)`	$A = B$
`B = A.CreateInv(lprecon)`	$B = \tilde{A}^{-1}$
`A.Mat_Trans(B)`	$B = A^T$
`A.Mat_Mat_Add(B, C, alpha)`	$C = A + \alpha B$
`A.Mat_Mat_Mult(B, C, alpha, beta)`	$C = \alpha AB + \beta C$
`A.Mat_Vec_Mult(b, c, alpha, beta)`	$c = \alpha Ab + \beta c$
`A.Mat_Trans_Vec_Mult(b, c, alpha, beta)`	$c = \alpha A^T b + \beta c$
`A.Mat_Vec_Solve(b, c)`	$c = A^{-1}b$
`A.Mat_Trans_Vec_Solve(b, c)`	$c = A^{-T}b$

Table 6.4: Functions for `LocalMat` objects.

Table 6.4 lists the functions that we have determined to be required for implementing the block preconditioners listed in Table 6.1. The functions are invoked by a block object represented by A. B and C are blocks of the same type as A, b and c are components from a block vector object, and α and β are scalars. The default value for α is 1 and for β is 0.

`CreateEmpty()` creates an empty block (0 by 0 dimensions) of the same class as that of `A`. This function is useful for constructing blocks in the preconditioner without knowing the types of blocks that are being used. `SetToZero(dim1, dim2)` sets `A` to zero, resetting its dimensions if necessary. This operation is not combined with `CreateEmpty()` because it is not always necessary to zero a block when creating it, and zeroing a block could be relatively expensive for some block types. `MatCopy(B)` copies its argument block to the invoking block. The original data held by the invoking block is released, and if the new block has a different size, the allocated space is resized. `CreateInv(lprecon)` provides a common interface for creating local preconditioners. `lprecon` is of a type that describes a local preconditioner with its arguments from Table 6.2. The exact or approximate inverse (explicit or implicit) of `A` is generated. The `CreateEmpty` and `CreateInv` functions create new objects (not just the real data space).

Overloading of the arithmetic operators such as + for blocks and local preconditioners has been sacrificed since chained operations such as $C = \alpha AB + \beta C$ would be inefficient if implemented as a sequence of elementary operations. In addition, these operators are difficult to implement without extra memory copying (for $A = B + C$, the + operator will first store the result into a temporary before the result is copied into A by the = operator).

These are the functions that we have found to be useful for block preconditioners. For example, $C = A + \alpha B$ is used in `BTIF`, $C = \alpha AB + \beta C$ is used in `BILUK`, and other functions are useful, for example, in matrix-vector product and triangular solve operations. Note in particular that `Mat_Trans_Mat_Mult` is not a useful function here, and has not been defined.

Note that local preconditioner objects also inherit these functions, although they do not need them all. For objects that are *implicit* local preconditioners, no matrix is formed, and operations such as addition (`Mat_Mat_Add`) do not make sense. For blocks for which no local preconditioner has been created, solving a system with that block (`Mat_Vec_Solve`) is not allowed. Here, again, we had to give the parent classes all the specializations of their derived classes. Table 6.5 indicates when the functions are allowed. An error condition is raised at run-time if the functions are used incorrectly.

Given these operations, a one-step block SOR code could be implemented as in Figure 6.2. `Ap` is a pointer to a block matrix object which stores its block structure in `CSR` format (the `ia` array stores the block row pointers, and the `ja` array stores the block column indices). The pointers to the diagonal elements in `idiag` and the inverses of the diagonal elements `diag` were computed during the call to `setup`. `V` is a block vector object that allows blocks in a vector to be accessed as individual entries. The rest of the code is self-explanatory.

Function	Coarse blocks	Explicit local precon.	Implicit local precon.
CreateEmpty	*	*	
SetToZero	*	*	
MatCopy	*	*	
CreateInv	*	*	
Mat_Trans	*	*	
Mat_Mat_Add	*	*	
Mat_Mat_Mult	*	*	
Mat_Vec_Mult	*	*	
Mat_Trans_Vec_Mult	*	*	
Mat_Vec_Solve		*	*
Mat_Trans_Vec_Solve		*	*

Table 6.5: The types of objects that may be used with each function.

```
for (i=0; i<Ap->numrow(); i++)
{
    for (j=ia[i]; j<idiag[i]; j++)
    {
        // V(i) = V(i) - omega * a[j] * V(ja[j])

        Ap->val(j).Mat_Vec_Mult(V(ja[j]), V(i), -omega, 1.0);
    }

    diag[i]->Mat_Vec_Solve(V(i), V(i));
}
```

Figure 6.2: Block SOR code fragment.

A block matrix that mixes different block types must be used very carefully. First, the restrictions for the different block types (Section 6.2.2) must not be violated. Second, unless we define arithmetic operations between blocks of different types, the incomplete factorization preconditioners cannot be used.

Our main design alternative was to create a block matrix class for each block type. The classes would be polymorphic and define a set of common operations that preconditioners may use to manipulate their blocks. A significant advantage of this design is that it is impossible to use local preconditioners of the wrong type (e.g., use incomplete factorization on a dense block). A disadvantage is that different block types (e.g., specialized types created for a particular application) cannot be used within the same block matrix.

Another alternative was to implement meta-matrices, i.e., blocks are nested recursively. It would be complicated, however, for users to specify these types of matrices and the levels of local preconditioners that could be used. In addition, there is very little need for such complexity in actual applications, and the two-level design (coarse and fine blocks) described in Section 6.2.1 should be sufficient.

6.4 Examples

Tables 6.6 and 6.8 show the results for SHERMAN1 with the block relaxation and incomplete factorization global preconditioners, using various local preconditioners. The arguments given for the global and local preconditioners in these tables correspond to those displayed in Tables 6.1 and 6.2 respectively. A block size of 100 was used. Since the matrix is block tridiagonal, `BILUK` and `BTIF` are equivalent. The tables show the number of steps of GMRES (FGMRES, if appropriate) that were required to reduce the residual norm by a factor of 10^{-8}. A dagger (†) is used to indicate that this was not achieved in 600 steps. Right preconditioning, 20 Krylov basis vectors and a zero initial guess were used. The right-hand side was provided with the matrix.

