Diss. ETH No. 9838

Solution of Large Unsymmetric Systems of Linear Equations

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
Doctor of Technical Sciences

presented by
CLAUDE POMMERELL
Dipl. Informatik-Ing. ETH
born 30 June 1964
citizen of Luxembourg

accepted on the recommendation of
Prof. Dr. W. Fichtner, examiner
Dr. M. Gutknecht, co-examiner

1992

fir d'Mamm an de Papp

Acknowledgements

I would like to express my gratitude to my adviser, Prof. Wolfgang Fichtner, for his confidence in me and my work, for leaving me enough freedom to try my own ideas, for establishing contacts to experts in different fields, and, most of all, for making me the right offer at the right time. I am grateful to Dr. Martin Gutknecht for agreeing to co-referee my thesis and checking its mathematical contents thoroughly.

I am deeply indebted to Henk van der Vorst from Utrecht University. Each of our occasional conversations at a conference, a visit, or an e-mail exchange increased my understanding of iterative solvers by a quantum leap. I would like to thank Bill Coughran from AT&T Bell Laboratories for fruitful discussions and continuing interest in my work.

All the people who formed the Integrated Systems Laboratory of ETH Zurich over the last four years contributed to the success of this thesis by their friendship, often over a beer or more, and by their collaboration on technical problems, especially in the K2 team and the device simulation group. Marco Annaratone introduced me to parallel computers. Gernot Heiser and Stephan Müller used my software since its very first buggy releases and bore a significant part of the debugging and experimentation efforts. Other interesting examples and bug reports came from Josef Bürgler, Paolo Conti, Hartmut Dettmer, Kevin Kells, and Ulrich Krumbein, as well as from Matt Noell from Motorola Inc. Chris Hegarty, Marco Fillo, and Roland Rühl improved the quality of the text by their constructive proofreading. Carlo Bach, Michael Halbherr, Markus Schmidt, and other dear friends among those mentioned above helped me keeping my morale and motivation.

I am grateful to my family, relatives, and friends outside of the Lab who keep standing by me despite the fact that I often neglected our relationship by giving too much priority to my work.

My work was supported initially by a grant from the Swiss National Science Foundation, and over the last three years by the Cray Research Inc. University Research & Development Grant Program. Randy Bramley and Ahmed Sameh from the Center for Supercomputing Research and Development at the University of Illinois, Urbana-Champaign, as well as Bill Coughran and Norm Schryer from AT&T Bell Laboratories at Murray Hill invited me to longer visits at their institutions in spring and summer 1990, and supported me during these stays.

Contents

Abstract

The numerical solution of large systems of linear equations lies at the heart of many scientific computing efforts. Sparse systems with several hundreds of thousands of unknowns have to be solved today. Ill-conditioned systems with unsymmetric matrices whose irregular sparsity structures reflect irregularly refined discretization grids for partial differential equations are particularly difficult to solve. Dozens or hundreds of such systems occur in a single semiconductor device simulation, and the linear solver is the main time-consuming operation in this application.

Direct sparse solvers, based on classical triangular factorizations, cannot be used for very large matrices. Their storage requirements increase superlinearly with the problem size, and the growth factor increases with the dimensionality of the discretization grid.

Iterative linear solvers are a viable alternative, even if none of the current approaches suits all the requirements alone. All successful iterative methods approximate the solution by stationary points in sequences of Krylov subspaces. GMRES uses the minimum number of matrix-vector products to minimize the residual in a given Krylov subspace, but its storage requirements increase linearly with the iteration number. Truncated or restarted variants are much less effective. BiCG and variants thereof do not have such a minimization property, but are often more successful, and have constant and low storage requirements. In our experience, Bi-CGSTAB is currently the fastest and most robust method in this class. BiCG and its variants have a bad reputation because of breakdown and cancellation problems, but the rare real occurrences of these effects can be fixed through restarting.

The convergence of iterative method is strongly improved, and often even enabled by preconditioning. Approximate factorizations constitute a family of effective preconditioners. Incomplete factorizations without fill are fast preconditioners capable of solving most linear systems. Particularly ill-conditioned systems can only be solved with more powerful preconditioners, such as a new efficient approximate factorization based on numerical dropping. Nested iterative solvers are another new alternative.

Because of the huge resource requirements, iterative solvers have to run on the most powerful supercomputers available. The efficient vectorization, parallelization, and balanced distribution of operations like regular and transposed matrix-vector multiplication with irregular sparsity structures, the solution of sparse triangular systems in incomplete factorization preconditioners, and the set-up of approximate factorizations, require sophisticated data structures. The techniques to exploit the different high-performance features of vector and parallel computers with shared or distributed memory are based on graph theory concepts, such as mapping, matching, and coloring. Flexibility, portability, and automatic adaptation can be achieved without efficiency penalty in an object-oriented software design for an iterative solver package.

The actual performance of iterative solvers on supercomputers depends mostly on aspects of the memory system, like indirect addressing, memory bandwidth, interconnection network latency, and storage capacity, and varies with the characteristics of the linear system. The fastest method on one platform may not be the most efficient on another computer.

Zusammenfassung

Die numerische Lösung großer linearer Gleichungssysteme gehört zu den Kernproblemen vieler Projekte des wissenschaftlichen Rechnens. Heutzutage müssen schwachbesetzte Systeme mit mehreren hunderttausend Unbekannten gelöst werden. Besonders schwierig gestaltet sich das Lösen von schlecht konditionierten Systemen mit unsymmetrischen Matrizen, deren unregelmäßige Struktur die ungleichmäßig verfeinerten Diskretisierungsgitter für partielle Differentialgleichungen widerspiegelt. Dutzende oder Hunderte solcher Gleichungssysteme treten in einer einzigen Simulation von Halbleiterstrukturen auf, und die Lösung linearer Systeme ist die zeitaufwendigste Tätigkeit in dieser Anwendung.

Direkte Lösungsverfahren für schwachbesetzte Gleichungssysteme, aufbauend auf der klassischen Zerlegung in Dreiecksmatrizen, können für sehr große Matrizen nicht eingesetzt werden. Ihr Speicherbedarf wächst stärker als linear mit der Problemgröße, und der Wachstumsfaktor nimmt mit der Dimensionalität des Diskretisierungsgitters zu.

Iterative Lösungsverfahren stellen eine gangbare Alternative dar, auch wenn keiner der gegenwärtigen Ansätze allein alle Bedürfnisse erfüllt. Alle erfolgreichen iterativen Methoden nähern sich der Lösung über stationäre Punkte in Folgen von Krylov-Unterräumen. GMRES braucht die minimale Anzahl Matrixvektorprodukte, um das Residuum in einem bestimmten Krylov-Raum zu minimieren, der Speicherbedarf steigt aber linear mit der Anzahl Iterationen an. Bei Abschneiden oder Neustarts verlieren GMRES-artige Methoden schnell an Wirkung. Varianten von BiCG haben keine solche Minimierungseigenschaften, sind aber oft erfolgreicher und haben konstanten, niedrigen Speicherbedarf. Innerhalb dieser Klasse ist nach unserer Erfahrung Bi-CGSTAB das derzeit schnellste und robusteste Verfahren. BiCG und Varianten haben einen schlechten Ruf aufgrund der Möglichkeit frühzeitigen

Abbrechens oder Auslöschungsfehlern, aber in der Praxis kann das seltene Vorkommen solcher Erscheinungen durch Neustarts behoben werden.

Die Konvergenz von iterativen Verfahren wird durch Vorkonditionierung stark verbessert, und manchmal sogar erst ermöglicht. Unvollständige Zerlegungen bilden eine Familie von wirksamen Vorkonditionierern. Schnelle Vorkonditionierer erhält man durch unvollständige Zerlegungen ohne Füllen ursprünglicher Nulleinträge. Außergewöhnlich schlecht konditionierte Gleichungssysteme können nur mit leistungsfähigeren Vorkonditionierern gelöst werden, zum Beispiel durch eine neue angenäherte Zerlegung mit numerisch begründeter Vernachlässigung von Matrixeinträgen. Verschachtelte iterative Lösungsverfahren bieten eine weitere neue Alternative.

Aufgrund des hohen Bedarfs an Rechen- und Speicheraufwand müssen iterative Löser auf den leistungsfähigsten zur Verfügung stehenden Superrechnern laufen können. Die effiziente Vektorisierung, Parallelisierung und ausgeglichene Verteilung von Operationen wie die reguläre und die transponierte Matrixvektormultiplikation mit unregelmäßigen Strukturen, die Lösung von schwachbesetzten Dreieckssystemen in auf unvollständiger Zerlegung aufbauenden Vorkonditionierern und der Aufbau von angenäherten Zerlegungen verlangt den Einsatz komplizierter Datenstrukturen. Die Algorithmen, welche die Ausnutzung der Hochleistungskomponenten von Vektor- oder Parallelrechner mit gemeinsamen oder verteiltem Speicher ermöglichen, beruhen auf Konzepten der Graphentheorie, wie Abbildung, Zuordnung und Färbung. Flexibilität, Portierbarkeit und automatische Anpassung können ohne Effizienzeinbußen mit Hilfe einer objektorientierten Programmentwicklung in einem Paket von iterativen Lösern erreicht werden.

Die tatsächliche Leistung iterativer Lösungsverfahren auf Superrechnern steht hauptsächlich im Zusammenhang mit den Eigenschaften des Speichers, wie indirekte Adressierung, Bandbreite, Netzwerklatenz und Kapazität, und ändert sich mit der Struktur der linearen Gleichungssysteme. Das schnellste Verfahren auf dem einen Rechner kann unter Umständen auf einer anderen Maschine nicht mehr das effizienteste sein.

1

Introduction

With the rapid advances of computer technology in the last forty years, computer-based mathematical modeling has become the most important method of experimentation in many different fields. Scientific computing allows scientists, engineers, and analysts to predict the behavior of complex systems that would be much more expensive to build in reality, and too difficult or even impossible to experiment with. Continued growth in available computational resources leads to more and more complicated models and larger problems.

The efficient solution of large (and usually sparse) systems of linear equations lies at the heart of many scientific computing efforts, and constitutes often the main resource-intensive component of these applications. An important class of such problems consists in the numerical solution of systems of partial differential equations (PDEs), as they occur in computational fluid dynamics, weather forecasting, oil exploration, structural mechanics, chemistry, or microelectronics.

Complicated state-of-the-art applications require the solution of linear systems with up to several hundreds of thousands unknowns, very irregular sparsity structures, and lacking nice numerical properties. Sparse variants of the classical elimination methods to solve linear systems cannot be used because of their enormous storage requirements. Solvers of linear systems have to run efficiently on high-performance computers, and exploit their architectural features, like vectorization and parallelism.

This thesis concentrates on iterative solvers for large, sparse, unsymmetric linear systems. The techniques are evaluated with a clear focus on their application in semiconductor device simulation and on their practical implementation on current supercomputers.

Section 1.1 below introduces the basic notions in iterative solvers, and indicates where these notions are detailed in this thesis. Section 1.2 presents the notation.

1.1 Overview and basics

A linear solver is a numerical method to solve a linear system

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

In this thesis, the matrix A is always presumed real, square, and non-singular, so that, for each real right-hand side vector b, there is a unique real solution x given by

$$x = A^{-1}b =: x^\star \; . \tag{1.2}$$

Chapter 2 describes where and how such linear systems arise, and what demands the environment makes on linear solvers.

A **direct solver** is a linear solver that constructs an explicit operator $v \mapsto A^{-1}v$ by computing the inverse A^{-1} itself or a factorization of it, and then applies this operator to b to find the solution x as in the explicit form (1.2). Direct solvers are briefly discussed in Section 2.7.

An **iterative solver** is a linear solver that produces a sequence $x_0, x_1, \ldots,$ $x_k \ldots$ of approximations to the solution. The step going from x_{k-1} to x_k is called the k-th iteration of the solver. x_0 is the initial guess. Unless the surrounding client application can provide a good initial guess, x_0 is usually set to the zero vector.

Many iterative methods are designed such that, in exact arithmetic, the solution is found after a finite number of iterations. This property, however, is rarely used. An iterative solver usually terminates if the approximation is "close" enough to the solution.

Several iterative methods to solve linear systems are presented in Chapter 3, including their derivation, their formulation, and their typical convergence behavior.

The convergence speed of a given iterative method depends heavily on the characteristics of the matrix A. A **preconditioned iterative method** is an iterative solver that solves another linear system

$$\widetilde{A}\widetilde{x} = \widetilde{b} \tag{1.3}$$

which is equivalent to the original system (1.1), but in which the characteristics of the preconditioned matrix $\widetilde{A}$ are more suitable for the convergence of the iterative method. Preconditioning is discussed in Chapter 4.

The error of the k-th approximation is defined as

$$\epsilon_k = x^\star - x_k \; . \tag{1.4}$$

An approximation x_k is said to be close to the solution if some given norm of the error vector $||\epsilon_k||$ is "small". The iterative solver is then said to have converged. The sequence of error norms of the approximations characterizes the convergence behavior of an iterative solver on a linear system. The residual of the k-th approximation is defined as

$$r_k = b - Ax_k \; . \tag{1.5}$$

A norm of the residual $||r_k||$ is a measure how well the approximation x_k solves the system (1.1). Since $r_k = Ae_k$, any norm on the residual vector is equivalent to another norm on the error vector. A popular measure of the closeness to the solution is the relative Euclidean residual norm, defined as the quotient of the ℓ_2-norm of the residual over the ℓ_2-norm of the right-hand side, $||r_k||_2/||b||_2$.

Ways to control convergence and termination of iterative solvers, as well as other convergence related issues, like optimality, breakdown, and effects resulting from finite-precision arithmetic, are discussed in Chapter 5.

The size and the number of linear systems to be solved in large scale applications such as device simulation exceed the computing power of most conventional computers. In order to keep acceptable turn-around, these applications have to be executed on high-performance computers. Chapter 6 discusses which transformations of the methods lead to a high rate of exploitation of the architectural features of both existing vector supercomputers and projected distributed-memory computers. Very large linear systems are usually sparse, that is, the large majority of the matrix entries are zero. Sparse matrix data structures therefore play a major role in the work discussed in Chapter 6. The last sections of Chapter 6 cover other implementation issues such as flexibility and portability, and culminate in an overview of PILS, a software package of iterative linear solvers based on the work in this thesis. PILS benchmarks on state-of-the-art machines, from workstations to supercomputers, conclude Chapter 6.

The final Chapter 7 draws some conclusions and raises a few of the numerous open questions about iterative solvers that still need to be addressed.

1.2 Symbols used in this document

The mathematical notation in this thesis is based on [GvL83, HJ85, Avr76, Roc70, Har72]. I have tried to use unique symbols throughout this document, at least for the most important quantities. Exceptions are only made to enhance mnemonics. The typographical rules are as follows:

1. Small roman letters between i and n designate integers.
2. Other small roman letters designate vectors.
3. Small greek letters designate scalars.
4. Capital roman letters designate matrices.
5. Capital greek letters designate polynomials.

The symbols A, x, and b are reserved for the linear system to be solved. The **current iteration** of an iterative method is usually the k-th iteration. Quantities indexed by k (such as r_k), or an offset relative to k (such as r_{k-1}) refer to the sequence item corresponding to this iteration of the method. x_k, r_k, and ϵ_k always denote the current approximation to the solution, the current residual (as defined by (1.5)), and the current error of the approximation (as defined by (1.4)), respectively. x_0 is the initial guess, and r_0 the initial residual. $x^\star$ is the exact solution $A^{-1}b$.

p_k is usually the search direction of the iteration, α_k the line search parameter, and β_k or β_{ki} are orthogonalization parameters for the search directions. In Biconjugate Gradients and related methods, vectors with a hat, like $\hat{r}_0$, refer to the dual system $A^T \hat{x} = \hat{b}$. Some methods have a storage parameter ℓ, which means that at most ℓ previous search directions are kept in memory.

Quantities of the preconditioned method, solving the preconditioned system $\widetilde{A}\widetilde{x} = \widetilde{b}$, are capped with a tilde. The preconditioning matrix is Q. Matrices named L, U, or D are lower triangular, upper triangular, or diagonal, respectively.

Unless otherwise stated, vectors have length n, and matrices are square and have size $n \times n$. Single entries of a matrix are denoted by the doubly indexed lower-case roman letter version of the matrix name. The entry at the intersection of the i-th row and the j-th column in the matrix A is named a_{ij}. Numbering starts with zero, so that $a_{0,n-1}$ is the upper right entry of the matrix A.

The number of nonzero entries in the sparse matrix A is m. The maximum number of nonzeros in a row of the matrix is d_{max}, the average number of nonzeros per row is $d_{\text{aver}} = m/n$.

Entire rows and columns viewed as vectors are written by substituting a star for the running index. $l_{i\star}$ is the i-th row of the matrix L, and $u_{\star j}$ is the j-th column of U. The index range from j to k (both included) is given by $[j : k]$. The entries of the vector v whose indices are in the range from j to k form a shorter vector named $v_{j:k}$. To improve readability, indices are sometimes not written as subscripts, like in $A_{0:j,0:k_j}$, but by enclosing them into brackets and appending them, like in $A[0 : j, 0 : k[j]]$.

The identity matrix is I, its i-th column vector is e_i. The size of these quantities is not given explicitly; it is whatever fits into the expression.

Scalar-valued scalar functions ($\mathbb{R} \rightarrow \mathbb{R}$) are denoted by a lower-case f, scalar-valued vector functions ($\mathbb{R}^n \rightarrow \mathbb{R}$) by a lower-case greek phi φ, and vector-valued vector functions ($\mathbb{R}^n \rightarrow \mathbb{R}^n$) by an upper-case calligraphic $\mathcal{F}$.

Empty sums and empty products are defined and have as value the neutral element of the operation:

$$\sum_{i=0}^{-1} v_i = 0 \,, \qquad \prod_{i=0}^{-1} \gamma_i = 1 \,.$$

Algorithms are sketched in a pseudo-code resembling PASCAL or ADA. A **for**-loop prescribes that its iterations are executed by increasing index values, whereas the iterations of a **foreach**-loop can be done in any order. The latter does not imply yet that the loop is parallel, but the former imposes sequential execution. The rest of this notation should be intuitively clear.

Unless otherwise stated, all plots in this thesis that depict the convergence behavior of an iterative solver display the relative Euclidean residual norm as a function of the iteration number.

The legend for data structure and memory access pattern drawings is presented in Section 6.1.

Counts of individual floating-point operations (additions or multiplications) are given in **flops** (lower-case). The unit of 64-bit precision floating-point operations per microsecond is **MFlops** (capitalized). One MFlops is achieved when one million flops are executed in one second.

2

Requirements for solvers of large linear systems

2.1 Scope of this thesis

The title of this thesis could just as well be

> *"Preconditioned conjugate gradients-like iterative methods for the solution of large sparse unsymmetric systems of linear equations in 3-D semiconductor device simulation, their convergence behavior, and their implementation on vector- and parallel supercomputers."*

This endless title describes accurately the contents of this work. The following list details why certain words in this title are considered redundant:

"conjugate gradients-like". Parameter independent Krylov subspace methods, which can all be viewed as generalizations of the Conjugate Gradient method [HS52], are the only successful iterative methods for ill-conditioned and unsymmetric linear systems. All other known iterative methods put too stringent conditions on the matrices, like certain numerical properties (e.g., SSOR), knowledge about the eigenspectrum (Chebychev methods), or properties of the sparsity structure (ADI).

"preconditioned". Standard preconditioning always increases the convergence rate of Krylov subspace methods in a significant way on practical problems, and unpreconditioned methods often do not succeed on ill-conditioned systems.

"iterative". Superlinearly growing storage requirements and even larger increases in CPU requirements make direct methods infeasible as alternative solvers for very large linear systems, especially for 3-D problems. Section 2.7 brings up direct methods for large sparse systems and compares them to iterative methods.

"methods for the solution". In this thesis, the term **iterative solver** is used only for the combination of an iterative method, a choice of a preconditioner and its position, and a termination control mechanism.

"sparse". The number of nonzero entries per row of the matrix is usually constant for applications simulating systems of PDEs. The largest sparse problem size that fits into the available memory therefore increases linearly with the increase of the storage capacities, while the fitting dense problem size increases only with the square root of the capacities. Therefore, a *large* matrix problem must be sparse today. Section 2.7 details the storage requirements for current problem sizes. Note however that even for the largest dense problems iterative methods seem to be more popular than direct dense solvers [Ede91].

"unsymmetric". State-of-the-art semiconductor device simulation is not limited to the solution of the Poisson equation. The carrier continuity equations engender unsymmetric matrices for the linear systems. Still, this word stays in the title, as the solution of symmetric positive-definite systems is much simpler, and has been studied more often.

"3-D". The accuracy of simulations cannot be increased only through larger grid sizes for 2-D models. *Large* means today that higher dimensional effects and refined numerical models (in device simulation, e.g., solving the temperature equation along with the conventional drift-diffusion equations) have to be considered as well [KMFW91].

"semiconductor device simulation". Although all the examples in this thesis are taken from device simulation, the unsymmetric linear systems appearing in other large scale sparse applications, like oil-reservoir modeling or fluid dynamics [Kui87] are very similar, as they arise also from finite difference, finite element, or finite volume discretizations

of systems of partial differential equations. A short introduction into semiconductor device simulation is given in Section 2.6. The relation between linear and nonlinear systems of equations is treated in Section 2.4.

"their convergence behavior". Unfortunately, the successful iterative solvers in device simulation are not those for which nice convergence theorems exist. Experimental study of the convergence behavior is therefore mandatory to assess the quality of the methods.

"their implementation". Almost all of the material presented in this thesis has been implemented in PILS, a package of iterative linear solvers [PF91b]. PILS is fully integrated in several device simulation codes [Bür90, Hei91, KMFW91] and other applications, and has been used in many different problems over a period of one and a half years. Only experience with such an embedded implementation allows one to draw generalizable conclusions on those aspects of iterative solvers that cannot be treated fully analytically.

"on vector- and parallel supercomputers". Large device simulations performed today take several hours on top-of-the-line supercomputers like the Cray Y/MP or the NEC SX/3, or up to a week on minisupercomputers like the Convex C-220. They would take months on average scalar architectures or workstations, so the use of high-performance computers is mandatory. Since the linear solver takes over 90% of the time in such simulations [HPWF91], it is obvious that this part of the code has to exploit architectural features like vector pipelining and parallel execution as much as possible. Although state-of-the-art device simulators are not yet ready to run on massively parallel computers, it is clear that such machines will have to be used in the future. Implementation aspects for distributed-memory multicomputers are thus given the same importance as current vector machines.

2.2 Sparse matrices

Very large matrices used in real-world applications are usually sparse, that is, most of their entries are zero. To save storage and avoid redundant operations, they are not stored in two-dimensional arrays of size $n \times n$, like dense matrices, but in special data structures. Sparse matrix data structures are discussed in Chapter 6. See also [GL81, Pis84, DDSvdV91, BCF$^+$85, Saa89, Saa90].

Data structures for general sparse matrices require the storage of at least one additional integer along with each nonzero entry. Special structural properties of a matrix, like symmetry, structural symmetry, full diagonal, co-diagonals, or small bandwidth, are usually exploited to further reduce storage requirements. We focus on matrices with irregular sparsity structures that have to be treated as general sparse matrices.

Many matrix-to-matrix operations like factorization, inversion, and multiplication do not preserve sparsity. Many other operations with sparse matrices have complexity $O(n^2)$ or $O(m^2/n)$. For many data structures, the only truly efficient operation (having complexity $O(m)$ and sufficient parallelism) is matrix-vector multiplication. Algorithms that use the matrix only as an operator to compute $w := Av$ for a given vector v are therefore often preferred. Most of the (unpreconditioned) Krylov subspace methods have this advantage. Some others require the transposed operation $w := A^T v$ as well, which is also supported efficiently by a few sparse matrix formats [BCD+89, PF91b].

2.3 Sparse graphs

There are several ways to view the sparsity structure of a matrix as a graph. Some properties and algorithms that seem obscure and hard to explain become much easier to comprehend in such an associated graph.

1. The matrix A is viewed as a directed graph (or **digraph**) with n vertices numbered as the rows and columns, and m arcs pointing from the i-th to the j-th vertex for each nonzero entry a_{ij}:

$$\begin{aligned} G &= (V, E) \text{ with} \\ V &= \{v_i\}_{i=0}^{n-1} \text{ and} \\ E &= \{(v_i, v_j) | a_{ij} \neq 0\} \ . \end{aligned}$$

2. A symmetric or structurally symmetric matrix can also be viewed as an **undirected graph** with n vertices numbered as the rows and columns, and $m/2$ edges between each pair of vertices v_i and v_j such that a_{ij} (and thus a_{ji}) is nonzero:

$$\begin{aligned} G &= (V, E) \text{ with} \\ V &= \{v_i\}_{i=0}^{n-1} \text{ and} \\ E &= \{\{v_i, v_j\} | a_{ij} \neq 0 \text{ or } a_{ji} \neq 0\} \ . \end{aligned}$$

3. Alternatively, the matrix may be viewed as a **bipartite graph** with $2n$ vertices partitioned into a set of n row vertices numbered as the rows of A and a set of n column vertices numbered as the columns of A. Each off-diagonal nonzero a_{ij} leads to an arc from the i-th row vertex to the j-th column vertex:

$$\begin{aligned} G &= (V_r \cup V_c, E) \text{ with} \\ V_r &= \{v_i\}_{i=0}^{n-1} , \\ V_c &= \{w_i\}_{i=0}^{n-1} \text{ and} \\ E &= \{(v_i, w_j) | a_{ij} \neq 0\} . \end{aligned}$$

For many purposes it is convenient to ignore the loop edge caused by nonzero entries on the diagonal of the matrix. As no more than one entry is defined at a given position in a matrix, there are no multiple edges or arcs between a pair of vertices.

Given a graph that describes the sparsity structure of a matrix, every arc can be weighted by the value of the corresponding entry of the matrix. This leads to an isomorphism between matrices and weighted graphs.

This thesis uses all three representations defined above, and refers to them as "the digraph", "the undirected graph", and "the bipartite graph" associated with the matrix.

2.4 Solution of nonlinear systems of equations

The solution of the equation

$$\mathcal{F}(u) = 0 \tag{2.1}$$

for a smooth nonlinear vector function $\mathcal{F} : \mathbb{R}^n \rightarrow \mathbb{R}^n$ meeting certain conditions (among others, existence and uniqueness of such a solution) can be found iteratively by a **Newton method** [Avr76, GMW81]. If the Jacobian matrix $\nabla\mathcal{F}$ is positive-definite in a sufficiently large region around the solution, the Newton iteration

$$u_{i+1} = u_i - \left(\nabla\mathcal{F}(u_i)\right)^{-1} \mathcal{F}(u_i) \tag{2.2}$$

converges monotonically to the solution of Equation (2.1). If the Jacobian is indefinite, the iteration may overshoot, so that $||\mathcal{F}(u_{i+1})|| > ||\mathcal{F}(u_i)||$. In this case a damped Newton iteration

$$u_{i+1} = u_i - \theta_i \left(\nabla\mathcal{F}(u_i)\right)^{-1} \mathcal{F}(u_i) \tag{2.3}$$

is applied, where the scalar θ_i enforces monotonicity.

Each iteration of such a nonlinear solver thus requires the solution of one linear system with the matrix $A = \nabla\mathcal{F}(u_i)$ and the right-hand side $b = \mathcal{F}(u_i)$.

An iterative linear solver will not provide the exact solution $x^\star = \left(\nabla\mathcal{F}(u_i)\right)^{-1} \mathcal{F}(u_i)$, but an approximation x_k to this solution[1]. The **approximate damped Newton iteration** is then

$$u_{i+1} = u_i - \theta_i \left(\nabla\mathcal{F}(u_i)\right)^{-1} \mathcal{F}(u_i) + \theta_i\epsilon_k \ , \tag{2.4}$$

where ϵ_k is the error of the approximation. Alternatively, the approximation can be viewed as the exact solution of the linear system with the (hopefully nonsingular) perturbed Jacobian $(A + \delta A)$, leading to an approximate damped Newton iteration

$$u_{i+1} = u_i - \theta_i \left(\nabla\mathcal{F}(u_i) + \delta A\right)^{-1} \mathcal{F}(u_i) \ . \tag{2.5}$$

To save time by reducing the number of inner linear iterations, the convergence tolerance of the linear solver should be as weak as possible without endangering the convergence of the outer nonlinear solver. Depending on which of the above views was adopted, different authors suggest different ways of matching the accuracy of the outer and inner iterations. The scheme by Bank and Rose [BR81] is the most popular in device simulation [Pin90, Bür90, Hei91]. It is based on the update formula (2.5) and defines an inner tolerance for the relative Euclidean residual norm $||r_k||_2/||b||_2$. The GIANT scheme by Deuflhard [Deu90] uses the update formula (2.4) to derive an inner tolerance for the relative Euclidean approximation error norm $||\epsilon_k||_2/||x^\star||_2$.

2.5 Discretization of PDEs

Partial differential equations (or systems thereof) are solved numerically by discretizing the domain and solving the discretized version of these PDEs.

---

[1] Note that the index i is here the iteration number of the (outer) nonlinear solver, and k is the iteration number at the termination of the (inner) linear solver.

Such a discretization partitions the (1-D, 2-D, or 3-D) domain into elements called segments, polygons, or polyhedrons, respectively. The ends, corners, or tips of the elements are called grid points. In the multi-dimensional case, the points are connected by edges separating two (2-D) or more (3-D) elements; usually, points are only allowed to reside at the ends, and not inside an edge.

The actual discretization of the PDEs on this grid is done using finite differences [Smi78], finite elements [SF73], or finite volumes [Var62] (also called the box method). The resulting discretized PDEs, together with the discretized boundary conditions, form a system of discrete equations with one unknown per grid point and PDE. Depending on the form of the PDEs, this system is linear and requires a linear solver, or it is nonlinear, but its iterative solution requires the solution of a linear system at each iteration (see Section 2.4).

For single PDE discretizations, every unknown in this linear system corresponds to one grid point. The discretization operator is local, that is, the unknown functions at each grid point depend only on function values at surrounding grid points. Depending on the discretization method, this set of neighbors of a grid point consists of those points that share an edge, face, or element with this point[2]. The matrix of the linear system is sparse, each row of the matrix has nonzero entries only at those column positions that correspond to neighbors of the row vertex. The average number of nonzero entries per row depends on the dimensionality of the grid, the grid generator, and the discretization method, but it is roughly independent of the grid size. If neighbors are defined as adjacent vertices (sharing a common edge), the undirected graph of the sparse matrix coincides with the discretization grid.

The discretization methods impose various additional constraints on the structure of the grid. Finite differences are the most restrictive: the grid has to be regular, that is, each interior point has to have exactly two neighbors in each dimension. Such a grid is also called a tensor-product grid, or a five-point stencil in 2-D and a seven-point stencil in 3-D. Using a natural ordering of the unknowns, the nonzeros of the matrix are only located on the main diagonal and 2 (2-D) or 3 (3-D) co-diagonals on each side of it. tensor-product graphs have also the property that no three distinct vertices v_i, v_j, v_k may be connected in a triangle, or, in other words, at least one of the three entries a_{ij}, a_{jk}, a_{ki} must be zero.

The finite element method can be applied to arbitrary grids, although

---

[2] These three different definitions are equivalent for grids consisting only of simplices.

stability problems can occur with "bad" grids. The box method requires that the Voronoi diagram of the grid points can be formed by using only the mid-perpendiculars of the grid edges [Con91].

The desired accuracy of the solution imposes density constraints on the grid. As the minimal density varies over the domain (by several orders of magnitude in the case of device simulation), the grid density may be finer in some parts of the domain and coarser in others. As the time for the numerical solution of PDEs increases at least linearly with the number of grid points (the grid size), it is imperative to exploit this possibility to vary the grid density. The local refinement potential of finite difference grids is poor and leads to very rapid grid size growth, so state-of-the-art device simulation requires powerful grid generators [CHF91, MKF91] and uses highly irregular grids.

Because any 2-D discretization grid is a partitioning, it is a planar graph. It follows therefore from Euler's Polyhedron Formula [Har72] that the average vertex degree in such a grid is less than six. The 2-D grid generators used at our Laboratory generate an average degree around 5.6, the 3-D grid generators around 6.7.[3]

2.6 Semiconductor device simulation

Numerical semiconductor device simulation predicts the behavior of semiconductor structures (diodes, transistors, memory cells, thyristors, sensors) by solving a set of three transient PDEs, the so-called drift-diffusion equations [vR50]. These consist of Poisson's equation and two continuity equations for the two carriers in semiconductors, electrons and holes. The drift-diffusion equations relate the values of the electrostatic potential and the two carrier concentrations.

The discretization of these PDEs leads to a system of nonlinear equations. This coupled system, involving three unknowns per grid point, can be solved by a damped Newton solver. This approach is called the fully coupled Newton solution. Alternatively, the blocks for each of the PDEs can be

---

[3] These numbers are based on empirical analysis of all the 224 distinct nontrivial (more than 1000 vertices) grids that where on disk at our Laboratory on October 9, 1991. The average vertex degree of the 172 2-D grids lay between 3.95 and 5.98, with an average of 5.60 and a standard deviation of 0.51. The average vertex degree of the 52 3-D grids lay between 5.37 and 7.79, with an average of 6.72 and a mean deviation of 0.65. The largest 2-D grid had 22,081 points, the largest 3-D grid had 96,589 points.

solved separately in turn in a block Gauss-Seidel iteration, applying a damped Newton method to each of the blocks. This latter approach is known as plug-in or Gummel iteration. The two techniques are often combined, for example by using a few Gummel iterations to obtain a good initial guess for the coupled Newton solution [FRB83, BRF83, Sel84, BCF+85, Bür90, Pin90, Hei91].

The linear systems arising in the Gummel scheme comprise a single unknown for each grid point[4], and the sparsity structure of the matrices corresponds to the grid structure. The number of off-diagonal nonzeros per row in the coupled matrices is less than simply three times the degree of the grid points, as some dependencies between the unknowns from different PDEs on neighboring grid points vanish. The average number of nonzeros per row in the coupled matrices is typically around 15 in 2-D and around 17 in 3-D device simulation problems[5]. The main diagonal entries of all these matrices are nonzero, but the matrices are not diagonally dominant. The Poisson matrices are symmetric positive definite, the continuity matrices are structurally symmetric.

Several tens of linear systems typically have to be solved in stationary simulations, several hundreds or even thousands in quasi-stationary or transient simulations. Only three distinct sparsity structures occur, one each for single Poisson matrices, single continuity matrices, and coupled systems.

2.7 Direct and iterative solvers

The alternative to iterative solvers are sparse direct solvers. Direct solvers are often preferred over their iterative counterparts because of their reliability and predictability: they are numerically stable (at least if numerical pivoting is used) [DDSvdV91], and they take constant time and storage for a given sparsity structure (at least if numerical pivoting is not used). Solving for more than one right-hand side costs very little.

Most direct solvers are based on variants of Gaussian elimination. They

---

[4] Or a few less, since Dirichlet boundary points can be ignored in all three equations, and the equations simplify to a single Laplace equation inside insulators.