Since the local preconditioners have different costs, Tables 6.7 and 6.8 show the CPU timings (system and user times) for `BSSOR(1.,3)` and `BTIF`. The tests were run on one processor of a Sun Sparcstation 10. For this particular problem and choice of partitioning, the ILU local preconditioners required the least total CPU time with `BSSOR(1.,3)`. With `BTIF`, an exact solve was most efficient (i.e., the preconditioner was an exact solve).

Table 6.9 shows the number of GMRES steps for the BARTHT2A matrix. A random right-

	BJacobi	BSOR(1.,1)	BSOR(1.,3)	BSSOR(1.,1)	BSSOR(1.,3)
LP_INVERSE	88	40	17	24	13
LP_RILUK(0,0.)	93	71	53	402	48
LP_RILUK(1,0.)	89	43	24	41	20
LP_ILUT(2,0.)	85	63	46	319	44
LP_TRIDIAG	138	115	98	†	99
LP_SOR(1.,1)	†	541	†	†	491
LP_SSOR(1.,1)	508	408	395	†	411
LP_GMRES(150,0.1)	91	45	21	36	19
LP_APINVS(5)	108	70	57	274	56
LP_APINV0	171	130	122	†	121

Table 6.6: Number of GMRES steps for solving the SHERMAN1 problem with block relaxation global preconditioners and various local preconditioners.

	BSSOR(1.,3)	CPU time (s)		
		precon	solve	total
LP_INVERSE	13	0.63	3.86	4.49
LP_RILUK(0,0.)	48	0.01	1.86	1.87
LP_RILUK(1,0.)	20	0.02	0.87	0.89
LP_ILUT(2,0.)	44	0.03	1.73	1.76
LP_TRIDIAG	99	0.01	3.87	3.88
LP_SOR(1.,1)	491	0.00	20.80	20.80
LP_SSOR(1.,1)	411	0.00	19.40	19.40
LP_GMRES(150,0.1)	19	0.00	29.23	29.23
LP_APINVS(5)	56	0.27	2.48	2.75
LP_APINV0	121	0.51	5.21	5.72

Table 6.7: Number of GMRES steps and timings for solving the SHERMAN1 problem with BSSOR(1.,3) and various local preconditioners.

	BTIF	CPU time (s)		
		precon	solve	total
LP_INVERSE	1	1.44	0.15	1.59
LP_DIAG	†	0.01	†	†
LP_APINV0	123	0.52	3.73	4.25
LP_APINVS(5)	64	0.32	1.98	2.30
LP_APINVS(10)	33	1.14	1.08	2.22

Table 6.8: Number of GMRES steps and timings for solving the SHERMAN1 problem with block incomplete factorization and various local preconditioners.

hand side was used, and the initial guess was zero. The GMRES tolerance was 10^{-8} and 50 Krylov basis vectors were used. In Table 6.9(a), block incomplete factorization was used as the global preconditioner, and LU factorization was used as the local preconditioner. In Table 6.9(b), block SSOR with one iteration and $\omega = 1$ was used as the global preconditioner, and level-3 ILU was used as the local preconditioner.

block size	BILUK level 0	1	2
5	436	183	130
10	184	118	95
15	141	100	94

(a) `BILUK-LP_LU`

block size	GMRES steps
60	266
120	273
240	210

(b) `BSSOR(1.,1)-LP_RILUK(3,0.)`

Table 6.9: Number of GMRES steps for solving the BARTHT2A problem.

Table 6.10 shows the results for WIGTO996 using block incomplete factorization. The right-hand side was the vector of all ones, and the GMRES tolerance was 10^{-8}. The other parameters were the same as those in the previous experiment. The failures in Table 6.10(a) are due to inaccuracy for low fill levels, and instability for high levels. In Table 6.10(b), `LP_SVD(0.1,0.)` used as the local preconditioner gave the best results. `LP_SVD(0.1,0.)` indicates that the singular values of the pivot blocks were thresholded at 0.1 times the largest singular value.

block size	BILUK level 0	1	2	3
4	†	†	†	†
8	†	†	94	75
16	†	77	400	†

(a) `BILUK-LP_INVERSE`

block size	BILUK level 0	1	2	3
4	50	42	39	35
8	44	35	32	30
16	40	36	32	30

(b) `BILUK-LP_SVD(0.1,0.)`

Table 6.10: Number of GMRES steps for solving the WIGTO966 problem.

There is often heated debate over the use of C++ in scientific computing. Ideally, C++ and Fortran 77 programs that are coded similarly should perform similarly. However, by using object-oriented features in C++ to make a program more flexible and maintainable, researchers usually encounter a 10 to 30 percent performance penalty [88]. If optimized kernels such as the BLAS are

called, then the C++ performance penalty can be very small for large problems, as a larger fraction of the time is spent in the kernels.

Since C++ and Fortran 77 programs will usually be coded differently, a practical comparison is made when a general code such as BPKIT is compared to a *specialized* Fortran 77 code. Here we compare BPKIT to an optimized block SSOR preconditioner with a GMRES accelerator. This code performs block relaxations of the form

$$\begin{aligned} \delta &\Leftarrow A_{ii}^{-1} r_i \\ x_i &\Leftarrow x_i + \delta \\ r &\Leftarrow r - A_{:,i}\delta \end{aligned}$$

for a block row i, where A_{ii} is the i-th diagonal block of A, $A_{:,i}$ is the i-th block column of A, x_i is the i-th block of the current solution, and r is the current residual vector. Notice that the update of the residual vector is very fast if A is stored by sparse columns and not by blocks. Since BPKIT stores the matrix A by blocks for flexibility, it is interesting to see what the performance penalty would be for this case.

Tables 6.11 and 6.12 show the timings for block SSOR on a Sun Sparcstation 10 and a Cray C90 supercomputer, for the WIGTO966 matrix. In this case, the right-hand side was constructed so that the solution is a vector of all ones; the other parameters were the same as before. All programs were optimized at the highest optimization level; `clock` was used to measure CPU time (user and system) for the C++ programs, and `etime` and `timef` were used to measure the times for the Fortran 77 programs on the Sun and Cray computers, respectively. One step of block SSOR with $\omega = 0.5$ was used in the tests. The local preconditioner was an exact LU factorization. Results are shown for a large range of block sizes, and in the case of BPKIT, for both `DENSE` and `CSR` storage schemes for the blocks. The last column of each table gives the average time to perform one iteration of GMRES.