[5] Using the same grid files as for footnote 3 on page 14 and the corresponding doping concentrations inside the device simulator SIMUL [KMFW91], the average number of off-diagonal nonzeros per row of the coupled matrix ranged for 2-D simulations between 9.19 and 15.70 with an average of 14.73 and a standard deviation of 1.28, and for 3-D simulations between 14.49 and 19.80 with an average of 17.20 and a standard deviation of 1.44.

construct a lower triangular matrix L and an upper triangular matrix U such that $LU = A$. The triangular factors L and U are also sparse, but usually much denser than the original matrix. Nonzeros in the factors appearing at zero positions of the original matrix are called **fill**. Reordering of the equations can reduce the amount of fill. The problem of computing the ordering that minimizes the fill is NP-complete. Heuristics like reverse Cuthill-McKee, nested dissection, or minimum degree and variants are used instead [GL81, GL89]. Numerical stability, storage, efficiency, and parallelism are usually traded for one another [DER86].

The amount of fill increases superlinearly with the problem size and with the dimensionality of the problem. The fill for a matrix arising from a 3-D discretization is higher than for a matrix from a 2-D discretization with the same number of grid points. It is the combination of these two growth factors that makes the storage requirements of sparse direct solvers an issue when passing from 2-D to 3-D models.

Figure 2.1 compares the memory requirements for direct and iterative linear solvers. The fill was determined without actually performing the factorization on all the examples (no available computer has enough memory for the large 3-D cases), but an algorithm with minimal storage [BS87] was assumed, using nested dissection ordering [GL81] to reduce the fill. The analysis involved 224 nontrivial examples from device simulation, using the matrices from the coupled Newton solution, and assuming 8-byte floating-point and 4-byte integer numbers[6].

In the double logarithmic plot of Figure 2.1, the sample points for each of the assorted categories lie close to a line. We can thus state an empirical law of fill given by

$$s = \mu n^{\nu} , \tag{2.6}$$

where s is the amount of storage for a given number of unknowns n. Table 2.1 gives the values obtained from our empirical study.

The largest 2-D discretizations for semiconductor devices exceed rarely 10k grid points, and thus 30k unknowns for coupled linear systems. Direct solvers for such 2-D problems fit into 100 MBytes of memory, which are usually available on departmental computers or in medium sized batch queues of supercomputers. On the other hand, iterative solvers for such 2-D problems

---

[6] Footnote 3 on page 14 and footnote 5 on page 15 give more informations about this empirical study.

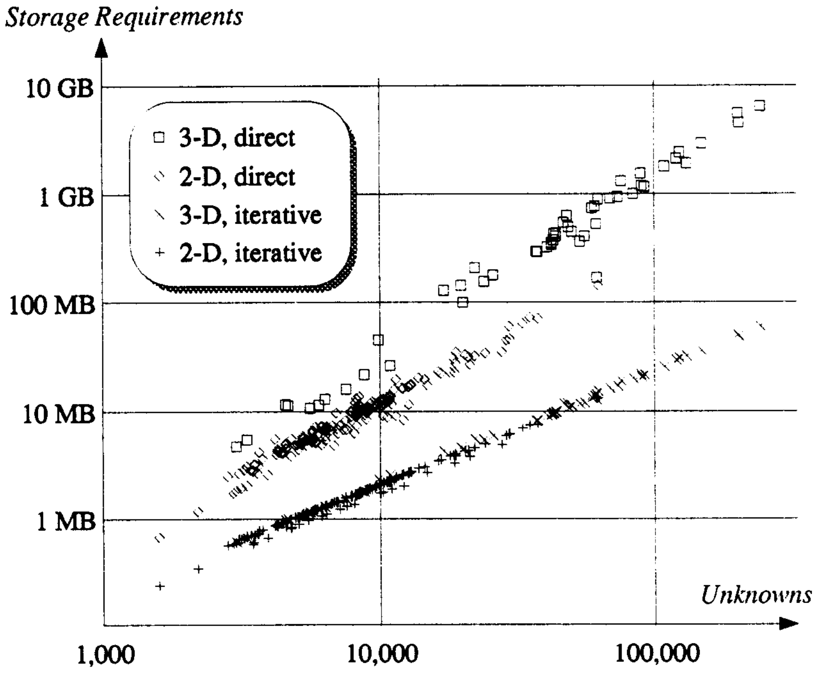


Figure 2.1: *Sampled storage requirements for direct and iterative solvers.*

grid dimension	*type of solver*	*factor μ [bytes/unknown]*	*exponent ν*
2-D	iterative	167.5	1.021
2-D	direct	62.8	1.319
3-D	iterative	155.7	1.036
3-D	direct	8.9	1.645

Table 2.1: *Empirical values for the storage requirement formula (2.6).*

fit into some 8 MBytes available on any workstation. Direct solvers for average size 3-D simulations can run only on very large computer memories, and no machine available today could accommodate truly large 3-D problems with several hundreds of thousands or even millions of grid points. Iterative solver storage requirements even for 3-D problems still increases linearly with problem size. At least for such large 3-D problems, their is no choice: direct methods are unfeasible because of their storage requirements, and iterative solvers have to be used.

The number of operations to perform the factorization grows even faster than the storage requirements. Timing comparisons between the two choices show that for grid sizes above a certain crossover point, iterative methods are more efficient [HPWF91]. Such a comparison will always be biased, because it involves many parameters from the problem, the accuracy requirements, the implementation, and the platform, but the critical grid size is certainly much lower for 3-D than for 2-D.

2.8 Matrix conditioning

The number of iterations that a given iterative method needs to solve a system of linear equations $Ax = b$ to a given accuracy depends heavily on the conditioning of the linear system.

The conditioning of a problem for an iterative solver cannot be described sufficiently by the condition number $\kappa(A) = |||A|||\,|||A^{-1}|||$ only. Depending on the method used, the convergence behavior is influenced by several characteristics of the system matrix, like the eigenspectrum, the pseudo-eigenspectrum, the singular values, or the distance from normality, and by characteristics of the right-hand side vector and the initial approximation to the solution [GvL83, NRT90, Tre, vdV89, vdV92, DvdV91]. Ill-conditioned then means that iterative solvers have trouble, e.g., need many iterations, suffer from cancellation, break down, stall.

Not only these relationships are still poorly understood, but the characteristics of the matrices are hard to evaluate. For instance, dense eigenvalue algorithms become unstable and too memory intensive for even moderately sized sparse matrices. Sparse eigenvalue algorithms, on the other hand, use the same basic principles as iterative solvers, so that their results cannot be trusted to analyze bad condition for the latter.

The unsymmetric linear systems arising in device simulation in the Gummel solution of the continuity equations and in the coupled solution are reputed to be very ill-conditioned [BCD+89]. Considerable efforts have been made in the investigation of variable transformations that improve the relative scaling of the variables and thus alleviate this problem [Pin90]. Unpreconditioned iterative methods generally fail to show any acceptable convergence behavior. Only incomplete factorization (ILU-like) preconditioners improve the condition enough so that a good iterative algorithm can solve the system in a reasonable number of iterations.

In some cases, these preconditioners work quite well for most of the linear systems occurring in one simulation but fail for a few particularly ill-conditioned systems. Sometimes problems with an insufficiently accurate solution can be overcome by the outer nonlinear iteration or even by the time-step control in some way or another, but until recently, some simulations could simply not be performed because some of the linear systems were impossible to solve [RSAR91, HPWF91]. The new approximate factorization preconditioner with numerical dropping presented in Section 6.9.4 improves the condition sufficiently and allows these experiments to be performed [PF91a, PF91b].

2.9 Exploiting the computer architecture

The storage requirements alone (see Section 2.7) call for the use of supercomputers for large applications requiring the solution of linear systems. The available resources and the turn-around time are the only limiting factors that prevent even larger problem sizes.

Large transient device simulations run up to a week on minisupercomputers and several hours on the fastest available supercomputers. The linear solves are the dominating operation, taking more than 95 percent of the total execution time [Hei91].

Hence, it is important that the solver exploits the architectural features of such machines as much as possible, that is, vectorization and parallelization. Most successful preconditioners show very little parallelism by nature. A lot of effort has been spent in the last years to increase the degree of parallelism in regular problems (e.g., finite difference discretizations) [DDSvdV91], but the difficulties are much more substantial for irregular problems.

2.10 Flexibility

As mentioned in Section 2.8, some combinations of an iterative method and a preconditioner appear to be fastest on most linear systems in one simulation, but fail on a few systems. Other combinations may solve these particularly ill-conditioned systems, but are too slow to be used for the other systems. A flexible solver package selects the more expensive method only for those systems that faster methods cannot solve.

Similar flexibility issues come to light when "trying out" new methods, or the effect of variations of the methods. Many researchers in iterative methods base all their analysis on artificial model problems only, and some of their "results" lead to disappointment when their methods are applied to real-world problems.

3

Iterative Methods

The first part of this chapter (Sections 3.1 to 3.3) describes some iterative methods for the solution of symmetric positive-definite linear systems and culminates in the description of the Conjugate Gradients method. Their presentation is included mainly to give the theoretical framework for some of the preconditioners in Chapter 4 and for the iterative methods for unsymmetric systems that constitute the rest of this chapter. The derivations in this chapter are based on [GvL83, HJ85, dH86, SS86, vdV92, Gut91]. The number of iterative methods that have been presented, especially in the last few years, is far too large for the scope of this chapter. Only methods with some relevance to and experimental experience with device simulation are presented. See [Elm82, Saa82, Saa89, NRT90, Gut90b, AMS90, Hac91, FGN92] for similar, possibly more complete overviews of iterative methods for unsymmetric systems.

All the iterative methods presented in this chapter have been published and implemented several times by different authors. In this chapter, they are all displayed in a consistent notation ready for immediate implementation. The derivations view the algorithms as methods to find a stationary point of a given function, where this point is selected to coincide with the solution of the linear system (1.1). This does not always correspond exactly to the ideas of the original authors, but the resulting algorithms are the same. Some of the algorithms can also be viewed as oblique projection methods or as variants of Lanczos algorithms.

Each section of this chapter describes one particular method. It starts with the theoretical derivation, presents the algorithm, and discusses its typical convergence behavior on unsymmetric systems from device simulation. Section 3.13 summarizes the resource requirements for the different methods.

All relevant iterative methods for the solution of unsymmetric linear systems are Krylov subspace methods and can be seen as generalizations of the Conjugate Gradients method [HS52]. A primary distinction between these methods is which matrix is used to construct the Krylov subspace. Normal equation methods use the matrix $A^T A$ and are described in Section 3.4. Orthogonalization methods (Sections 3.5 to 3.7) use A itself, and biorthogonalization methods (Sections 3.8 to 3.12) use $\begin{bmatrix} A & 0 \\ 0 & A^T \end{bmatrix}$.

All the convergence behavior plots in this chapter refer to the same linear system. It arises from the coupled Newton solver in a 3-D simulation of a trench DRAM[7] cell, using 15k grid points. This system was chosen because it exhibits many of the characteristics that occur typically in device simulation. All nonsymmetric methods are applied with split D-ILU preconditioning (see Sections 4.2 and 4.4.3). The plots sketch the relative residual norm $||r_k||_2/||b||_2$ as a function of the iteration number. Note that the first such plot appears only in Section 3.5, as the methods explained before that section are incapable of solving the system.

3.1 Fixpoint search: Matrix Splitting Methods

Assume that the system matrix A is split into the form

$$A = M - N ,$$

where M is nonsingular. The solution $x^\star$ of Equation (1.1) is then the **fixpoint** of the function $\mathcal{F}$ defined by

$$\begin{array}{rcl} \mathcal{F} : \mathbb{R}^n & \rightarrow & \mathbb{R}^n \\ x & \mapsto & \mathcal{F}(x) = M^{-1}(Nx + b) . \end{array}$$

Theorem 3.1 *[GvL83, HY81] If the spectral radius of the matrix $M^{-1}N$ satisfies $\rho(M^{-1}N) < 1$, then the iteration $x_{k+1} = \mathcal{F}(x_k)$ converges to the fixpoint $x^\star$ for every starting iterate x_0.*

---

[7] Dynamic random access memory.

Some standard matrix splitting methods and their basic iteration schemes are listed below. D stands for the diagonal of A, and L and U are the strictly lower and upper triangular parts of A, respectively, so that $A = L + D + U$.

- Jacobi:

$$x_{k+1} = D^{-1}(b - (L+U)x_k)$$

- Gauss-Seidel (GS):

$$x_{k+1} = (D+L)^{-1}(b - Ux_k)$$

- Symmetric Gauss-Seidel (SGS):

$$\begin{aligned} x_{k+\frac{1}{2}} &= (D+L)^{-1}(b - Ux_k) \\ x_{k+1} &= (D+U)^{-1}\left(b - Lx_{k+\frac{1}{2}}\right) \end{aligned}$$

- Successive Over-Relaxation (SOR):

$$x_{k+1} = (D+\omega L)^{-1}(\omega b - ((1-\omega)D - \omega U)x_k)$$

- Symmetric Successive Over-Relaxation (SSOR):

$$\begin{aligned} x_{k+\frac{1}{2}} &= (D+\omega L)^{-1}(\omega b - ((1-\omega)D - \omega U)x_k) \\ x_{k+1} &= (D+\omega U)^{-1}\left(\omega b - ((1-\omega)D - \omega L)x_{k+\frac{1}{2}}\right) \end{aligned}$$

SOR and SSOR are generalizations of GS and SGS, respectively, and equivalent to them for $\omega = 1$. The form of SOR used above is also called forward SOR. Backward SOR is obtained by exchanging the roles of L and U in the iteration formula. The symmetrized form SSOR arises by applying one forward SOR step immediately followed by one backward SOR step. The optimum value of the relaxation parameter ω is known for some particular types of problems.

These standard iterative methods do not play any significant role in the solution of linear systems in device simulation, as the splitting matrices usually do not fulfill the requirements for convergence, and if they do, the Conjugate Gradient method presented below in Section 3.3 converges faster anyway. They are listed here only because of their similarity to some preconditioning methods presented in Chapter 4. See Hageman and Young [HY81] for a complete analysis of these methods.

3.2 Minimizing a convex function: Steepest Descent

An iterative solver is expected to find a "closer" approximation to the solution at each iteration. This means that $||x^\star - x_k||$ (where $|| \bullet ||$ is some appropriate norm) gets smaller in every iteration.

Definition. *A function* $|| \bullet || : \mathbb{R}^n \rightarrow \mathbb{R}$ *is a* **vector norm** *in* $\mathbb{R}^n$ *if for all vectors* $x, y \in \mathbb{R}^n$ *and for all scalars* $\alpha \in \mathbb{R}$:

$$\begin{aligned} ||x|| &\geq 0 && (3.1)\\ ||x|| = 0 &\Leftrightarrow x = 0 && (3.2)\\ ||\alpha x|| &= |\alpha|\, ||x|| && (3.3)\\ ||x+y|| &\leq ||x|| + ||y|| && (3.4) \end{aligned}$$

It is easy to see that every norm is convex[8], so that the function

$$\varphi(x) = ||x^\star - x||$$

has one single global minimum and no local minima. The global minimum occurs in the point $x = x^\star$. Looking for the solution of Equation (1.1) is thus equivalent to minimizing the function φ.

The convexity ensures that, unless we have found the solution already, there is always a direction in which the function φ decreases:

$$x \neq x^\star \quad \Rightarrow \quad \exists p \neq 0 : \; \varphi(x+p) < \varphi(x) \; . \qquad (3.5)$$

Such a direction is called a **descent direction**. Since the (one-dimensional) function $f(\alpha) = \varphi(x + \alpha p)$ is also convex, there is exactly one point that minimizes φ when moving from x into the direction p. The value of α at this minimum is called the **line search parameter**. If the function f is continuously differentiable, the minimum must occur at a point where f is stationary:

$$\frac{d}{d\alpha} f(\alpha) = 0 \; ,$$

that is, a point where the directional derivative of φ in the direction p is zero.

---

[8] **Proof.** $$0 \leq \alpha \leq 1 : \; \begin{aligned} ||\alpha x + (1-\alpha) y|| &\leq ||\alpha x|| + ||(1-\alpha)\, y|| \\ &= |\alpha|\, ||x|| + |1-\alpha|\, ||y|| \\ &= \alpha\, ||x|| + (1-\alpha)\, ||y|| \end{aligned} \qquad \square$$

Definition. *The* **directional derivative** *of a function* $\varphi : \mathbb{R}^n \to \mathbb{R}$ *at* x *in the direction* $p \neq 0$ *is*

$$D\varphi(x;p) = \lim_{\varepsilon \to 0} \frac{\varphi(x+\varepsilon p) - \varphi(x)}{\varepsilon} ,$$

if this limit exists. The **right-sided directional derivative** $D^+\varphi(x;p)$ *is defined similarly by using* $\lim_{\varepsilon \to 0^+}$, *and the* **left-sided directional derivative** *by using* $\lim_{\varepsilon \to 0^-}$.

If the directional derivative at x *exists for every direction* p *and depends linearly on* p, *then* φ *is called* **differentiable** *at* x. *In this case, the gradient of* φ *at* x *is defined by*

$$\nabla\varphi(x) = \left[\ D\varphi(x;e_0)\ ;\ D\varphi(x;e_1)\ ;\ \ldots\ ;\ D\varphi(x;e_{n-1})\ \right]^T ,$$

and the directional derivative can also be written as

$$D\varphi(x;p) = p^T \nabla\varphi(x) .$$

If the gradient can be computed analytically, an implicit equation for the exact line search parameter α is obtained:

$$p^T \nabla\varphi(x+\alpha p) = 0 . \tag{3.6}$$

If the function φ is differentiable, then φ decreases most in the direction opposite to the gradient $\nabla\varphi$. This direction $(-\nabla\varphi)$ is also called the **steepest descent direction.**

The idea behind the **method of steepest descent** is to select the next approximation as the minimum along the direction of steepest descent, i.e., to select

$$x_{k+1} = x_k - \alpha_k \nabla\varphi(x_k) , \tag{3.7}$$

where α_k is found by substituting $p = -\nabla\varphi(x_k)$ in condition (3.6):

$$(\nabla\varphi(x_k))^T \nabla\varphi(x_k - \alpha_k \nabla\varphi(x_k)) = 0 . \tag{3.8}$$

Equation (3.6) is applicable only if φ is differentiable along the ray $\{x + \alpha p | \alpha > 0\}$. Due to the homogeneity property (3.3), no norm is differentiable at the zero vector[9]. The gradient of the function φ defined by $\phi(x) = \|x^\star - x\|$ does not exist in the minimum $x = x^\star$ itself.

The derivation (3.5)-(3.8) still holds for some other convex functions that have their only minimum in $x = x^\star$, in particular for the square of a norm of the error.

Definition. *$x \in \mathbb{R}$ is a* **stationary point** *of the function $\varphi : \mathbb{R}^n \to \mathbb{R}$ if φ is differentiable at x and*

$$\nabla\varphi(x) = 0 \,. \tag{3.9}$$

Theorem 3.2 *The only stationary point of a squared norm $\| \bullet \|^2$ is 0.*

Proof. The directional derivative of $\| \bullet \|^2$ at the point x in the direction $p \neq 0$ is

$$D(\|x;p\|^2) = \lim_{\varepsilon \to 0} \frac{\|x + \varepsilon p\|^2 - \|x\|^2}{\varepsilon} \,.$$

At the point $x = 0$ this leads to

$$D(\|0;p\|^2) = \lim_{\varepsilon \to 0} \frac{\|\varepsilon p\|^2}{\varepsilon} = \lim_{\varepsilon \to 0} \frac{|\varepsilon|^2}{\varepsilon}\|p\|^2 = \lim_{\varepsilon \to 0} \varepsilon\|p\|^2 = 0$$

and thus $\nabla\varphi(0) = 0$.

At any other point $x \neq 0$, the directional derivative in the direction x can be computed as

$$D(\|x;x\|^2) \quad = \quad \lim_{\varepsilon \to 0} \frac{\|(1+\varepsilon)x\|^2 - \|x\|^2}{\varepsilon}$$

---

[9] **Proof.** The right-sided directional derivative of a vector norm at the point 0 in the direction $p \neq 0$ is

$$D^+\|0;p\| = \lim_{\varepsilon \to 0+} \frac{\|0 + \varepsilon p\| - \|0\|}{\varepsilon} = \lim_{\varepsilon \to 0+} \frac{\|\varepsilon p\|}{\varepsilon} = \lim_{\varepsilon \to 0+} \frac{|\varepsilon|}{\varepsilon}\|p\| = \|p\| \,,$$

but the left-sided directional derivative of $\| \bullet \|$ in the direction p is

$$D^-\|0;p\| = \lim_{\varepsilon \to 0-} \frac{\|0 + \varepsilon p\| - \|0\|}{\varepsilon} = \lim_{\varepsilon \to 0-} \frac{\|\varepsilon p\|}{\varepsilon} = \lim_{\varepsilon \to 0-} \frac{|\varepsilon|}{\varepsilon}\|p\| = -\|p\| \,.$$

□

$$
\begin{aligned}
&= \lim_{\varepsilon \to 0} \frac{(1+\varepsilon)^2\|x\|^2 - \|x\|^2}{\varepsilon} \\
&= \lim_{\varepsilon \to 0} \frac{2\varepsilon + \varepsilon^2}{\varepsilon}\|x\|^2 \\
&= 2\|x\|^2 \neq 0 \,.
\end{aligned}
$$

Since $D(\|x; x\|^2) = x^T \nabla\varphi(x)$, $\nabla\varphi(x)$ cannot be the zero vector. □

By selecting our convex function φ as $\varphi(x) = \|x^\star - x\|^2$, where $\| \bullet \|$ is a norm that is continuously differentiable everywhere except in $\mathbb{R}^n - \{0\}$, the gradient is defined everywhere.

Example. *The l_1- and l_∞-norms cannot be used in this derivation because of their lack of differentiability. For the Euclidean norm (l_2-norm), the gradient of the function $\varphi_{l_2}(x) = \|x^\star - x\|_2^2$ is*

$$\nabla\varphi_{l_2}(x) = 2x - 2x^\star \,.$$

Inserted in condition (3.8), we get

$$\alpha_k = \frac{1}{2} \,.$$

Steepest descent using the Euclidean norm thus consists only of setting

$$x_1 = x_0 - \frac{1}{2}(2x_0 - 2x^\star)$$

and converges in one single step, but is not useful since the gradient cannot be computed.

To find a realistic steepest descent method, we have to choose another type of norm for φ, a norm whose gradient can be computed.

Consider the bilinear form

$$
\begin{aligned}
\langle \bullet, \bullet \rangle_H : \mathbb{R}^{n\times n} &\to \mathbb{R} \\
(v, w) &\mapsto \langle v, w \rangle_H := v^T H w
\end{aligned}
\tag{3.10}
$$

and its quadratic form

$$
\begin{aligned}
\langle \bullet \rangle_H : \mathbb{R}^{n} &\to \mathbb{R} \\
v &\mapsto \langle v \rangle_H := \langle v, v \rangle_H = v^T H v \,.
\end{aligned}
\tag{3.11}
$$

Definition. *A matrix* $H \in \mathbb{R}^{n \times n}$ *is* **symmetric positive definite** *(***s.p.d.***) iff* $H = H^T$ *and*

$$\forall v \in \mathbb{R}^n : x \neq 0 \Rightarrow \langle v \rangle_H > 0 \ .$$

If the matrix H is symmetric positive-definite, then the form $\langle \bullet, \bullet \rangle$ is an **inner product**, and the square-root of the quadratic form $\sqrt{\langle \bullet \rangle_H}$ is a norm [HJ85] (also called the **energy norm**). Note that $\| \bullet \|_2 \equiv \sqrt{\langle \bullet \rangle_I}$.

Choosing the function $\varphi_H(x) := \langle x^\star - x \rangle_H$ for minimization, the matrix H should now be selected such that the gradient

$$\nabla \varphi_H(x) = (H + H^T)(x - x^\star) \tag{3.12}$$

can be computed. The easiest way to achieve this is to require that A itself is symmetric positive-definite and to choose $H = A$. The gradient becomes

$$\begin{aligned} \nabla \varphi_A(x) &= 2A(x - x^\star) \\ &= 2(Ax - b) \\ &= -2r \end{aligned} \tag{3.13}$$

The steepest descent direction $(-\nabla \varphi_A)$ is thus the residual vector[10] r. The condition for exact line search (3.6) becomes

$$r_k^T (r_k - \alpha_k A r_k) = 0 \ .$$

The steepest descent method is now almost ready for implementation. We can save the effort to compute the residual from (1.5) in each iteration. Multiplying (3.7) by $(-A)$ from the left and adding b on each side leads to the recursion

$$r_{k+1} = r_k - \alpha_k A r_k \ . \tag{3.14}$$

Algorithm 3.1 lists the entire algorithm for steepest descent.

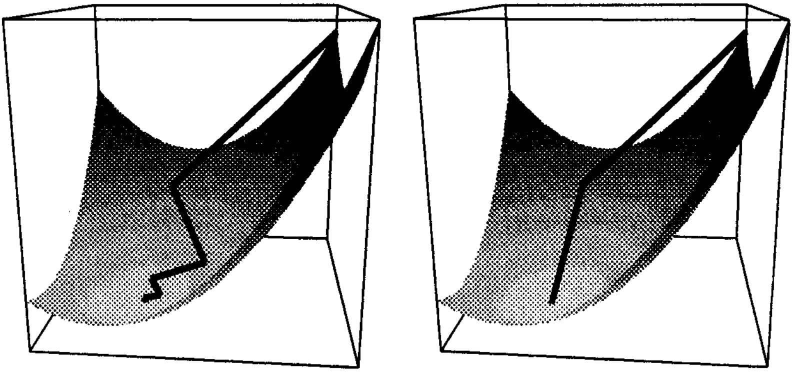
The Steepest Descent method reduces the norm $\langle \epsilon_k \rangle_A$ in every iteration. However, this reduction can be very small. Figure 3.1(a) shows the typical sequence of iterates, for a problem of size $n = 2$.

---

[10] We drop the scaling factor (-2) here to simplify notation.

```
$r_0 := b - Ax_0$
for $k := 0, 1, \ldots$ until convergence do
    $\alpha_k := \frac{r_k^T r_k}{r_k^T A r_k}$
    $x_{k+1} := x_k + \alpha_k r_k$
    $r_{k+1} := r_k - \alpha_k A r_k$
end for
```

Algorithm 3.1: *Steepest Descent.*



(a) Steepest Descent

(b) Conjugate Gradients

Figure 3.1: *Descent behavior on a problem of size 2. The vertical axis shows the value of the function* φ_A.

3.3 Using previous history when minimizing a convex function: Conjugate Gradients

In the two-dimensional problem shown in Figure 3.1, the steepest descent iterates (3.1(a)) bounce between the walls of a narrow valley without making any real progress. They reuse only two search directions (the initial residual and the direction orthogonal to it). This type of behavior is also called **hemstitching** [Avr76].

Previous iterations have already found a minimum of φ_A in a given direction, although from a different starting point. This fact should be exploited. We know that each x_k minimizes φ_A in the space $\{x_{k-1}\}$ + span $\{p_{k-1}\}$, and we want to take advantage of this fact. Observe that

$$x_k \in \{x_0\} + \text{span}\,\{p_0, p_1, \ldots, p_{k-1}\} \,. \tag{3.15}$$

It would be useful (and avoid the problem mentioned above) to find a method for which x_k minimizes φ_A over the whole subspace spanned by the previous residuals (shifted by x_0, as in Relation (3.15)[11]).

From the update formula for the residual (3.14), we can see that the k-th residual of the steepest descent method can be written as

$$r_k = \Phi_k(A)\, r_0 \,, \tag{3.16}$$

where $\Phi_k(A)$ is a polynomial of degree k in A.

Definition. *For $A \in \mathbb{R}^{n \times n}$, $v \in \mathbb{R}^n$, $k \in \mathbb{N}$, the subspace of $\mathbb{R}^n$ defined by*

$$\mathcal{K}_k(A, v) = \text{span}\left\{v, Av, A^2v, \ldots, A^{k-1}v\right\} \tag{3.17}$$

is called a **Krylov subspace**.

Using Equations (3.16) and (3.17), Equation (3.15) can be rewritten as

$$x_k \in \{x_0\} + \mathcal{K}_k(A, r_0) \,.$$

The basic idea behind the **Conjugate Gradient** method (CG) [HS52] is to select the k-th search direction p_k such that it points from the minimum of φ_A in the k-th Krylov subspace to the minimum of φ_A in the $(k+1)$-th Krylov subspace. The new approximation x_{k+1} can thus be derived from x_k through a line search into the direction p_k:

$$x_{k+1} = x_k + \alpha_k p_k \,. \tag{3.18}$$

As x_k now minimizes φ_A over the whole shifted Krylov subspace $\{x_0\} + \mathcal{K}_k(A, r_0)$, the gradient of φ_A at this minimum (and thus the residual r_k, because of (3.13)) must be orthogonal to this subspace:

$$r_k \perp \mathcal{K}_k(A, r_0) \,. \tag{3.19}$$

---

[11] In the remainder of this chapter, mentioning a subspace in the solution space always refers to the subspace shifted by x_0.

The k first search directions or the k first residuals form alternative bases for the k-dimensional Krylov subspace:

$$\mathcal{K}_k(A, r_0) = \text{span}\{p_0, p_1, \ldots, p_{k-1}\} = \text{span}\{r_0, r_1, \ldots, r_{k-1}\} . \qquad (3.20)$$

Lemma 3.3 *Unless the exact solution has been found already, Conjugate Gradients reduces $\varphi_A(x_k)$ in each iteration.*

Proof.

$$\begin{aligned} x_k \neq x^\star \quad &\Rightarrow \quad \nabla\varphi_A(x_k) = -2r_k \neq 0 \\ &\Rightarrow \quad \exists\varepsilon > 0 : \varphi_A(\underbrace{x_k + \varepsilon r_k}_{\in\mathcal{K}_{k+1}(A,r_0)}) < \varphi_A(x_k) \quad \square \end{aligned}$$

The orthogonality condition (3.19) thus implies that the new residual is orthogonal to all previous search directions and all previous residuals:

$$\forall i < k : r_k^T p_i = r_k^T r_i = 0 . \qquad (3.21)$$

The residual $r_k = b - Ax_k$ can still be updated in a similar way as in the steepest descent method. Multiplying the update relation for x_{k+1}, (3.18), on the left by $(-A)$, and adding b, we get:

$$r_{k+1} = r_k - \alpha_k A p_k . \qquad (3.22)$$

Using this residual update in (3.21),

$$\forall i < k : \underbrace{p_i^T r_{k+1}}_{=0} = p_i^T(r_k - \alpha_k A p_k) = \underbrace{p_i^T r_k}_{=0} - \alpha_k p_i^T A p_k ,$$

and knowing that (unless we have found the exact solution) $\alpha_k \neq 0$ because of Lemma 3.3, leads to the **conjugacy relation** of Conjugate Gradients: all the search directions are A-conjugate:

$$\forall i < k : p_i^T A p_k = 0 . \qquad (3.23)$$

Now that we have established the basic relations (3.21) and (3.23) for Conjugate Gradients, we can find the direction p_k that meets the requirements for the update of the solution (3.18). If the minimum of φ_A in $\mathbb{R}^n$ has not yet been found, p_k must have a non-vanishing component in the direction of r_k.

Using the previous search directions as basis for the k-th Krylov subspace $\mathcal{K}_k(A, r_0)$, the new search direction $p_k \in \mathcal{K}_{k+1}(A, r_0)$ can be expressed as

$$p_k = r_k + \sum_{i=0}^{k-1} \beta_{ki} p_i \; . \tag{3.24}$$

The coefficients of this linear combination are found using (3.23):

$$\begin{aligned} p_i^T A p_k = 0 \quad &\Rightarrow \quad p_i^T A \left(r_k + \sum_{j=0}^{k-1} \beta_{kj} p_j \right) = 0 \\ &\Rightarrow \quad p_i^T A r_k + \sum_{j=0}^{k-1} \beta_{kj} \underbrace{p_i^T A p_j}_{=0 \text{ for } i \neq j} = 0 \\ &\Rightarrow \quad \beta_{ki} = -\frac{p_i^T A r_k}{p_i^T A p_i} \; . \end{aligned}$$

Because of the construction of the search directions for (3.20), the Krylov subspace can also be written as

$$\mathcal{K}_k(A, r_0) = \text{span}\left\{ p_0, Ap_0, Ap_1, \ldots, Ap_{k-2} \right\} ,$$

so that the orthogonality condition (3.19) annihilates all the β_{ki} except for $\beta_{k,k-1}$. The new search direction is thus a linear combination of the residual and the previous search direction only, so the other search directions do not need to be stored:

$$p_k = r_k + \beta_k p_{k-1} \tag{3.25}$$

The coefficient for p_{k-1} in this update relation is

$$\beta_k := \beta_{k,k-1} = -\frac{p_{k-1}^T A r_k}{p_{k-1}^T A p_{k-1}} \; .$$

The line search parameter α_k follows from (3.21) and (3.22):

$$\begin{aligned} p_k^T r_{k+1} = 0 \quad &\Rightarrow \quad p_k^T (r_k - \alpha_k A p_k) = 0 \\ &\Rightarrow \quad \alpha_k = \frac{p_k^T r_k}{p_k^T A p_k} \; . \end{aligned}$$

Some further transformations using (3.21), (3.22), (3.25), and the symmetry of A save a few vector dot products in the computation of the scalars α_k and β_k while delivering the Euclidean norm of r_k for convergence control:

$$\begin{aligned} r_k^T r_k &= (r_{k-1} - \alpha_{k-1} A p_{k-1})^T r_k &&= -\alpha_{k-1} p_{k-1}^T A r_k \\ r_k^T r_k &= (p_k - \beta_k p_{k-1})^T r_k &&= p_k^T r_k \\ r_{k-1}^T r_{k-1} &= p_{k-1}^T r_{k-1} = p_{k-1}^T (r_k + \alpha_{k-1} A p_{k-1}) &&= \alpha_{k-1} p_{k-1}^T A p_{k-1} \end{aligned}$$

The entire Conjugate Gradient algorithm is listed in Algorithm 3.2.

```
$r_0 := b - Ax_0$
for $k := 0, 1, \ldots$ until convergence do
    if $k = 0$ then
        $p_0 := r_0$
    else
        $p_k := r_k + \frac{r_k^T r_k}{r_{k-1}^T r_{k-1}} p_{k-1}$
    end if
    $\alpha_k := \frac{r_k^T r_k}{p_k^T A p_k}$
    $x_{k+1} := x_k + \alpha_k p_k$
    $r_{k+1} := r_k - \alpha_k A p_k$
end for
```

Algorithm 3.2: *Conjugate Gradients.*

By the above construction, and the fact that

$$\langle x^\star - x_k \rangle_A = \epsilon_k^T A \epsilon_k = \epsilon_k^T r_k$$

follows:

Theorem 3.4 *The k-th iteration of Conjugate Gradients finds an approximation x_k that minimizes $\epsilon_k^T r_k = \langle x^\star - x_k \rangle_A$ over the shifted Krylov subspace $\{x_0\} + \mathcal{K}_k(A, r_0)$.*

Conjugate Gradients has a monotonic (and usually smooth) convergence behavior for solving s.p.d. systems. CG solves the two-dimensional example of Figure 3.1(b) in only two iterations.