The results show that the specialized Fortran 77 code has better performance over a wide range of block sizes. This is expected because the update of the residual, which is the most major computation, is not affected by the blocking.

If dense blocks are used, BPKIT can be competitive on the Cray by using large block sizes, such as 128. Blocks of this size contain many zero entries which are treated as general nonzero entries when a dense storage scheme is used. However, vectorization on the Cray makes operations

Specialized Fortran 77 program					
block	GMRES	time (s)			
size	steps	precon	solve	total	average
4	500	1.87	193.09	194.96	0.3899
8	240	1.21	91.47	92.68	0.3862
16	306	1.04	118.78	119.82	0.3916
32	300	1.21	124.41	125.63	0.4188
64	221	1.79	103.85	105.65	0.4781
128	212	3.86	124.15	128.02	0.6039

BPKIT, dense blocks					
block	GMRES	time (s)			
size	steps	precon	solve	total	average
4	500	0.25	305.97	306.22	0.6124
8	240	0.25	119.16	119.41	0.4975
16	306	0.38	193.97	194.35	0.6351
32	300	0.73	303.10	303.83	1.0128
64	221	1.27	376.02	377.29	1.7072
128	212	3.66	559.05	562.71	2.6543

BPKIT, sparse blocks					
block	GMRES	time (s)			
size	steps	precon	solve	total	average
4	500	0.24	284.69	284.93	0.5699
8	240	0.34	106.67	107.01	0.4459
16	306	0.62	129.96	130.58	0.4267
32	300	1.06	137.50	138.56	0.4619
64	221	1.87	123.34	125.21	0.5666
128	212	4.42	162.58	167.00	0.7877

Table 6.11: WIGTO966: `BSSOR(0.5,1)-LP_LU`, Sun Sparc 10 timings.

Specialized Fortran 77 program					
block	GMRES	time (s)			
size	steps	precon	solve	total	average
4	500	0.075	13.92	14.00	0.0280
8	240	0.065	5.57	5.64	0.0235
16	306	0.065	7.02	7.09	0.0232
32	300	0.089	7.00	7.09	0.0237
64	221	0.140	6.03	6.17	0.0279
128	212	0.296	7.23	7.53	0.0355

BPKIT, dense blocks					
block	GMRES	time (s)			
size	steps	precon	solve	total	average
4	500	0.115	183.92	184.04	0.3681
8	240	0.080	36.07	36.15	0.1506
16	306	0.072	26.75	26.83	0.0877
32	300	0.079	16.79	16.87	0.0563
64	221	0.127	8.24	8.37	0.0379
128	212	0.290	6.73	7.03	0.0332

BPKIT, sparse blocks					
block	GMRES	time (s)			
size	steps	precon	solve	total	average
4	500	0.27	282.32	282.59	0.5652
8	240	0.39	92.27	92.67	0.3861
16	306	0.83	106.72	107.56	0.3515
32	300	1.38	110.42	111.80	0.3727
64	221	2.44	99.15	101.59	0.4597
128	212	5.39	132.92	138.31	0.6524

Table 6.12: WIGTO966: `BSSOR(0.5,1)-LP_LU`, Cray C90 timings.

with large dense blocks much more efficient.

If sparse blocks are used, BPKIT can be competitive on the workstation with moderate block sizes of 8 or 16. Operations with smaller sparse blocks are inefficient, while larger blocks imply larger LU factorizations for the local preconditioner.

This comparison using block SSOR is dramatic since two very different data structures are used. Comparisons of level-based block ILU in C++ and Fortran 77 show very small differences in performance, since the data structures used are similar [88].

In conclusion, the types and sizes of blocks must be chosen carefully in BPKIT to attain high performance on a particular machine. The types and sizes of blocks should also be chosen in conjunction with the requirements of the preconditioning algorithm and the block structure of the matrix. Based on the above experiments, Table 6.13 gives an idea of the approximate block sizes that should be used for BPKIT, given no other constraints.

Block type	Sun	Cray
DENSE	8	128
CSR	16	16

Table 6.13: Recommended block sizes.

Chapter 7

Conclusions and future work

This thesis has presented several preconditioning techniques, discussed their implementation, and compared their performance. New techniques for approximating the inverse of a matrix A by a sparse matrix M were presented in Chapter 3. These new techniques perform *approximate* minimization of the Frobenius norm of the residual matrix $(I - AM)$, and have a much simpler strategy for dynamically selecting the sparsity pattern of the approximate inverse. Indeed, tests by independent researchers [20] have shown that this approximate method can be more economical, particularly when relatively dense approximations are desired.

For well-conditioned problems, numerical tests show that sparse approximate inverses are comparable to ILU preconditioners. Although the former are more expensive to construct, they can be constructed and applied easily on parallel computers. Sparse approximate inverses for these problems on parallel machines are superior, and therefore should be preferred over ILU preconditioners in these cases. In addition, for very unstructured matrices, particularly those that have many zero diagonal elements, ILU preconditioners are not applicable (i.e., they will fail due to zero pivots), but sparse approximate inverses may perform well. This was shown in Chapter 3, for example, for the very unstructured Westerberg chemical engineering matrices from the Harwell-Boeing collection. Unlike ILU and other preconditioners, the new methods using dynamic selection of the sparsity pattern do not depend on there being nonzero diagonal elements in the matrix.

Several problems are still open. The dynamic selection of approximate inverse sparsity patterns is effective, but computationally expensive compared to methods that use predetermined sparsity patterns. Thus the challenge is to predetermine good sparsity patterns for approximate inverses. This may be accomplished through graph theoretic ideas about the structure of inverses, particularly of triangular factorized forms. It may also be possible to select the pattern of an ILU factorization which has an accurate sparse inverse approximation.

Most sparse approximate inverses can be constructed easily in parallel. This fact, however, suggests the drawback that they are *local*, rather than *global* operators. On a PDE grid, sparse approximate inverses have the effect of only transmitting information locally, rather than across the entire PDE domain. For very large problems, the performance of these preconditioners is expected

to deteriorate. We thus investigated in Chapter 4 the application of sparse approximate inverse techniques to matrices partitioned into blocks, such as via domain decomposition. Approximate inverses are used to approximately invert blocks corresponding to a local subdomain, for example, and the subdomains are *coupled* (in order to transmit global information) via the Schur complement. It is becoming realized that these two-level and other multilevel approaches are imperative in order to robustly solve very large problems.