Corollary 3.5 *Conjugate Gradients terminates with the correct solution after at most n iterations.*

In exact arithmetic, Conjugate Gradients needs $2n^3 + O(n^2)$ flops to find the exact solution of a dense linear system. This is only by a constant factor higher than the $\frac{1}{3}n^3 + O(n^2)$ flops needed to perform Cholesky factorization. In the classical reference [HS52], Hestenes and Stiefel mainly compare Conjugate Gradients with n or $n+1$ iterations to elimination methods, but they also include theoretical and experimental evidence that *"justifies the procedure of stopping the algorithm before the final step is reached."* See [GvL83, CGO76, GO89, Gut90b] for more about the history of Conjugate Gradients.

3.4 Making a general matrix look s.p.d.: Conjugate Gradients applied to the normal equations

The crucial point in the derivation of Conjugate Gradients is the restriction to problems where the system matrix is symmetric and positive definite. One obvious step [HS52] to permit the use of Conjugate Gradients on general problems is to change the problem in order to make the system matrix s.p.d.

Theorem 3.6 *Let $A \in \mathbb{R}^{n \times n}$ be non-singular. The matrix $A^T A$ is symmetric positive definite.*

Proof. $A^T A$ is symmetric: $(A^T A)^T = A^T (A^T)^T = A^T A$.
$A^T A$ is positive definite: Let $v \in \mathbb{R}^n$, $v \neq 0$:

$$\langle v \rangle_{A^T A} = v^T (A^T A) v = (v^T A^T)(Av) = (Av)^T (Av) = \|Av\|_2^2 > 0 .$$

$\square$

Instead of solving the system $Ax = b$ directly, we can apply Conjugate Gradients to the system

$$A^T A x = A^T b , \tag{3.26}$$

which is s.p.d. Its solution is the same as for the original system:

$$(A^T A)^{-1} A^T b = A^{-1} A^{-T} A^T b = x^\star .$$

Equation (3.26) is usually called the **normal equations.**

There are several variants of **Conjugate Gradients applied to the normal equations.** The most common version [Elm82, DDSvdV91], known under the abbreviation **CGNR** (Conjugate Gradients applied to the normal equations minimizing the residual) can be derived immediately from Algorithm 3.2. It is listed as Algorithm 3.3. A numerically more stable version, called LSQR, has been suggested by Paige and Saunders [PS82a].

```
$r_0 := b - Ax_0$
for $k := 0, 1, \ldots$ until convergence do
    if $k = 0$ then
        $p_0 := A^T r_0$
    else
        $p_k := A^T r_k + \frac{(A^T r_k)^T (A^T r_k)}{(A^T r_{k-1})^T (A^T r_{k-1})} p_{k-1}$
    end if
    $\alpha_k := \frac{(A^T r_k)^T (A^T r_k)}{(Ap_k)^T (Ap_k)}$
    $x_{k+1} := x_k + \alpha_k p_k$
    $r_{k+1} := r_k - \alpha_k A p_k$
end for
```

Algorithm 3.3: *CGNR.*

CGNR minimizes the function $\varphi_{A^T A}$ at each iteration. Since

$$\langle x^\star - x_k \rangle_{A^T A} = \epsilon_k^T A^T A \epsilon_k = r_k^T r_k \,,$$

Theorem 3.4 implies that the following holds:

Theorem 3.7 *The k-th iteration of CGNR finds an approximation x_k that minimizes the Euclidean norm of the residual ($\|r_k\|_2$, or $\langle x^\star - x_k \rangle_{A^T A}$) over the shifted Krylov subspace $\{x_0\} + \mathcal{K}_k(A^T A, A^T r_0)$.*

The condition number of $A^T A$ is the square of the condition number of A, and so CGNR makes ill-conditioned systems much worse [DDSvdV91]. The plot for the coupled DRAM system selected as a test example in this section was omitted here, as the convergence curve levels off at $\|r_k\|/\|b\| \approx 0.72$.

3.5 Minimizing the residual: GCR

Instead of minimizing the convex function φ_{A^TA} over the Krylov subspace $\{x_0\} + \mathcal{K}_k(A^TA, A^Tr_0)$, as CGNR does, we will now minimize the same function φ_{A^TA} over the Krylov subspace that Conjugate Gradients uses, namely $\{x_0\} + \mathcal{K}_k(A, r_0)$. The method is known as **Generalized Conjugate Residuals** or **GCR** [Elm82].

Similar to Conjugate Gradients, GCR performs a line search along the search direction p_k, which is chosen to be colinear to $(x_{k+1} - x_k)$, where x_k and x_{k+1} minimize φ_{A^TA} in the k-th and the $(k+1)$-th Krylov subspaces, respectively. The update relations for the approximation to the solution (3.18) and for the residual (3.22) as well as the alternative bases for the Krylov subspace (3.20) still hold.

It is easy to see that $\varphi_{A^TA}(x_k)$ cannot increase in GCR. A property analogous to Lemma 3.3, however, cannot be stated. It is possible that x_k also minimizes φ_{A^TA} in $\mathcal{K}_{k+1}(A, r_0)$. In this case the algorithm stalls with $\alpha_k = 0$.

From Equation (3.12) the gradient of the function φ_{A^TA} is

$$\nabla\varphi_{A^TA}(x) = -2A^Tr \ . \tag{3.27}$$

Since $\varphi_{A^TA}(x_k)$ is minimal over $\mathcal{K}_k(A, r_0)$, the gradient is orthogonal to this subspace:

$$A^Tr \perp \mathcal{K}_k(A, r_0) \ . \tag{3.28}$$

Using the alternative bases from Equation (3.20) in (3.28), we have shown that the residuals in GCR are A-conjugate to the previous residuals and the previous search directions:

$$\forall i < k : r_k^TAp_i = r_k^TAr_i = 0 \ . \tag{3.29}$$

Assuming that the algorithm does not yet stall at the k-th iteration, we find that the search directions are (A^TA)-conjugate:

$$\forall i < k : p_i^TA^TAp_k = 0 \ . \tag{3.30}$$

Proof.

$$i < k : \underbrace{r_{k+1}^TAr_i}_{=0} = \underbrace{r_k^TAr_i}_{=0} - \underbrace{\alpha_k}_{\neq 0} p_k^TA^TAp_i \ . \qquad \square$$

The new search direction p_k is selected as the current residual plus a linear combination of the previous search directions, as in Equation (3.24). Using (3.24) in the conjugacy relation (3.30)

$$\underbrace{p_i^T A^T A p_k}_{=0} = p_i^T A^T A r_k + \sum_{j=0}^{k-1} \beta_{kj} \underbrace{p_i^T A^T A p_j}_{=0 \text{ for } i \neq j}$$

gives the coefficients of the previous search directions:

$$\beta_{ki} = -\frac{p_i^T A^T A r_k}{p_i^T A^T A p_i} . \tag{3.31}$$

The residual update (3.22) in Equation (3.29) gives the value of the line search parameter:

$$\alpha_k = \frac{r_k^T A p_k}{p_k^T A^T A p_k} . \tag{3.32}$$

The algorithm stalls with $\alpha_k = 0$ whenever r_k gets orthogonal to Ap_k. If continued anyway, GCR will inevitably break down in the $(k+1)$-th iteration because $p_{k+1} = 0$.

By construction we have proven:

Theorem 3.8 *The k-th iteration of GCR finds an approximation x_k that minimizes the Euclidean norm of the residual ($\|r_k\|_2$, or $\langle x^\star - x_k \rangle_{A^T A}$) over the shifted Krylov subspace $\{x_0\} + \mathcal{K}_k(A, r_0)$ (unless the algorithm breaks down before).*

Unfortunately Equation (3.31) allows no further simplification that would lead to a similarly simple update relation as (3.25) for Conjugate Gradients. Instead, all the previous search directions p_i are needed in the computation of the new search direction. The associated requirements both in storage (for all the previous p_i as well as all the vectors Ap_i) and computation increase unacceptably with the iteration number k. In fact, it is not possible to construct a general method satisfying Theorem 3.8 with limited storage requirements [FM84]. GCR is therefore usually restarted at regular intervals. Algorithm 3.4 lists GCR(ℓ), which is restarted every ℓ iterations and thus keeps up to ℓ back-vectors p_i.

```
$r_0 := b - Ax_0$
for $k := 0, 1, \ldots$ until convergence do
    $p_k := r_k - \sum_{i=\lfloor \frac{k}{\ell} \rfloor \ell}^{k-1} \frac{(Ar_k)^T(Ap_i)}{(Ap_i)^T(Ap_i)} p_i$
    $\alpha_k := \frac{r_k^T Ap_k}{(Ap_k)^T(Ap_k)}$
    $x_{k+1} := x_k + \alpha_k p_k$
    $r_{k+1} := r_k - \alpha_k Ap_k$
end for
```

Algorithm 3.4: *GCR(ℓ).*

The need for the vector Ap_k does not mean that two matrix-vector multiplications are needed in each iteration. Ap_k can be obtained by multiplying the assignment for p_k on the left by A:

$$Ap_k := Ar_k - \sum_{i=\lfloor \frac{k}{\ell} \rfloor \ell}^{k-1} \frac{(Ar_k)^T(Ap_i)}{(Ap_i)^T(Ap_i)} Ap_i \,. \tag{3.33}$$

Direct multiplication by A is still faster if the summation involves many terms. The critical number of terms for which this occurs depends on the density of the matrix and on the machine. For the typical values of the restarting interval ℓ, the variant with Equation (3.33) is the most efficient.

The minimization property derived above does not hold under restarting, and all that is left from Theorem 3.8 is

Corollary 3.9 *The k-th iteration of GCR(ℓ) finds (unless the algorithm breaks down before) an approximation x_k that minimizes the Euclidean norm of the residual ($\|r_k\|_2$, or $\langle x^* - x_k \rangle_{A^TA}$) over the shifted Krylov subspace* $\{x_{\lfloor \frac{k}{\ell} \rfloor \ell}\} + \mathcal{K}_k(A, r_{\lfloor \frac{k}{\ell} \rfloor \ell})$.

Figure 3.2 shows that the number of iterations to achieve a given accuracy increases drastically when GCR is restarted. Although more back-vectors generally lead to lower iteration numbers, the reverse effect, due to numerical instability, can also occur (compare the curves for GCR(15) and GCR(20)).

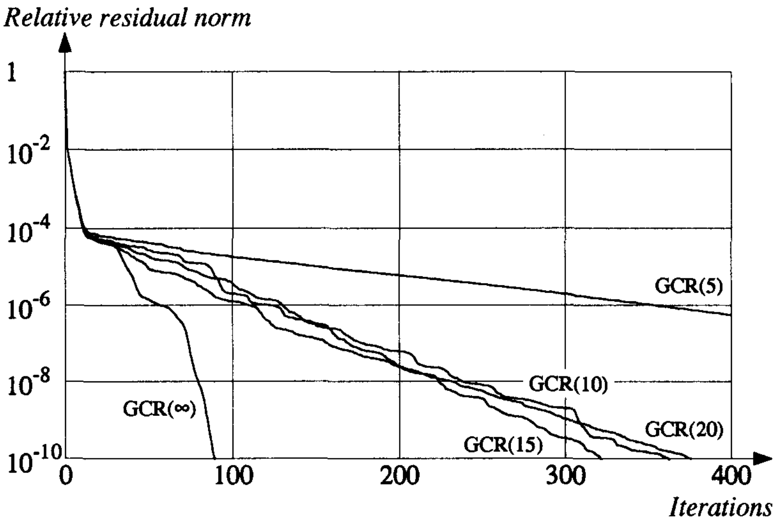


Figure 3.2: *Convergence behavior of full GCR (lower curve) and restarted GCR, with restarts every 5, 10, 15, or 20 iterations.*

3.6 Truncating GCR: Orthomin

Instead of throwing away all the previous search directions every ℓ iterations, as restarting does, we save the same amount of memory by just throwing the oldest search direction, keeping the ℓ most recent directions. This approach is called **truncation**, and the truncated version of GCR is called **Orthomin(ℓ)** [Vin76]. It is listed in Algorithm 3.5.

Theorem 3.8 becomes

Corollary 3.10 *The k-th iteration of Orthomin(ℓ) finds (unless the algorithm breaks down before) an approximation* x_k *that minimizes the Euclidean norm of the residual* ($\|r_k\|_2$, *or* $\langle x^\star - x_k\rangle_{A^TA}$) *over the shifted Krylov subspace* $\{x_0\} + \mathcal{K}_k(A, r_0)$ *for* $k \leq \ell$, *and over the shifted subspace spanned by the* ℓ *latest search directions,* $\{x_{k-\ell}\} + \operatorname{span}\{p_{k-\ell}, \ldots, p_{k-1}\}$, *for* $k > \ell$.

Figure 3.3 plots the convergence behavior of Orthomin(ℓ) for different values of the truncation parameter ℓ. In comparison to Figure 3.2, it can be

```
$r_0 := b - Ax_0$
for $k := 0, 1, \ldots$ until convergence do
    $p_k := r_k - \sum_{i=\max\{0,k-\ell\}}^{k-1} \frac{(Ar_k)^T(Ap_i)}{(Ap_i)^T(Ap_i)} p_i$
    $\alpha_k := \frac{r_k^T Ap_k}{(Ap_k)^T(Ap_k)}$
    $x_{k+1} := x_k + \alpha_k p_k$
    $r_{k+1} := r_k - \alpha_k Ap_k$
end for
```

Algorithm 3.5: *Orthomin(ℓ).*

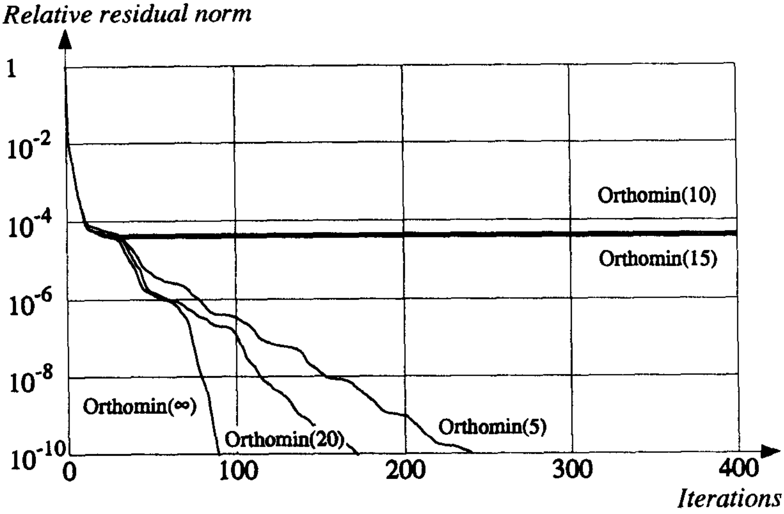


Figure 3.3: *Convergence behavior of full Orthomin (lower curve) and truncated Orthomin, with 5, 10, 15, or 20 back vectors.*

seen in Figure 3.3 how truncating can prevail over restarting by increasing the convergence speed (for $\ell = 5$ and $\ell = 20$). It also shows how this method can stall because of near-breakdown (for $\ell = 10$ and $\ell = 15$).

3.7 GCR with an orthonormal basis: GMRES

The determination (3.32) of the line search parameter in GCR is precarious, as the algorithm may stall and possibly break down whenever r_k gets almost orthogonal to Ap_k. This problem can be avoided by selecting a different basis for the Krylov subspace. The **Generalized Minimal Residual Method** (GMRES) [SS86] constructs an orthonormal basis for $\mathcal{K}_k(A, r_0)$.

Let $\mathcal{P}_k = \{p_0, p_1, \ldots, p_{k-1}\}$ be an alternative basis for the Krylov subspace:

$$\operatorname{span}\{\mathcal{P}_k\} = \mathcal{K}_k(A, r_0) . \tag{3.34}$$

Let us assume $\mathcal{P}_k$ is orthonormal. Writing the p_i as the columns of an $n \times k$-matrix $P_k = \begin{bmatrix} p_0 \,;\, p_1 \,;\, \ldots \,;\, p_{k-1} \end{bmatrix}$, this means that

$$P_k^T P_k = I . \tag{3.35}$$

Equations (3.34) and (3.35) hold also for $k = 1$, so that the first basis vector has to be colinear to the initial residual r_0 (and, of course, have unit length). It can thus be chosen as

$$p_0 = \frac{1}{\|r_0\|_2} r_0 .$$

For any other iteration $k \geq 1$, we need a basis vector p_k in the difference of the Krylov subspaces $\mathcal{K}_{k+1}(A, r_0) - \mathcal{K}_k(A, r_0)$. Unless the two subspaces are equal, in which case the algorithm has converged already, the vector Ap_{k-1} is contained in this difference. Hence, it can be expressed in the new basis $\mathcal{P}_{k+1}$ as

$$Ap_{k-1} = P_{k+1} h_{0:k,k-1} , \tag{3.36}$$

where $h_{0:k,k-1} = \begin{bmatrix} h_{0,k-1} \,;\, h_{1,k-1} \,;\, \ldots \,;\, h_{k,k-1} \end{bmatrix}^T$ is the representation of Ap_{k-1} in this basis. The coefficients for the previous basis vectors can be computed as

$$\forall i < k : h_{i,k-1} = p_i^T A p_{k-1} . \tag{3.37}$$

Equation (3.36) is rearranged in the explicit form

$$\bar{p}_k := h_{k,k-1}p_k = Ap_{k-1} - P_k h_{0:k-1,k-1} \ . \tag{3.38}$$

The new basis vector p_k is obtained by using the values computed from (3.37) for the right-hand side of Equation (3.38) and selecting $h_{k,k-1} = \|\bar{p}_k\|$. Instead of using the **classical Gram-Schmidt** orthogonalization algorithm (Equations (3.37) and (3.38)), **modified Gram-Schmidt** is a numerically stabler method to compute $h_{0:k,k-1}$ and p_k [GvL83].

In each previous iteration $i < k$, a similar vector $h_{0:i,i-1}$ has been computed. Extending each $h_{0:i,i-1}$ with $k - i$ zeros, we obtain a $(k+1) \times k$ upper Hessenberg matrix

$$\bar{H}_k := \begin{bmatrix} h_{0,0} & h_{0,1} & \cdots & h_{0,k-2} & h_{0,k-1} \\ h_{1,0} & h_{1,1} & \cdots & h_{1,k-2} & h_{1,k-1} \\ 0 & h_{2,1} & \ddots & \vdots & \vdots \\ \vdots & 0 & \ddots & h_{k-2,k-2} & \vdots \\ \vdots & \vdots & \ddots & h_{k-1,k-2} & h_{k-1,k-1} \\ 0 & 0 & \cdots & 0 & h_{k,k-1} \end{bmatrix} .$$

Expressing each Ap_i in the style of Equation (3.36) and assembling these expressions in matrix form leads to the important matrix equation

$$AP_k = P_{k+1}\bar{H}_k \ . \tag{3.39}$$

The k-th approximation x_k should be an element of the shifted Krylov subspace $\{x_0\} + \mathcal{K}_k(A, r_0)$ and can be written as

$$x_k = x_0 + \sum_{i=0}^{k-1} \alpha_{ik} p_i = x_0 + P_k \alpha_k \ . \tag{3.40}$$

Note that $\alpha_k = [\ \alpha_{0k}\ ; \alpha_{1k}\ ; \ldots ; \alpha_{k-1,k}\ ]^T$ in (3.40) is a vector of length k rather than a scalar.

The k-th residual is similarly given by

$$r_k = r_0 - AP_k\alpha_k \ , \tag{3.41}$$

but using (3.39) in (3.41) leads to a cheaper way to compute the residual:

$$r_k = r_0 - P_{k+1}\bar{H}_k\alpha_k \ . \tag{3.42}$$

If both x_k and r_k need to be computed and k is large enough, then r_k should be computed through its defining Equation (1.5), instead of using Equation (3.42).

The coefficient vector α_k is selected such that the approximation x_k minimizes the function

$$\varphi_{A^TA}(x) = \langle x^\star - x \rangle_{A^TA} = \|r\|^2$$

in $\mathcal{K}_k(A, r_0)$. Noticing that $r_0 = \|r_0\| P_{k+1} e_0$, and since the vectors in $\mathcal{P}_{k+1}$ are orthonormal (Equation (3.35) for $k+1$), we see that

$$\|r_k\| = \left\| P_{k+1} \left(\|r_0\| e_0 - \bar{H}_k \alpha_k \right) \right\| = \left\| \|r_0\| e_0 - \bar{H}_k \alpha_k \right\| \ .$$

α_k is the solution of the least squares problem

$$\min_{\alpha_k \in \mathbf{R}^k} \left\| \|r_0\| e_0 - \bar{H}_k \alpha_k \right\|^2 \ . \tag{3.43}$$

This least squares problem is solved in the standard way [GvL83], constructing an orthogonal $(k+1) \times (k+1)$ matrix $\bar{Q}_{k+1}$ and a $(k+1) \times k$ upper triangular matrix $\bar{R}_k$ such that

$$\bar{Q}_k \bar{H}_k = \bar{R}_k \ .$$

Let Q_k be the upper left $k \times k$ submatrix of $\bar{Q}_k$, and let R_k be the submatrix obtained by removing the last, entirely zero row of $\bar{R}_k$. The solution of the least squares problem (3.43) is given by

$$\alpha_k = R_k^{-1} Q_k \left(\|r_0\| e_0 \right) \ .$$

The Hessenberg matrix $\bar{H}_{k+1}$ for the next iteration is obtained by extending $\bar{H}_k$ by a zero row and then by the nonzero column $h_{0:k+1,k}$. Similarly, the triangular matrix $\bar{R}_k$ is a submatrix of $\bar{R}_{k+1}$. The orthogonal matrix $\bar{Q}_{k+1}$ is obtained from $\bar{Q}_k$ by premultiplicating it with a single Givens rotation. In practice, $\bar{H}_k$ is never stored entirely; only R_k and the $2k$ scalars for the k Givens rotations that define $\bar{Q}_k$, as well as the $k+1$ basis vectors, are kept in memory.

Since the relative contributions α_{ik} of all the previous basis vectors p_i change in every iteration, each new x_k can only be recomputed through (3.40). There are no short recurrence formulae for x_k and r_k. However, for monitoring the convergence behavior, the current x_k or r_k are not needed, as the residual norm of the linear system is equal to the minimum residual of the least squares problem, given by

$$\|r_k\| = e_k^T \bar{Q}_k \left(\|r_0\| e_0 \right) \ . \tag{3.44}$$

Thus, only the final approximation to the solution is computed from (3.40).

GMRES does not need to store the previous vectors Ap_i and therefore requires only half as much storage as an efficient implementation of GCR. All the previous basis vectors p_i, however, still need to be stored. The algorithm is therefore usually restarted every ℓ iterations. Algorithm 3.6 outlines the unrestarted version of GMRES. See [SS86, Wal88, JC91] for more detail about the implementation of GMRES.

```
$r_0 = b - Ax_0$
$p_0 = \frac{1}{\|r_0\|_2} r_0$
for $k := 0, 1, \ldots$ until convergence do
    $h_{0:k,k} := P_{k+1}^T A p_k$
    $\bar{p}_{k+1} := Ap_k - P_{k+1} h_{0:k,k}$
    $h_{k+1,k} := \|\bar{p}_{k+1}\|_2$
    $p_{k+1} := \frac{1}{h_{k+1,k}} \bar{p}_k$
    update $\bar{Q}_{k+1}$ and $\bar{R}_{k+1}$ such that $\bar{Q}_{k+1} \bar{H}_{k+1} = \bar{R}_{k+1}$
    $\|r_{k+1}\|_2 := e_{k+1}^T \bar{Q}_{k+1} \left(\|r_0\|_2 e_0\right)$
end for
$\alpha_k := R_k^{-1} Q_k \left(\|r_0\|_2 e_0\right)$
$x_k := x_0 + P_k \alpha_k$
```

Algorithm 3.6: *Outline of GMRES.*

GMRES is generally considered the most stable of the family of orthogonalization methods, which includes GCR and Orthomin as presented above, as well as GCG [CGO76], Orthores, and Orthodir [YJ80]. By construction, full GMRES requires the minimum number of matrix-vector products to minimize the residual in a given Krylov subspace and cannot break down.

If minimizing the residual is not the primary goal, GMRES may be less than optimal. Variants may be better in reducing the solution error [DFW90].

In practice, the storage requirements make full GMRES infeasible for large problems. Even if the storage is available, the number of linear operations per full GMRES iteration grows linearly with the iteration number and thus

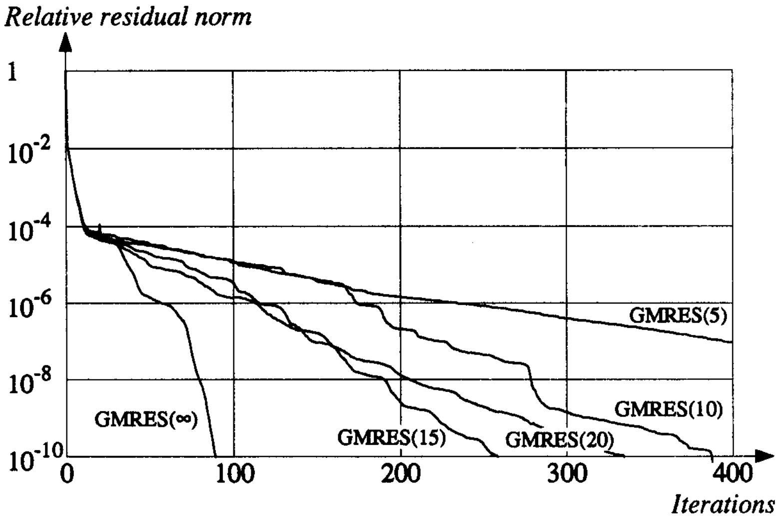


Figure 3.4: *Convergence behavior of full GMRES (lower curve) and restarted GMRES, with restarts every 5, 10, 15, or 20 iterations.*

dominates the total operation count over sparse matrix-vector multiplication. Restarted GMRES often does not solve the linear systems in device simulation, or requires a very large number of iterations. It is difficult to find the optimal parameter ℓ for which GMRES(ℓ) still converges [VvdV92]. For too low values, the restarted algorithm often stalls or proceeds only very slowly.

3.8 Abandoning convergence security: Biconjugate Gradients

The k-th iteration of full GMRES finds the optimal approximation in $\{x_0\} + \mathcal{K}_k(A, r_0)$ (with respect to the residual norm), and this fact guarantees an improvement in each iteration. The restarted or truncated variants of this method preserve this monotonic convergence behavior, for the price of giving up optimality in the full Krylov subspace. **Biconjugate Gradients (BiCG)** [Fle76] abandons monotonic convergence, but it keeps some form of optimality in a larger Krylov subspace of which $\{x_0\} + \mathcal{K}_k(A, r_0)$ is a projection.

BiCG solves not only the primal linear system (1.1) alone, but also the dual linear system

$$A^T \hat{x} = \hat{b} \,. \tag{3.45}$$

In other words, BiCG solves the combination of the linear systems (1.1) and (3.45):

$$\begin{bmatrix} A & 0 \\ 0 & A^T \end{bmatrix} \begin{bmatrix} x \\ \hat{x} \end{bmatrix} = \begin{bmatrix} b \\ \hat{b} \end{bmatrix} . \tag{3.46}$$

By using the notation

$$B := \begin{bmatrix} A & 0 \\ 0 & A^T \end{bmatrix} , \qquad y := \begin{bmatrix} x \\ \hat{x} \end{bmatrix} , \qquad c := \begin{bmatrix} b \\ \hat{b} \end{bmatrix} ,$$

the combined system (3.46) can be rewritten as

$$By = c \,. \tag{3.47}$$

Consider now the function $\varphi_H(y) := \langle B^{-1}c - y \rangle_H$, where H is defined by $H := \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix}$. The matrix H is symmetric by construction and is non-singular if A is non-singular, but H is possibly indefinite. Therefore, the function φ_H may not be convex, and the solution of (3.47), the vector $B^{-1}c$, does not necessarily minimize φ_H. However, looking at the gradient computed from Equation (3.12), we see that $B^{-1}c$ is the only stationary point of φ_H:

$$\nabla \varphi_H(y) = 2H(y - B^{-1}c) \,. \tag{3.48}$$

Biconjugate Gradients now hunts after a stationary point which is not necessarily a minimum. In imitation of Theorem 3.4, BiCG tries to find at the k-th iteration a vector y_k that is stationary with respect to the Krylov subspace $\mathcal{K}_k(B, s_0)$ (where $s_0 = c - By_0$)[12]. This means that the gradient of φ_H at y_k should be orthogonal to this Krylov subspace:

$$\nabla \varphi_H(y_k) \perp \mathcal{K}_k(B, s_0) \,. \tag{3.49}$$

To complete the notation[13], let's write the residual of the combined system as $s_k = c - By_k$. BiCG proceeds by search directions denoted by q_k. The

---

[12] In other words, y_k is a stationary point of φ_H under the constraint $y_k \in \{y_0\} + \mathcal{K}_k(B, s_0)$.

[13] The alert reader will have noticed by now the typographical rule to denote vectors and matrices relative to the combined system (3.47).

search direction q_k is selected to point from the stationary point with respect to $\mathcal{K}_k(B, s_0)$ to the stationary point with respect to $\mathcal{K}_{k+1}(B, s_0)$, so that a line search suffices:

$$y_{k+1} = y_k + \alpha_k q_k \ . \tag{3.50}$$

From Equation (3.50) it follows that the residual can be updated by

$$s_{k+1} = s_k - \alpha_k B q_k \ . \tag{3.51}$$

The residuals or the search directions can be used as alternative bases to the Krylov subspace:

$$\mathcal{K}_k(B, s_0) = \text{span}\{q_0, q_1, \ldots, q_{k-1}\} = \text{span}\{s_0, s_1, \ldots, s_{k-1}\} \ . \tag{3.52}$$

Defining an additional duality matrix $J := \begin{bmatrix} 0 & I \\ I & 0 \end{bmatrix}$ gives the relation between the two important matrices B and H:

$$H = JB \ . \tag{3.53}$$

The gradient of φ_H from Equation (3.48) can now be rewritten as

$$\nabla\varphi_H(y_k) = 2(JBy_k - Jc) = -2Js_k \ , \tag{3.54}$$

so that the orthogonality condition (3.49) becomes

$$Js_k \perp \mathcal{K}_k(B, s_0) \ . \tag{3.55}$$

Using the alternative bases of Equation (3.52) gives the **biorthogonality relation** of BiCG:

$$\forall i < k : q_i^T J s_k = s_i^T J s_k = 0 \ . \tag{3.56}$$

Since the gradient $\nabla\varphi_H(y_k)$ does in general not belong to $\mathcal{K}_{k+1}(B, s_0)$, the constrained stationary points y_k and y_{k+1} are not necessarily different, even if y_k is not equal to the global stationary point $B^{-1}c$ (an equivalent to Lemma 3.3 does not exist). In this case, s_k is biorthogonal to itself: $\langle s_k \rangle_J = 0$.

In the present formulation of BiCG, the algorithm breaks down if this occurs. Breakdown condition can be avoided through a reformulation of the algorithm. See Section 5.3 for what can be done to avoid this problem (which occurs rarely in practice). For now, we assume that no such breakdown occurs, i.e. $\alpha_k q_k \neq 0$. We then find from (3.51), (3.56), and (3.53) that

$$\forall i < k : \underbrace{q_i^T J s_{k+1}}_{=0} = q_i^T J(s_k - \alpha_k B q_k) = \underbrace{q_i^T J s_k}_{=0} - \underbrace{\alpha_k}_{\neq 0} q_i^T \underbrace{JB}_{=H} q_k ,$$

giving the biconjugacy relation of BiCG:

$$\forall i < k : q_i^T H q_k = 0 . \tag{3.57}$$

Since we assumed that the new y_{k+1} is different from y_k, the new search direction $q_k \in \mathcal{K}_{k+1}(B, s_0)$ can be expressed as a linear combination of the residual s_k and the previous search directions (that form a basis for $\mathcal{K}_k(B, s_0)$). Since y_k is a stationary point of φ_H in $\mathcal{K}_k(B, s_0)$, q_k must have a non-vanishing component in the direction of $B^k s_0$. As our assumption includes that $\mathcal{K}_k(B, s_0)$ is not B-invariant, this implies that the component in the direction of s_k cannot vanish. Normalizing this s_k-component, we get

$$q_k = s_k + \sum_{i=0}^{k-1} \beta_{ki} q_i . \tag{3.58}$$

Relation (3.57) helps to find the coefficients β_{ki}:

$$\begin{aligned} q_i^T H q_k = 0 \quad &\Rightarrow \quad q_i^T H \left(s_k + \sum_{j=0}^{k-1} \beta_{kj} q_j \right) = 0 \\ &\Rightarrow \quad q_i^T H s_k + \sum_{j=0}^{k-1} \beta_{kj} \underbrace{q_i^T H q_j}_{=0 \text{ for } i \neq j} = 0 \\ &\Rightarrow \quad \beta_{ki} = - \frac{q_i^T H s_k}{q_i^T H q_i} . \end{aligned}$$

As $Bq_i \in \mathcal{K}_{i+1}(B, s_0)$, a by-product of the biorthogonality relation (3.56) is

$$\forall i < k-1 : (Bq_i)^T J s_k = q_i^T B^T J s_k = q_i^T H s_k = 0 ,$$

using (3.53) and the symmetry of H and J. This simplifies (3.58) to the recurrence formula

$$q_k = s_k + \beta_k q_{k-1} \tag{3.59}$$

with the coefficient

$$\beta_k := \beta_{k,k-1} = -\frac{q_{k-1}^T H s_k}{q_{k-1}^T H q_{k-1}} \ .$$

The line search parameter α_k is computed from (3.56) and (3.51):

$$\begin{aligned} q_k^T J s_{k+1} = 0 \quad &\Rightarrow \quad q_k^T J(s_k - \alpha_k B q_k) = 0 \\ &\Rightarrow \quad \alpha_k = \frac{q_k^T J s_k}{q_k^T J B q_k} = \frac{q_k^T J s_k}{q_k^T H q_k} \end{aligned}$$

The similarity of this derivation to that of Conjugate Gradients in Section 3.3 can be carried on further by simplifying the computation of the scalars α_k and β_k through

$$\begin{aligned} s_k^T J s_k = &\quad s_k^T J s_{k-1} - \alpha_{k-1} s_k^T J B q_{k-1} &&= -\alpha_{k-1} s_k^T H q_{k-1} \\ s_k^T J s_k = &\quad q_k^T J s_k - \beta_k q_{k-1}^T H s_k &&= q_k^T J s_k \\ s_{k-1}^T J s_{k-1} = &\quad q_{k-1}^T J(s_k + \alpha_{k-1} B q_{k-1}) &&= \alpha_{k-1} q_{k-1} H q_{k-1} \end{aligned}$$

This allows us now to formulate the Biconjugate Gradient algorithm in Algorithm 3.7 as a solver for the combined system (3.47). Recalling that we are interested in solving the primal system (1.1) only, and writing

$$s_k = \begin{bmatrix} r_k \\ \hat{r}_k \end{bmatrix} \text{ and } q_k = \begin{bmatrix} p_k \\ \hat{p}_k \end{bmatrix} ,$$

the defining relations for the scalars can be simplified through

$$s_k^T J s_k = \begin{bmatrix} r_k \\ \hat{r}_k \end{bmatrix}^T \begin{bmatrix} 0 & I \\ I & 0 \end{bmatrix} \begin{bmatrix} r_k \\ \hat{r}_k \end{bmatrix} = r_k^T \hat{r}_k + \hat{r}_k^T r_k = 2\hat{r}_k^T r_k$$

and

$$q_k^T H q_k = \begin{bmatrix} p_k \\ \hat{p}_k \end{bmatrix}^T \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} p_k \\ \hat{p}_k \end{bmatrix} = p_k^T A^T \hat{p}_k + \hat{p}_k^T A p_k = 2\hat{p}_k^T A p_k \ .$$

Algorithm 3.8 lists Biconjugate Gradients delivering only the approximation x_k for the solution of the primal system.

```
$s_0 := c - By_0$
for $k := 0, 1, \ldots$ until convergence do
    if $k = 0$ then
        $q_0 := s_0$
    else
        $q_k := s_k + \frac{s_k^T J s_k}{s_{k-1}^T J s_{k-1}} q_{k-1}$
    end if
    $\alpha_k := \frac{s_k^T J s_k}{q_k^T H q_k}$
    $y_{k+1} := y_k + \alpha_k q_k$
    $s_{k+1} := s_k - \alpha_k B q_k$
end for
```

Algorithm 3.7: *Biconjugate Gradients as a solver for the combined system (3.47).*

Theorem 3.11 *The k-th iteration of BiCG finds an approximation y_k to the combined system (3.47) that is the stationary point of φ_H with respect to the shifted Krylov subspace $\{y_0\} + \mathcal{K}_k(B, s_0)$ (unless the algorithm breaks down before).*

Theorem 3.11 has been proven by construction. The primal solution space $\{x_0\} + \mathcal{K}_k(A, r_0)$ is obtained by restricting the combined solution space $\{y_0\} + \mathcal{K}_k(B, s_0)$ to the first n variables, and so Theorem 3.11 meets the optimality promise from the beginning of this section.