Specifically, we introduced in Chapter 4 a technique for finding sparse approximate solutions for sparse linear systems with sparse right-hand sides. As shown in that chapter, this technique is useful in opening new ways to construct preconditioners. Among the methods that are enhanced are incomplete factorizations of block tridiagonal matrices. The applicability of these methods was limited in the past to structured matrices where the pivot blocks have a banded pattern. General approximate inverse techniques that do not depend on a preset sparsity pattern allow these techniques to be used on block tridiagonal matrices that have general structure within their blocks. Care must be taken, for example, to ensure that the sequence of pivot blocks produced by the factorization do not become denser and denser. This is accomplished in our dynamic methods by preserving sparsity by dropping small elements. Further work is required to make this an even more robust procedure.

Another application of the approximate inverse technique that we presented in Chapter 4 is approximate block LU (ABLU) factorization. This form of preconditioning is in line with the recent trend towards mixed and fully-coupled solution techniques for multi-physics problems. When solving strongly coupled PDE's, these techniques may be faster than segregated approaches. In ABLU techniques, an outer iterative method using matrix-vector multiplies with the original matrix is used to ensure that the iteration is driven to the solution of the original problem. A flexible iterative method such as FGMRES can be used if the preconditioning operation, which can be very complicated, is not constant. The advantage of ABLU is that specialized solvers can be used for different equations describing different phenomena. Some of these equations may be diagonally dominant, for instance, and the approximation for these equations may take this information into account. Fast solvers or low-order approximations may be available for other equations in the system. In Chapter 4, approximate inverse techniques were used to find sparse approximations to the Schur complement or its inverse. The observations from that chapter show that these block preconditioners using approximate inverses are less effective at reducing the number of iterations on difficult problems than ILU preconditioners, but can require much less storage.

The usefulness of ILU preconditioners in general settings is clear. Until recently, however,

their behavior on general problems has not been properly understood. Users faced with an ILU failure had little understanding of the cause of failure, and often responded incorrectly by attempting to increase the allowable fill-in. Chapter 5 furthers the understanding of the failure modes of ILU factorizations. We introduced a number of statistics that can be monitored during or after the factorization. The most useful of these is perhaps an estimate of $\|(LU)^{-1}\|$, where LU is the incomplete LU factorization. The instability of the triangular solves, particularly for general nonsymmetric and indefinite problems, is now better recognized. This was poorly understood in the past, partially because this problem does not occur in *complete* factorizations. Also, complete factorizations are much more oriented towards preserving the stability of the factorization (i.e., growth of the elements in the factors), which is not a problem for *incomplete* factorizations. Chapter 5 also investigates several variants of ILU factorizations and shows how certain causes of failure can sometimes be avoided. A variant of ILU that produces incomplete factors that are structurally symmetric to each other was one of the methods presented.

The results in Chapter 5 can be used to design new ILU preconditioners. One conclusion from that chapter is that symmetric pivoting should be used for symmetrically structured matrices, if possible. This can be accomplished with not too much computational and storage expense due to another conclusion from that chapter: pivoting is only essential for avoiding very small pivots, rather than trying to minimize the growth in the factorization. Thus, finding a *suitable* pivot, rather than the *best* pivot, is adequate. An ILU factorization code can maintain a set of partially factored rows which can be used for symmetric pivoting. This may be combined with the use of block pivots to enhance stability, if necessary.

In Chapter 6, we addressed the dependence of efficient preconditioning codes on the data structure of the original problem. This dependence can be relieved to some extent by using object-oriented techniques to hide details of the data structures and replacing them with abstractions. We described a framework called BPKIT for block preconditioning that uses polymorphism to handle different block types. The data structure of the blocks is independent of the global preconditioner, and specialized solvers (depending on the data structure) can be used for the local equations. The block size parameterizes between a local and global method, and is valuable for compromising between accuracy and cost, or combining the effect of two methods. The combination of local and global preconditioners leads to a variety of useful methods, all of which may be applicable in different circumstances. BPKIT is a relatively popular toolkit, and is currently being evaluated by

the National High-Performance Computing Consortium Software Exchange.[1]

As preconditioners are becoming more mature, their use in certain ways is becoming better understood and more standardized. One method in this category is nonoverlapping domain decomposition using the Schur complement, which was used in Chapter 4. Various methods of solving the subdomain equations and the global Schur complement system should be available to match the preconditioner to a given problem. The BPKIT software framework described in Chapter 6 can be extended for parallel computation based on this type of domain decomposition. The selection of specialized subdomain solvers and blocking for the variables should depend on the type and location of physical phenomena being modeled over the domain. This selection may be semi-automatic to a certain extent. The framework stays constant, however, in order to minimize the software engineering cost.

[1] See http://www.nhse.org/hpc-netlib/sw_eval/linalg/sparse.html.

Bibliography

[1] F. Alvarado, H. Dağ, and M. ten Bruggencate. Block-bordered diagonalization and parallel iterative solvers. In *Prelim. Proc. Colorado Conference on Iterative Methods*, Breckenridge, CO, April 5–9, 1994.

[2] E. C. Anderson and Y. Saad. Solving sparse triangular systems on parallel computers. *Intl. J. High Speed Computing*, 1:73–96, 1989.

[3] K. Arrow, L. Hurwicz, and H. Uzawa. *Studies in Nonlinear Programming.* Stanford University Press, Stanford, CA, 1958.

[4] S. F. Ashby, T. A. Manteuffel, and P. E. Saylor. A taxonomy for conjugate gradient methods. *SIAM J. Numer. Anal.*, 27:1542–1568, 1990.

[5] O. Axelsson. Incomplete block matrix factorization preconditioning methods. The ultimate answer? *J. Comput. Appl. Math.*, 12 & 13:3–18, 1985.

[6] O. Axelsson, S. Brinkkemper, and V. P. Il'in. On some versions of incomplete block-matrix factorization iterative methods. *Lin. Alg. Appl.*, 58:3–15, 1984.

[7] O. Axelsson and L. Yu. Kolotilina. Diagonally compensated reduction and related preconditioning methods. *Num. Lin. Alg. Appl.*, 1:155–177, 1994.