However, there is no statement about the minimization of residuals. In practice, there are always some BiCG iterations that increase the residual in norm, even if the entire iteration process converges globally. This manifests itself in a **zigzagging** in the convergence behavior of BiCG, as in Figure 3.5. Zigzagging indicates that a large quantity is added to a small one, and a similarly large quantity is then subtracted again. This can cause cancellation. Section 5.5 outlines a way to deal with this problem.

BiCG often converges (globally) despite the (local) zigzagging effect. BiCG is able to solve many linear systems in device simulation where

```
$r_0 := b - Ax_0$
select $\hat{r}_0$
for $k := 0, 1, \ldots$ until convergence do
  if $k = 0$ then
    $p_0 := r_0$
    $\hat{p}_0 := \hat{r}_0$
  else
    $\beta_k := \frac{\hat{r}_k^T r_k}{\hat{r}_{k-1}^T r_{k-1}}$
    $p_k := r_k + \beta_k p_{k-1}$
    $\hat{p}_k := \hat{r}_k + \beta_k \hat{p}_{k-1}$
  end if
  $\alpha_k := \frac{\hat{r}_k^T r_k}{\hat{p}_k^T A p_k}$
  $x_{k+1} := x_k + \alpha_k p_k$
  $r_{k+1} := r_k - \alpha_k A p_k$
  $\hat{r}_{k+1} := \hat{r}_k - \alpha_k A^T \hat{p}_k$
end for
```

Algorithm 3.8: *Biconjugate Gradients.*

restarted GMRES or truncated Orthomin stagnate, and it often requires much fewer iterations than these methods even if the latter converge. I do not know of any real case in device simulation where exact breakdown occurred in BiCG.

The selection of the initial residual $\hat{r}_0$ for the dual system (3.45) is still free in Algorithm 3.8. It is usually set equal to the primal initial residual,

$$\hat{r}_0 = r_0 , \tag{3.60}$$

which amounts to selecting the initial approximation $\hat{x}_0$ of (3.45) as

$$\hat{x}_0 = A^{-T} A x_0 + A^{-T} (\hat{b} - b) .$$

With this choice, BiCG is identical to CG if the matrix A is symmetric. Choices other than (3.60) are discussed in [DvdV91].

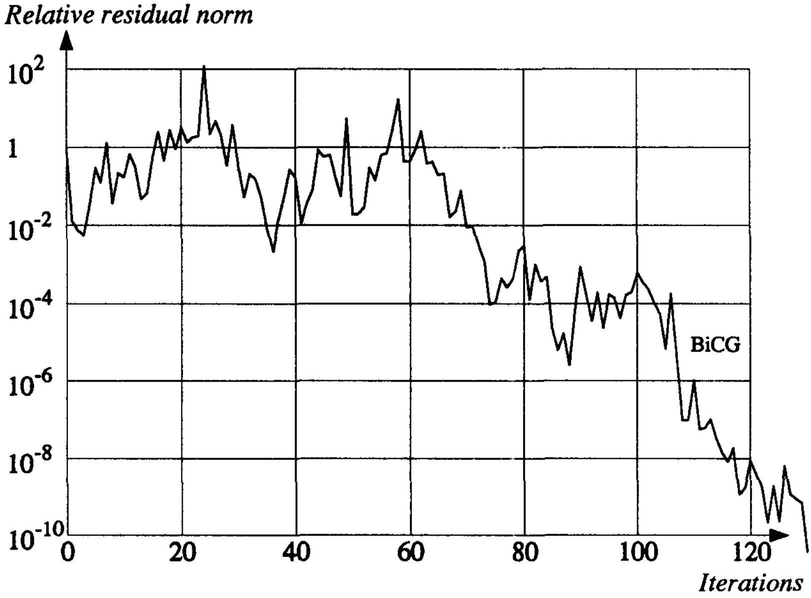


Figure 3.5: *Convergence behavior of BiCG.*

3.9 Squaring the polynomials: CGS

Since the solution of the dual system in BiCG is generally not needed, the dual residual $\hat{r}_k$ and the dual search direction $\hat{p}_k$ are only computed to be used in vector dot products to evaluate the scalars α_k and β_k. Updating these vectors involves one transposed matrix-vector multiplication per iteration (yielding $A^T\hat{p}_k$).

In many data structures for sparse matrices, such transposed operations cannot be implemented as efficiently as (right) matrix-vector multiplication. It is therefore interesting to look for a reformulation of BiCG that avoids transposed matrix operations.

It follows from the update formula for the combined residual (3.51) that the k-th combined residual can be written as the product of a polynomial Φ_k

of degree k in B with the initial combined residual:

$$s_k = \Phi_k(B)\, s_0 \,. \tag{3.61}$$

Observing that powers of B can be written as

$$B^k = \begin{bmatrix} A & 0 \\ 0 & A^T \end{bmatrix}^k = \begin{bmatrix} A^k & 0 \\ 0 & (A^T)^k \end{bmatrix},$$

Equation (3.61) is equivalent to the combination of (3.62) and (3.63):

$$r_k = \Phi_k(A)\, r_0 \tag{3.62}$$

$$\hat{r}_k = \Phi_k(A^T)\, \hat{r}_0 \,. \tag{3.63}$$

In a similar way, the search directions can be written using the polynomials Ψ_k:

$$q_k = \Psi_k(B)\, s_0 \Leftrightarrow \left\{ \begin{array}{lcl} p_k & = & \Psi_k(A)\, r_0 \\ \hat{p}_k & = & \Psi_k(A^T)\, \hat{r}_0 \end{array} \right. .$$

Matrix transposition applied to the argument of a matrix polynomial or applied to the value of the same polynomial are equivalent, so that the inner products that define the scalars in Algorithm 3.8 can also be computed as

$$\hat{r}_k^T r_k = \hat{r}_0^T \Phi_k^2(A)\, r_0 \tag{3.64}$$

$$\hat{p}_k^T A p_k = \hat{r}_0^T A \Psi_k^2(A)\, r_0 \tag{3.65}$$

Equations (3.64) and (3.65) are derived by applying only the usual properties of matrix multiplication and the fact that multiplication of polynomials over the same matrix is commutative.

Squaring the update formulas (3.51) and (3.59) now leads to a simultaneous recurrence for the three product polynomials Φ_k^2, Ψ_k^2, and $\Phi_k \Psi_k$:

$$\Phi_{k+1}^2 = \Phi_k^2 - A\,(2\alpha_k \Phi_k \Psi_k - \alpha_k^2 A \Psi_k^2) \tag{3.66}$$

$$\Phi_{k+1}\Psi_{k+1} = \Phi_{k+1}^2 + \beta_{k+1}(\Phi_k \Psi_k - \alpha_k A \Phi_k \Psi_k) \tag{3.67}$$

$$\Psi_{k+1}^2 = \Phi_{k+1}\Psi_{k+1} + \beta_{k+1}(\Phi_k \Psi_k - \alpha_k A \Phi_k \Psi_k) + \beta_{k+1}^2 \Psi_k^2 \tag{3.68}$$

Of course it would be foolish now to use Equations (3.66), (3.67), and (3.68) to compute the matrices $\Phi_k^2(A)$, $\Psi_k^2(A)$, and $\Phi_k(A)\Psi_k(A)$, which become quite dense as k increases. We should rather map the recurrence into one for vectors defined by

$$u_k = \Phi_k^2(A)r_0\ , \quad v_k = \Phi_k(A)\Psi_k(A)r_0\ , \quad w_k = \Psi_k^2(A)r_0\ .$$

Algorithm 3.9 is a version of BiCG where the scalars can be computed without any transposed matrix operations.

```
$r_0 := b - Ax_0$
select $\hat{r}_0$
for $k := 0, 1, \ldots$ until convergence do
  if $k = 0$ then
    $p_0 := r_0$
    $u_0 := r_0$
    $v_0 := r_0$
    $w_0 := r_0$
  else
    $\beta_k := \frac{\hat{r}_0^T u_k}{\hat{r}_0^T u_{k-1}}$
    $p_k := r_k + \beta_k p_{k-1}$
    $u_k := u_{k-1} - A\ (2\alpha_{k-1}v_{k-1} - \alpha_{k-1}^2 A w_{k-1})$
    $v_k := u_k + \beta_k(v_{k-1} - \alpha_{k-1}Aw_{k-1})$
    $w_k := v_k + \beta_k(v_{k-1} - \alpha_{k-1}Aw_{k-1}) + \beta_k^2 w_{k-1}$
  end if
  $\alpha_k := \frac{\hat{r}_0^T u_k}{\hat{r}_0^T A w_k}$
  $x_{k+1} := x_k + \alpha_k p_k$
  $r_{k+1} := r_k - \alpha_k A p_k$
end for
```

Algorithm 3.9: *Biconjugate Gradients without transposed matrix operations.*

The two matrix-vector products of Algorithm 3.8 (one transposed, one regular) have been replaced in Algorithm 3.9 by three matrix-vectors products (all three without transposition), and there are also some additional linear vector operations. Although this is an overhead of about 50% in the number of floating-point operations, this modification may reduce the execution time on vector- or parallel computers for data structures that allow for high parallel speedup in the regular matrix multiplication, but where the transposed operation is not parallelizable.

The real advantage of the squaring, however, emanates from the following reflection. Assume BiCG does converge in the residual, that is, $||r_k||$ is smaller than $||r_0||$ in some norm. The polynomial $\Phi_k(A)$ thus contracts r_0. There is guarded hope that $\Phi_k(A)$ contracts r_k as well. In other words, the "contraction effect" of $\Phi_k^2(A)$ applied to r_0 is expected to be "stronger" than that of $\Phi_k(A)$.

The **Conjugate Gradients Squared** method (CGS) [Son89] is based on this consideration. The vector u_k from Algorithm 3.9 is regarded as the residual vector in CGS. Looking at the update formula for u_k in Algorithm 3.9, we see that the update to x_k that keeps x_{k+1} consistent with the CGS residual $r_{k+1}^{\mathrm{CGS}} = u_{k+1}$ is the vector $p_k^{\mathrm{CGS}} = (2\alpha_k v_k - \alpha_k^2 A w_k)$. Algorithm 3.10 lists the entire algorithm for CGS.

The "contraction" argument is very informal. The following reflections intend to improve our understanding of this argument (see also Driessen and van der Vorst [DvdV91]). Let w be an eigenvector of A^T, associated with the eigenvalue[14] λ: $A^T w = \lambda w$. w is also an eigenvalue of $\Phi_k(A^T)$, associated with the eigenvalue $\Phi_k(\lambda)$. Consider the projections of our residual vectors on w:

$$\gamma_k := w^T r_k \ .$$

We see[15] that $\gamma_k^{\mathrm{BiCG}} = \Phi_k(\lambda)\gamma_0$ and $\gamma_k^{\mathrm{CGS}} = \Phi_k^2(\lambda)\gamma_0$. So if BiCG reduces (in its first k iterations) this projection (that is, if $|\Phi(\lambda)| < 1$), then CGS reduces the projection even more. On the other hand, if BiCG amplifies the projection, CGS amplifies it even more.

The contraction argument works out on our example. Nearly each converging BiCG iteration in Figure 3.5 results in an even stronger converging CGS iteration in Figure 3.6. On the other hand, diverging BiCG iterations make CGS diverge even more. Observe how the peak at iteration number 24

---

[14] So the complex conjugate $\bar{w}$ is a left eigenvector [HJ85] of A. λ is also an eigenvalue of A, but w is not necessarily an eigenvector of A.

[15] **Proof.**

$$w^T r_k^{\mathrm{BiCG}} = w^T \Phi_k(A) r_0 = \left(\Phi_k(A^T) w\right)^T r_0 = \left(\Phi_k(\lambda) w\right)^T r_0 = \Phi_k(\lambda)(w^T r_0) \ .$$

□

```
$r_0 := b - Ax_0$
select $\hat{r}_0$
for $k := 0, 1, \ldots$ until convergence do
    if $k = 0$ then
        $v_0 := r_0$
        $w_0 := r_0$
    else
        $\beta_k := \frac{\hat{r}_0^T r_k}{\hat{r}_0^T r_{k-1}}$
        $v_k := r_k + \beta_k (v_{k-1} - \alpha_{k-1} A w_{k-1})$
        $w_k := v_k + \beta_k (v_{k-1} - \alpha_{k-1} A w_{k-1}) + \beta_k^2 w_{k-1}$
    end if
    $\alpha_k := \frac{\hat{r}_0^T r_k}{\hat{r}_0^T A w_k}$
    $p_k := 2\alpha_k v_k - \alpha_k^2 A w_k$
    $x_{k+1} := x_k + p_k$
    $r_{k+1} := r_k - A p_k$
end for
```

Algorithm 3.10: *Conjugate Gradients Squared.*

is amplified. The zigzagging effect in CGS is therefore much stronger than in BiCG, and this increases the risk for cancellation problems (see Section 5.5).

CGS converges much faster than BiCG for most problems in device simulation. It often requires only slightly more than half of the number of iterations BiCG needs, which is then close to the maximum performance gain that could be expected through squaring. It makes sense to use CGS even for ill-conditioned s.p.d. systems, rather than CG, because the higher cost per iteration is roughly compensated by the lower iteration count, and CGS can accommodate for apparent unsymmetry and indefiniteness caused by rounding errors.

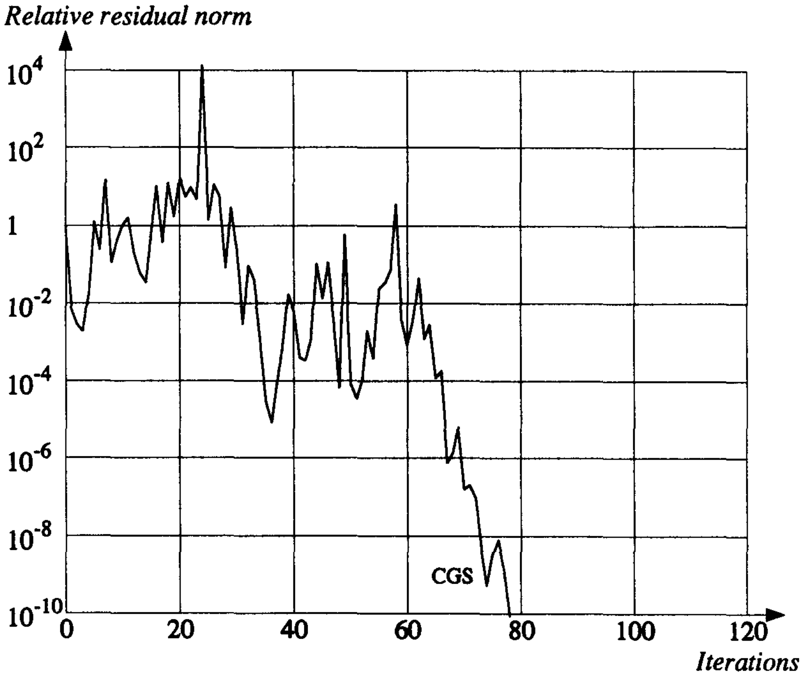


Figure 3.6: *Convergence behavior of CGS.*

3.10 Smoothing the squaring with local minimization: Bi-CGSTAB

CGS applies the operator $\Phi_k(A)$ to the BiCG residual $r_k^{\text{BiCG}} = \Phi_k(A)r_0$ with the expectation that it will contract the BiCG residual. The end of the previous section showed that this hope is not always fulfilled. However, nobody forces us to use $\Phi_k(A)$ as the "contraction operator". Bi-CGSTAB [vdV92] chooses another polynomial Ω_k of degree k and defines

$$r_k^{\text{Bi-CGSTAB}} = \Omega_k(A)\Phi_k(A)r_0 \, .$$

To keep the storage needs from growing with the iteration number, the sequence of polynomials $\Omega_0, \Omega_1, \ldots, \Omega_k, \ldots$ should again be selected such that the residuals $r_k^{\text{Bi-CGSTAB}}$ can be computed through simple recurrences.