[8] O. Axelsson and G. Lindskog. On the eigenvalue distribution of a class of preconditioning methods. *Numer. Math.*, 48:479–498, 1986.

[9] O. Axelsson and G. Lindskog. On the rate of convergence of the preconditioned conjugate gradient method. *Numer. Math.*, 48:499–523, 1986.

[10] O. Axelsson and B. Polman. On approximate factorization methods for block matrices suitable for vector and parallel processors. *Lin. Alg. Appl.*, 77:3–26, 1986.

[11] Owe Axelsson. *Iterative Solution Methods.* Cambridge University Press, Cambridge, 1994.

[12] S. T. Barnard and R. Clay. A portable MPI implementation of the SPAI preconditioner in ISIS++. In *Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, March, 1997.

[13] T. Barth, T. F. Chan, and W.-P. Tang. An algebraic nonoverlapping domain decomposition method for convection-diffusion problems. Technical report, NASA Ames Research Center, Moffett Field, CA, 1997.

[14] M. W. Benson. Iterative solution of large scale linear systems. Master's thesis, Lakehead University, Thunder Bay, ON, 1973.

[15] M. W. Benson. Frequency domain behavior of a set of parallel multigrid smoothing operators. *Intl. J. Comp. Math.*, 36:77–88, 1990.

[16] M. W. Benson and P. O. Frederickson. Iterative solution of large sparse linear systems arising in certain multidimensional approximation problems. *Utilitas Math.*, 22:127–140, 1982.

[17] M. W. Benson, J. Krettmann, and M. Wright. Parallel algorithms for the solution of certain large sparse linear systems. *Intl. J. Comp. Math.*, 16:245–260, 1984.

[18] M. Benzi, C. D. Meyer, and M. Tůma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM J. Sci. Comput.*, 17:1135–1149, 1996.

[19] M. Benzi, D. B. Szyld, and A. van Duin. Orderings for incomplete factorization preconditioning of nonsymmetric problems. Technical Report LA-UR-97-3525, Los Alamos National Laboratory, Los Alamos, NM, 1997.

[20] M. Benzi and M. Tůma. Sparse approximate inverse preconditioners: a comparative study. Technical report, CERFACS, Toulouse, France, 1997.

[21] M. Benzi and M. Tůma. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM J. Sci. Comput.*, to appear, 1998.

[22] C. H. Bischof, J. G. Lewis, and D. J. Pierce. Incremental condition estimation for sparse matrices. *SIAM J. Matrix Anal. Appl.*, 11:644–659, 1990.

[23] E. F. F. Botta, A. van der Ploeg, and F. W. Wubs. Nested grids ILU-decomposition (NGILU). *J. Comput. Appl. Math.*, 66:515–526, 1996.

[24] E. F. F. Botta and F. W. Wubs. MRILU: it's the preconditioning that counts. Technical Report W-9703, Department of Mathematics, University of Groningen, Groningen, 1997.

[25] M. E. Braaten and S. V. Patankar. Fully coupled solution of the equations for incompressible recirculating flows using a penalty-function finite-difference formulation. *Computational Mechanics*, 6:143–155, 1990.

[26] R. Bramley and T. Loos. EMILY: A visualization tool for large sparse matrices. Technical Report 412 A, Computer Science Department, Indiana University–Bloomington, Bloomington, IN, 1996. Software available at ftp://ftp.cs.indiana.edu/pub/tloos/emily.

[27] F. Brezzi and J. Pitkäranta. On the stabilization of finite element approximations of the Stokes equations. In W. Hackbush, editor, *Efficient Solutions of Elliptic Systems, Notes on Numerical Fluid Mechanics*, volume 10. Braunschweig Wiesbaden, Vieweg, 1984.

[28] R. Bridson and W.-P. Tang. An ordering method for a factorized approximate inverse preconditioner. Technical report, Department of Computer Science, University of Waterloo, Waterloo, ON, in preparation.

[29] A. Brooks and T. J. R. Hughes. Streamline upwind/Petrov-Galerkin formulation for convection dominated flows. *Computer Methods in Applied Mechanics and Engineering*, 32:199–259, 1982.

[30] P. N. Brown and A. C. Hindmarsh. Matrix-free methods for stiff systems of ODEs. *SIAM J. Numer. Anal.*, 23:610–638, 1986.

[31] A. M. Bruaset and H. P. Langtangen. Object-oriented design of preconditioned iterative methods in Diffpack. *ACM Trans. Math. Softw.*, 23:50–80, 1997.

[32] A. M. Bruaset, A. Tveito, and R. Winther. On the stability of relaxed incomplete LU factorizations. *Math. Comp.*, 54:701–719, 1990.

[33] N. I. Buleev. A numerical method for the solution of two-dimensional and three-dimensional equations of diffusion. *Math. Sb.*, 51:227–238, 1960. English transl.: Rep. BNL-TR-551, Brookhaven National Laboratory, Upton, New York, 1973.

[34] J. R. Bunch and L. Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Math. Comp.*, 31:162–179, 1977.

[35] G. F. Carey, A. I. Pehlivanov, Y. Shen, A. Bose, and K. C. Wang. Least-squares finite elements for fluid flow and transport. *Intl. J. Num. Meth. Fluids*, to appear, 1997.

[36] S. Carney, M. A. Heroux, G. Li, and K. Wu. A revised proposal for a sparse BLAS toolkit. Technical Report 94-034, Army High Performance Computing Research Center, Minneapolis, MN, 1994.

[37] T. F. Chan, E. Chow, Y. Saad, and M. C. Yeung. Preserving symmetry in preconditioned Krylov subspace methods. *SIAM J. Sci. Comput.*, to appear, 1998.

[38] T. F. Chan, W.-P. Tang, and W. L. Wan. Wavelet sparse approximate inverse preconditioners. *BIT*, 3:644–660, 1997.

[39] T. F. Chan and H. A. Van der Vorst. Approximate and incomplete factorizations. Technical Report 871, Department of Mathematics, University of Utrecht, 1994.

[40] A. Chapman and Y. Saad. Deflated and augmented krylov subspace techniques. *Num. Lin. Alg. Appl.*, pages 43–66, 1997.

[41] A. Chapman, Y. Saad, and L. Wigton. High-order ILU preconditioners for CFD problems. Technical Report UMSI 96/14, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 1996.

[42] E. Chow and M. A. Heroux. An object-oriented framework for block preconditioning. *ACM Trans. Math. Softw.*, to appear, 1998.

[43] E. Chow and Y. Saad. Approximate inverse techniques for block-partitioned matrices. *SIAM J. Sci. Comput.*, 18:1657–1675, 1997.

[44] E. Chow and Y. Saad. Experimental study of ILU preconditioners for indefinite matrices. *J. Comput. Appl. Math.*, 85, 1997.

[45] E. Chow and Y. Saad. ILUS: an incomplete LU preconditioner in sparse skyline format. *Intl. J. Num. Meth. Fluids*, 25:739–748, 1997.

[46] E. Chow and Y. Saad. Approximate inverse preconditioners via sparse-sparse iterations. *SIAM J. Sci. Comput.*, 19, 1998.

[47] R. L. Clay. ISIS++: Iterative scalable implicit solver (in C++). Sandia National Laboratories, Livermore, CA, 1997.

[48] S. S. Clift and W.-P. Tang. Weighted graph based ordering techniques for preconditioned conjugate gradient methods. *BIT*, 35:30–47, 1995.

[49] P. Concus, G. H. Golub, and G. Meurant. Block preconditioning for the conjugate gradient method. *SIAM J. Sci. Statist. Comput.*, 6:309–332, 1985.

[50] J. D. F. Cosgrove, J. C. Díaz, and A. Griewank. Approximate inverse preconditioning for sparse linear systems. *Intl. J. Comp. Math.*, 44:91–110, 1992.

[51] E. F. D'Azevdo, P. A. Forsyth, and W.-P. Tang. Towards a cost-effective ILU preconditioner with high level fill. *BIT*, 32:442–463, 1992.

[52] J. Demmel. LAPACK: A portable linear algebra library for supercomputers. In *Proc. 1989 IEEE Control Systems Society Workshop on Computer-Aided Control System Design*, December 1989.

[53] V. Deshpande, M. J. Grote, P. Messmer, and W. Sawyer. Parallel implementation of a sparse approximate inverse preconditioner. In A. Ferreira, J. Rolim, Y. Saad, and T. Yang, editors, *Parallel Algorithms for Irregularly Structured Problems (Proc. IRREGULAR '96)*, pages 63–74, Santa Barbara, CA, 1996.

[54] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16:1–17, 1990.

[55] J. J. Dongarra, A. Lumsdaine, X. Niu, R. Pozo, and K. Remington. A sparse matrix library in C++ for high performance architectures. In *Proc. Object Oriented Numerics Conference*, Sun River, OR, 1994.

[56] J. J. Dongarra, R. Pozo, and D. W. Walker. An object oriented design for high performance linear algebra on distributed memory architectures. In *Proc. Object Oriented Numerics Conference*, Snowbird, CO, 1993.

[57] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, New York, 1989.

[58] I. S. Duff, N. I. M. Gould, J. K. Reid, and J. A. Scott. The factorization of sparse indefinite matrices. *IMA J. Num. Anal.*, 11:181–204, 1991.

[59] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Softw.*, 15:1–14, 1989.

[60] I. S. Duff and G. A. Meurant. The effect of ordering on preconditioned conjugate gradients. *BIT*, 29:635–657, 1989.

[61] L. C. Dutto. The effect of ordering on preconditioned GMRES algorithms, for solving the compressible Navier-Stokes equations. *Intl. J. Num. Meth. Engrg.*, 36:457–497, 1993.

[62] L. C. Dutto, W. G. Habashi, and M. Fortin. Parallelizable block diagonal preconditioners for the compressible Navier-Stokes equations. *Computer Methods in Applied Mechanics and Engineering*, 117:15–47, 1994.

[63] V. Eijkhout and T. F. Chan. ParPre: A parallel preconditioners package. Technical Report CAM 97-24, Department of Mathematics, University of California, Los Angeles, CA, 1997.

[64] S. C. Eisenstat. Efficient implementation of a class of preconditioned conjugate gradient methods. *SIAM J. Sci. Statist. Comput.*, 2:1–4, 1981.

[65] H. C. Elman. A stability analysis of incomplete LU factorizations. *Math. Comp.*, 47:191–217, 1986.

[66] H. C. Elman. Relaxed and stabilized incomplete factorizations for non-self-adjoint linear systems. *BIT*, 29:890–915, 1989.

[67] M. S. Engelman. *FIDAP: Examples Manual, Revision 6.0.* Fluid Dynamics International, Evanston, IL, 1991.

[68] M. S. Engelman and I. Hasbani. Matrix-free solution algorithms in a finite element context. Technical Report 88-1, Fluid Dynamics International, Evanston, IL, 1988.

[69] Q. Fan, P. A. Forsyth, J. R. F. McMacken, and W.-P. Tang. Performance issues for iterative solvers in device simulation. *SIAM J. Sci. Comput.*, 17:100–117, 1996.

[70] M. R. Field. An efficient parallel preconditioner for the conjugate gradient algorithm. Technical Report HDL-TR-97-175, Hitachi Dublin Laboratory, Trinity College, Dublin, 1997.

[71] M. Fortin. Finite element solution of the Navier-Stokes equations. In *Acta Numerica 1993*, pages 239–284. Cambridge University Press, Cambridge, 1993.

[72] P. O. Frederickson. Fast approximate inversion of large sparse linear systems. Technical Report 7-75, Lakehead University, Thunder Bay, ON, 1975.

[73] M. Gates and A. Cleary. Private communication, 1997.

[74] Alan George and Joseph W. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice Hall, Englewood Cliffs, NJ, 1981.

[75] U. Ghia, K. N. Ghia, and C. T. Shin. High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *J. Comput. Phys.*, 48:387–411, 1982.

[76] J. R. Gilbert and S. Toledo. An assessment of incomplete-LU preconditioners for nonsymmetric linear systems. Technical report, Xerox Palo Alto Research Center, Palo Alto, CA, 1997.

[77] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, MD, 3rd edition, 1996.

[78] N. I. M. Gould and J. A. Scott. On approximate-inverse preconditioners. Technical Report RAL-95-026, Rutherford Appleton Laboratory, Chilton, England, 1995.

[79] M. Grote and T. Huckle. Effective parallel preconditioning with sparse approximate inverses. In *Proc. Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 466–471, San Francisco, CA, 1994.

[80] M. Grote and T. Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM J. Sci. Comput.*, 18:838–853, 1997.

[81] M. Grote and H. D. Simon. Parallel preconditioning and approximate inverses on the Connection Machine. In R. F. Sincovec, D. E. Keyes, L. R. Petzold, and D. A. Reed, editors, *Parallel Processing for Scientific Computing*, volume 2, pages 519–523, SIAM, Philadelphia, PA, 1993.

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

[83] V. Haroutunian, M. S. Engelman, and I. Hasbani. Segregated finite element algorithms for the numerical solution of large-scale incompressible flow problems. *Intl. J. Num. Meth. Fluids*, 17:323–348, 1993.

[84] Y. Hasbani and M. S. Engelman. Out-of-core solution of linear equations with non-symmetric coefficient matrix. *Computers & Fluids*, 7:13–31, 1979.

[85] A. S. Householder. *The Theory of Matrices in Numerical Analysis*. Dover, New York, 1964.

[86] T. Huckle. Efficient computation of sparse approximate inverses. Technical Report 342/04/96, Technische Universität München, München, Germany, 1996.

[87] A. Jennings and G. M. Malik. Partial elimination. *J. Inst. Maths. Applics.*, 20:307–316, 1977.

[88] H. Jiang and P. A. Forsyth. Robust linear and nonlinear strategies for solution of the transonic Euler equations. *Computers & Fluids*, 24:753–770, 1995.

[89] M. T. Jones and P. E. Plassmann. BlockSolve95 users manual: Scalable library software for the parallel solution of sparse linear systems. Technical Report ANL-95/48, Argonne National Laboratory, Argonne, IL, 1995.

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

[91] G. Karypis and V. Kumar. Parallel threshold-based ILU factorization. Technical Report 96-061, Computer Science Department, University of Minnesota, Minneapolis, MN, 1996.

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

[93] D. E. Keyes and W. D. Gropp. A comparison of domain decomposition techniques for elliptic partial differential equations and their parallel implementation. *SIAM J. Sci. Statist. Comput.*, 8:s166–s202, 1987.

[94] L. Yu. Kolotilina. Explicit preconditioning of H-matrices. In *Numerical analysis and mathematical modeling*, pages 97–108, 1989.

[95] L. Yu. Kolotilina, I. E. Kaporin, and A. Yu. Yeremin. Block SSOR preconditionings for high-order 3D FE systems. Incomplete BSSOR preconditionings. *Lin. Alg. Appl.*, 154–156:647–674, 1991.

[96] L. Yu. Kolotilina, A. A. Nikishin, and A. Yu. Yeremin. Factorized sparse approximate inverse (FSAI) preconditionings for solving 3D FE systems on massively parallel computers. II. Iterative construction of FSAI preconditioners. In R. Beauwens and P. de Groen, editors, *Proc. IMACS Intl. Symp. Iterative Methods in Linear Algebra*, pages 311–312, Brussels, Belgium, 1992.

[97] L. Yu. Kolotilina and A. Yu. Yeremin. On a family of two-level preconditionings of the incomplete block factorization type. *Soviet J. Numer. Anal. Math. Model.*, 1:293–320, 1986.

[98] L. Yu. Kolotilina and A. Yu. Yeremin. Factorized sparse approximate inverse preconditionings I. Theory. *SIAM J. Matrix Anal. Appl.*, 14:45–58, 1993.

[99] L. Yu. Kolotilina and A. Yu. Yeremin. Incomplete block factorizations as preconditioners for sparse SPD matrices. In G. Golub, M. Lorkin, and A. Greenbaum, editors, *Recent Advances in Iterative Methods*, pages 119–133, Berlin, 1994. Springer-Verlag.

[100] L. Yu. Kolotilina and A. Yu. Yeremin. Factorized sparse approximate inverse preconditionings II. Solution of 3D FE systems on massively parallel computers. *Intl. J. High Speed Computing*, 7:191–215, 1995.

[101] C.-J. Lin and J. J. Moré. Incomplete Cholesky factorizations with limited memory. Technical Report MCS-P682-0897, Argonne National Laboratory, Argonne, IL, 1997.

[102] S. Ma and Y. Saad. Distributed ILU(0) and SOR preconditioners for unstructured sparse linear systems. Technical Report 94-027, Army High Performance Computing Research Center, Minneapolis, MN, 1994.

[103] L. Machiels and M. O. Deville. Fortran 90: An entry to object-oriented programming for solution of partial differential equations. *ACM Trans. Math. Softw.*, 23:32–49, 1997.

[104] H. Manouzi. Private communication, 1993.

[105] T. A. Manteuffel. An incomplete factorization technique for positive definite linear systems. *Math. Comp.*, 34:473–497, 1980.

[106] J. A. Meijerink and H. A. Van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Math. Comp.*, 31:148–162, 1977.

[107] J. A. Meijerink and H. A. Van der Vorst. Guidelines for the usage of incomplete decompositions in solving sets of linear equations as they occur in practical problems. *J. Comput. Phys.*, 44:134–155, 1981.

[108] A. Meyer. The concept of special inner products for deriving new conjugate gradient-like solvers for non-symmetric sparse linear systems. *Num. Lin. Alg. Appl.*, 1:129–140, 1994.

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

[110] Y. Notay and Z. O. Amar. Incomplete factorization preconditioning may lead to multigrid like speed of convergence. In A. S. Alekseev and N. S. Bakhvalov, editors, *Advanced Mathematics: Computations and Applications*, pages 435–446. NCC Publisher, 1995.

[111] T. A. Oliphant. An implicit numerical method for solving two-dimensional time-dependent diffusion problems. *Quart. Appl. Math.*, 19:221–229, 1961.

[112] T. A. Oliphant. An extrapolation process for solving linear systems. *Quart. Appl. Math.*, 20:257–267, 1962.

[113] J. O'Neil and D. B. Szyld. A block ordering method for sparse matrices. *SIAM J. Sci. Statist. Comput.*, 11:811–823, 1990.

[114] H. L. Ong. Fast approximate solution of large scale sparse linear systems. *J. Comput. Appl. Math.*, 10:45–54, 1984.

[115] J. M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems.* Plenum Press, New York, 1988.

[116] O. Østerby and Z. Zlatev. *Direct Methods for Sparse Matrices*, volume 157 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1983.

[117] V. Pan and J. Reif. Efficient parallel solution of linear systems. In *Proc. 17th Annual ACM Symposium on Theory of Computing*, pages 143–152, 1985.

[118] V. Pan and R. Schreiber. An improved Newton iteration for the generalized inverse of a matrix, with applications. *SIAM J. Sci. Statist. Comput.*, 12:1109–1130, 1991.

[119] S. V. Parter. The use of linear graphs in Gauss elimination. *SIAM Rev.*, 3:119–130, 1961.

[120] S. V. Patankar. *Numerical Heat Transfer and Fluid Flow.* Hemisphere, Washington, DC, 1980.

[121] K. A. Remington and R. Pozo. NIST sparse BLAS user's guide. Technical report, National Institute of Standards and Technology, Gaithersburg, MD, 1996.

[122] Y. Robert. Regular incomplete factorizations of real positive definite matrices. *Lin. Alg. Appl.*, 48:105–117, 1982.

[123] Y. Saad. Preconditioning techniques for indefinite and nonsymmetric linear systems. *J. Comput. Appl. Math.*, 24:89–105, 1988.

[124] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA, 1990.

[125] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Statist. Comput.*, 14:461–469, 1993.

[126] Y. Saad. Highly parallel preconditioners for general sparse matrices. In G. Golub, M. Lorkin, and A. Greenbaum, editors, *Recent Advances in Iterative Methods*, pages 165–199. Springer-Verlag, Berlin, 1994.

[127] Y. Saad. ILUT: A dual threshold incomplete ILU factorization. *Num. Lin. Alg. Appl.*, 1:387–402, 1994.

[128] Y. Saad. Preconditioned Krylov subspace methods for CFD applications. In W. G. Habashi, editor, *Proceedings of the International Workshop on Solution Techniques for Large-Scale CFD Problems*, pages 179–195, Montréal, Québec, 1994.

[129] Y. Saad. ILUM: A multi-elimination ILU preconditioner for general sparse matrices. *SIAM J. Sci. Comput.*, 17:830-847, 1996.

[130] Y. Saad and A. Malevsky. P_SPARSLIB: A portable library of distributed memory sparse iterative solvers. In *Proceedings of Parallel Computing Technologies (PaCT-95), 3rd International Conference*, St. Petersburg, 1995.

[131] Y. Saad and M. Sosonkina. Distributed Schur complement techniques for general sparse linear systems. Technical Report UMSI 97/159, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 1997.

[132] Y. Saad and K. Wu. DQGMRES: A direct quasi-minimal residual algorithm based on incomplete orthogonalization. *Num. Lin. Alg. Appl.*, 3:329-343, 1996.

[133] Y. Saad and J. Zhang. http://www.cs.umn.edu/~jzhang/bilum.html, 1997.

[134] Y. Saad and J. Zhang. BILUM: Block versions of multi-elimination and multi-level ILU preconditioners for general sparse linear systems. Technical Report UMSI 97/126, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 1997.

[135] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Co., Boston, MA, 1996.

[136] B. Smith, W. Gropp, and L. C. McInnes. PETSc 2.0 users' manual. Technical Report ANL-95/11, Argonne National Laboratory, Argonne, IL, 1995.

[137] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 2nd edition, 1991.

[138] K. Takahishi, J. Fagan, and M.-S. Chin. Formation of a sparse bus impedance matrix and its application to short circuit study. In *Proc. 8th PICA Conf.*, pages 63–69, Minneapolis, MN, June 4–6, 1973.

[139] W.-P. Tang. Towards an effective sparse approximate inverse preconditioner. *SIAM J. Sci. Comput.*, submitted.

[140] R. S. Tuminaro, J. N. Shadid, and S. A. Hutchinson. Aztec user's guide. Technical Report SAND95-1559, Sandia National Laboratories, Albuquerque, NM, 1995.

[141] R. R. Underwood. An approximate factorization procedure based on the block Cholesky decomposition and its use with the conjugate gradient method. Technical Report NEDO-11386, General Electric Co., Nuclear Energy Div., San Jose, CA, 1976.

[142] A. Van der Sluis and H. A. Van der Vorst. The rate of convergence of conjugate gradients. *Numer. Math.*, 48:543–560, 1986.

[143] H. A. Van der Vorst. Iterative solution methods for certain sparse linear systems with a non-symmetric matrix arising from PDE-problems. *J. Comput. Phys.*, 44:1–19, 1981.

[144] H. A. Van der Vorst. Stabilized incomplete LU-decompositions as preconditionings for the Tchebycheff iteration. In D. J. Evans, editor, *Preconditioning methods: Analysis and Applications*, New York, 1983. Gordon and Breach.

[145] R. S. Varga. Factorization and normalized iterative methods. In R. E. Langer, editor, *Boundary Problems in Differential Equations*, pages 121–142, Santa Barbara, CA, 1960. University of Wisconsin Press.

[146] R. S. Varga, E. B. Saff, and V. Mehrman. Incomplete factorizations of matrices and connections with H-matrices. *SIAM J. Numer. Anal.*, 17:787–793, 1980.

[147] J. W. Watts III. A conjugate gradient truncated direct method for the iterative solution of the reservoir simulation pressure equation. *Soc. Pet. Eng. J.*, 21:345–353, 1981.

[148] G. Wittum. On the robustness of ILU-smoothing. *SIAM J. Sci. Statist. Comput.*, 10:699–717, 1989.

[149] K. Wu and G. Li. BKAT: An object-oriented block Krylov accelerator toolkit. Presentation at Cray Research, Inc., September 1995. Available from `kewu@kjwu.lbl.gov`.

[150] A. Yu. Yeremin. Private communication, 1995.

[151] D. M. Young and K. C. Jea. Generalized conjugate-gradient acceleration of nonsymmetrizable iterative methods. *Lin. Alg. Appl.*, 34:159–194, 1980.

[152] Z. Zlatev, V. A. Barker, and P. G. Thomsen. SLEST: A Fortran IV subroutine for solving sparse systems of linear equations. User's guide. Technical Report NI-78-01, Numerisk Institut, Lyngby, Denmark, 1978.