The Bi-CGSTAB residual vector is chosen by starting from the point

$$r_{k-\frac{1}{2}} = \Omega_{k-1}(A)\Phi_k(A)\, r_0 \tag{3.69}$$

and performing a single steepest descent step on the residual norm function:

$$\|r_k^{\text{Bi-CGSTAB}}\| = \min_{\omega} \|(I - \omega A) r_{k-\frac{1}{2}}\| \tag{3.70}$$

The polynomial Ω_k is thus defined as

$$\Omega_k(A) = (I - \omega_{k-1}A)(I - \omega_{k-2}A)\cdots(I - \omega_0 A)\ . \tag{3.71}$$

with (using the Euclidean norm in Equation (3.70))

$$\omega_k = \frac{r^T_{k+\frac{1}{2}} A r_{k+\frac{1}{2}}}{r^T_{k+\frac{1}{2}} A^T A r_{k+\frac{1}{2}}}\ . \tag{3.72}$$

Note that the rare case that the denominator in (3.72) vanishes does not entail another source of breakdown. It occurs only if $r_{k+\frac{1}{2}} = 0$, which means that the exact solution has been found in the first "half" of this iteration. A careful implementation avoids problems by leaving the iteration at this point.

The recurrence relations for the product polynomials can now be derived from (3.51), (3.59), and (3.71):

$$\Omega_k\Phi_{k+1} = \Omega_k\Phi_k - \alpha_k A\Omega_k\Psi_k \tag{3.73}$$

$$\Omega_{k+1}\Phi_{k+1} = \Omega_k\Phi_{k+1} - \omega_k A\Omega_k\Phi_{k+1} \tag{3.74}$$

$$\Omega_{k+1}\Psi_{k+1} = \Omega_{k+1}\Phi_{k+1} + \beta_{k+1}(\Omega_k\Psi_k - \omega_k A\Omega_k\Psi_k) \tag{3.75}$$

The CGS choice was motivated by the easy computation of the scalars α_k and β_k. Taking another look back to the properties of BiCG applied to the combined system (3.47) reveals other ways to obtain these scalars. Using the sequence $\{B^i s_0\}_{i=0}^{k-1}$ as basis for the Krylov subspace $\mathcal{K}_k(B, s_0)$ in the biorthogonality condition (3.55) leads to

$$\forall i < k : (B^i s_0)^T J s_k = 0\ . \tag{3.76}$$

Similarly, since $\{q_i\}_{i=0}^{k-1}$ is also a basis for the subspace, every $B^i s_0$ can be written as a linear combination of the q_i's. Using the biconjugacy relation (3.57) then gives

$$\forall i < k : (B^i s_0)^T H q_k = 0\ . \tag{3.77}$$

Let us now express s_k and q_k in the basis $\{B^i s_0\}_{i=0}^{k}$ for $\mathcal{K}_{k+1}(B, s_0)$:

$$s_k = \sum_{i=0}^{k} \mu_{ki} B^i s_0 \text{ and } q_k = \sum_{i=0}^{k} \nu_{ki} B^i s_0 \,. \tag{3.78}$$

Using these representations and eliminating terms according to (3.76) and (3.77) leads to an alternative way to compute the inner products needed for the scalars in Algorithm 3.7:

$$\begin{aligned} s_k^T J s_k &= \mu_{kk} (B^k s_0)^T J s_k \\ q_k^T H q_k &= \nu_{kk} (B^k s_0)^T H q_k \end{aligned}$$

The leading coefficients in (3.78) are

$$\mu_{kk} = \nu_{kk} = \prod_{i=0}^{k-1} (-\alpha_i) = (-1)^k \prod_{i=0}^{k-1} \alpha_i \tag{3.79}$$

Translating the notations for the combined system into the polynomial representation of the primal system, as introduced in Section 3.9, Equations (3.76) and (3.77) are equivalent to

$$\forall i < k : \hat{r}_0^T A^i \Phi_k(A) r_0 = \hat{r}_0^T A^{i+1} \Psi_k(A) r_0 = 0 \,. \tag{3.80}$$

The inner products for the scalars, as they are used in Equations (3.64) and (3.65) and in Algorithms 3.8 and 3.10, are now

$$\begin{aligned} \hat{r}_0^T \Phi_k^2(A) r_0 &= (-1)^k \left(\prod_{i=0}^{k-1} \alpha_i \right) \hat{r}_0^T A^k \Phi_k(A) r_0 \,, & (3.81) \\ \hat{r}_0^T A \Psi_k^2(A) r_0 &= (-1)^k \left(\prod_{i=0}^{k-1} \alpha_i \right) \hat{r}_0^T A^{k+1} \Psi_k(A) r_0 \,. & (3.82) \end{aligned}$$

Since $\Omega_k(A)$ is a polynomial of degree k in A with leading coefficient

$$\xi_{kk} = (-1)^k \prod_{i=0}^{k-1} \omega_i \,,$$

it can be shown, using Relations (3.80), that

$$\begin{aligned} \hat{r}_0^T \Omega_k(A) \Phi_k(A) r_0 &= (-1)^k \left(\prod_{i=0}^{k-1} \omega_i \right) \hat{r}_0^T A^k \Phi_k(A) r_0 \,, & (3.83) \\ \hat{r}_0^T A \Omega_k(A) \Psi_k(A) r_0 &= (-1)^k \left(\prod_{i=0}^{k-1} \omega_i \right) \hat{r}_0^T A^{k+1} \Psi_k(A) r_0 \,. & (3.84) \end{aligned}$$

The computation of the BiCG line search parameter α_k can thus be obtained as the quotient of (3.83) and (3.84) rather than the quotient of (3.81) and (3.82). Similarly, Bi-CGSTAB finds the scalar β_k according to

$$\begin{aligned}
\beta_k &= \frac{\hat{r}_k^T r_k^{\text{BiCG}}}{\hat{r}_{k-1}^T r_{k-1}^{\text{BiCG}}} = \frac{\hat{r}_0^T r_k^{\text{CGS}}}{\hat{r}_0^T r_{k-1}^{\text{CGS}}} \\
&= \frac{\hat{r}_0^T \Phi_k^2(A) r_0}{\hat{r}_0^T \Phi_{k-1}^2(A) r_0} \\
&= \frac{(-1)^k \left(\prod_{i=0}^{k-1} \alpha_i\right) \hat{r}_0^T A^k \Phi_k(A) r_0}{(-1)^{k-1} \left(\prod_{i=0}^{k-2} \alpha_i\right) \hat{r}_0^T A^{k-1} \Phi_{k-1}(A) r_0} \\
&= (-\alpha_{k-1}) \cdot \frac{\hat{r}_0^T A^k \Phi_k(A) r_0}{\hat{r}_0^T A^{k-1} \Phi_{k-1}(A) r_0} \\
&= (-\alpha_{k-1}) \cdot \frac{1}{-\omega_{k-1}} \cdot \frac{(-1)^k \left(\prod_{i=0}^{k-1} \omega_i\right) \hat{r}_0^T A^k \Phi_k(A) r_0}{(-1)^{k-1} \left(\prod_{i=0}^{k-2} \omega_i\right) \hat{r}_0^T A^{k-1} \Phi_{k-1}(A) r_0} \\
&= \frac{\alpha_{k-1}}{\omega_{k-1}} \cdot \frac{\hat{r}_0^T r_k^{\text{Bi-CGSTAB}}}{\hat{r}_0^T r_{k-1}^{\text{Bi-CGSTAB}}} \,.
\end{aligned}$$

Here we do create another possible source of breakdown, as ω_{k-1} may well be zero even far from the final solution. We will return to this problem in Section 5.3.

Bi-CGSTAB, displayed in Algorithm 3.11, does not guarantee monotonic convergence, but it clearly smoothens the convergence behavior in comparison to CGS. This can be seen in Figure 3.7. Observe how the steep peak on iteration 24 could not be flattened entirely.

Very often, Bi-CGSTAB requires less iterations to converge than CGS. There is even a significant number of cases which Bi-CGSTAB solves, while CGS and BiCG fail [PF91a].

```
$r_0 := b - Ax_0$
select $\hat{r}_0$
for $k := 0, 1, \ldots$ until convergence do
    if $k = 0$ then
        $p_0 := r_0$
    else
        $\beta_k := \frac{\alpha_{k-1}}{\omega_{k-1}} \frac{\hat{r}_0^T r_k}{\hat{r}_0^T r_{k-1}}$
        $p_k := r_k + \beta_k (p_{k-1} - \omega_{k-1} A p_{k-1})$
    end if
    $\alpha_k := \frac{\hat{r}_0^T r_k}{\hat{r}_0^T A p_k}$
    $r_{k+\frac{1}{2}} := r_k - \alpha_k A p_k$
    $\omega_k = \frac{r_{k+\frac{1}{2}}^T A r_{k+\frac{1}{2}}}{(A r_{k+\frac{1}{2}})^T (A r_{k+\frac{1}{2}})}$
    $r_{k+1} := r_{k+\frac{1}{2}} - \omega_k A r_{k+\frac{1}{2}}$
    $x_{k+1} := x_{k+\frac{1}{2}} + \alpha_k p_k + \omega_k r_{k+\frac{1}{2}}$
end for
```

Algorithm 3.11: *Bi-CGSTAB*.

3.11 Catching complex eigenvalues: BiCGStab2

A real unsymmetric matrix may have conjugate pairs of complex eigenvalues. The only roots of the polynomial defined in (3.71) as a product of real polynomials of degree one, however, are the real numbers $1/\omega_0$, $1/\omega_1$, $\ldots$, $1/\omega_{k-1}$. The polynomial can be enriched with complex conjugate roots by choosing it as a product of real polynomials of degree two [Gut91]:

$$\Omega_k(A) = (I + \gamma_{k-2} A + \gamma_{k-1} A^2) \cdots (I + \gamma_0 A + \gamma_1 A^2) \qquad (k \text{ even})\ . \quad (3.85)$$

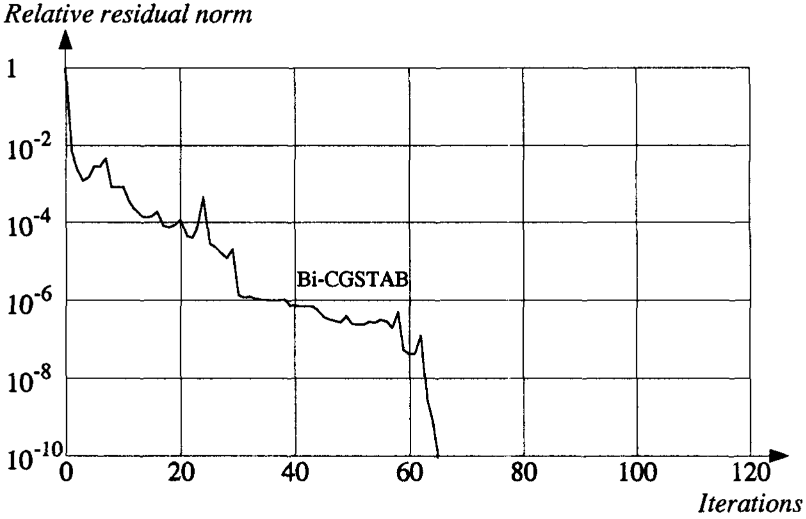


Figure 3.7: *Convergence behavior of Bi-CGSTAB.*

Equation (3.85) defines Ω_k only for even iteration numbers k. BiCGStab2 defines it for odd k's as well, by adding a real root in the same way as in Bi-CGSTAB:

$$\Omega_k(A) = (I - \omega_{k-1}A)\,\Omega_{k-1}(A) \qquad (k \text{ odd})\,. \tag{3.86}$$

This real root is then dropped in the succeeding even iteration. The effort for the odd iteration can be recovered by defining the recursion for the even iteration as

$$\Omega_k(A) = (1 + \zeta_{k-2})\Omega_{k-2}(A) - (\zeta_{k-2}I + \eta_{k-1}A)\Omega_{k-1}(A) \qquad (k \text{ even})\,, \tag{3.87}$$

setting $\gamma_{k-2} = \zeta_{k-2}\omega_{k-1} - \eta_{k-1}$ and $\gamma_{k-1} = \eta_{k-1}\omega_{k-1}$.

We can choose the same names as in Bi-CGSTAB for the vectors obtained by multiplying the matrix-polynomials with the initial residual:

$$\begin{aligned} r_{k-\frac{1}{2}} &= \Omega_{k-1}(A)\Phi_k(A)r_0\,, \\ r_k &= \Omega_k(A)\Phi_k(A)r_0\,, \\ p_k &= \Omega_k(A)\Psi_k(A)r_0\,. \end{aligned}$$

For even iterations, the computation of r_k and p_k according to Equation (3.87) requires another vector:

$$r_{k-\frac{2}{3}} = \Omega_{k-2}(A)\Phi_k(A)r_0 \qquad (k \text{ even}) .$$

The final residual vector for an even iteration is hence given by

$$r_k = r_{k-\frac{2}{3}} - \zeta_{k-2}(r_{k-\frac{1}{2}} - r_{k-\frac{2}{3}}) - \eta_{k-1}Ar_{k-\frac{1}{2}} , \tag{3.88}$$

where ζ_{k-2} and η_{k-1} are chosen to minimize the (Euclidean) residual norm $||r_k||$. To ease notation, define the two-column matrix

$$S = \left[\ (r_{k-\frac{1}{2}} - r_{k-\frac{2}{3}})\ ;\ Ar_{k-\frac{1}{2}}\ \right]$$

and the two-row vector $v = \left[\ \zeta_{k-2}\ ;\ \eta_{k-1}\ \right]^T$, so that (3.88) becomes

$$r_k = r_{k-\frac{2}{3}} - Sv .$$

The square of the residual norm,

$$||r_k||^2 = ||r_{k-\frac{2}{3}}||^2 - 2r_{k-\frac{2}{3}}^T Sv + v^T S^T Sv ,$$

is minimized by

$$v = \begin{bmatrix} \zeta_{k-2} \\ \eta_{k-1} \end{bmatrix} = (S^T S)^{-1} S^T r_{k-\frac{2}{3}} .$$

Again, breakdown can occur if S is singular. Section 5.3 explains when this happens.

The leading coefficient ξ_{kk} is given by

$$\xi_{kk} = \begin{cases} (-1)^k \omega_0\eta_1 \cdots \omega_{k-3}\eta_{k-2}\omega_{k-1} & (k \text{ odd}) \\ (-1)^k \omega_0\eta_1 \cdots \omega_{k-4}\eta_{k-3}\omega_{k-2}\eta_{k-1} & (k \text{ even}) \end{cases}$$

so that the BiCG scalars α_k and β_k can be computed analogously as in Bi-CGSTAB[16]. The further recursion formulae for $r_{k-\frac{1}{2}}$, $r_{k-\frac{2}{3}}$, and p_k are easily derived from (3.51), (3.59), (3.86), and (3.87), leading to Algorithm 3.12.

For our test case, the convergence plot of BiCGStab2 was almost identical to the one of Bi-CGSTAB (shown in Figure 3.7), although the algorithm did truly construct complex conjugate roots in some iterations. In a few other cases, however, BiCGStab2 did require significantly fewer iterations.

---

[16] A further distinction for the signs, as mentioned in the original formulation [Gut91], was avoided with the present selection of variables.

```
$r_0 := b - Ax_0$
select $\hat{r}_0$
for $k := 0, 1, \ldots$ until convergence do
    if $k = 0$ then
        $p_0 := r_0$
    else if $k$ odd then
        $\beta_k := \frac{\alpha_{k-1}}{\omega_{k-1}} \frac{\hat{r}_0^T r_k}{\hat{r}_0^T r_{k-1}}$
        $p_k := r_k + \beta_k(p_{k-1} - \omega_{k-1} A p_{k-1})$
    else
        $\beta_k := \frac{\alpha_{k-1}}{\eta_{k-1}} \frac{\hat{r}_0^T r_k}{\hat{r}_0^T r_{k-1}}$
        $p_k := r_k + \beta_k(1 + \zeta_{k-2})(r_{k-\frac{3}{2}} + \beta_{k-1} p_{k-2})$
                $- \beta_k \zeta_{k-2} p_{k-1} - \beta_k \eta_{k-1} A p_{k-1}$
    end if
    $\alpha_k := \frac{\hat{r}_0^T r_k}{\hat{r}_0^T A p_k}$
    $r_{k+\frac{1}{2}} := r_k - \alpha_k A p_k$
    if $k$ even then
        $\omega_k := \frac{r_{k+\frac{1}{2}}^T A r_{k+\frac{1}{2}}}{(A r_{k+\frac{1}{2}})^T (A r_{k+\frac{1}{2}})}$
        $r_{k+1} := r_{k+\frac{1}{2}} - \omega_k A r_{k+\frac{1}{2}}$
        $x_{k+1} := x_k + \alpha_k p_k + \omega_k r_{k+\frac{1}{2}}$
    else
        $r_{k+\frac{1}{3}} := r_{k-\frac{1}{2}} - \alpha_k A(r_{k-\frac{1}{2}} + \beta_k p_{k-1})$
        $S_k := \left[ \; (r_{k+\frac{1}{2}} - r_{k+\frac{1}{3}}) \; ; \; A r_{k+\frac{1}{2}} \; \right]$
        $\begin{bmatrix} \zeta_{k-1} \\ \eta_k \end{bmatrix} := (S_k^T S_k)^{-1} S_k^T r_{k+\frac{1}{3}}$
        $r_{k+1} := r_{k+\frac{1}{3}} - \zeta_{k-1}(r_{k+\frac{1}{2}} - r_{k+\frac{1}{3}}) - \eta_k A r_{k+\frac{1}{2}}$
        $x_{k+1} := x_k + (1 + \zeta_{k-1}) \alpha_k \beta_k p_{k-1}$
                $+ (1 + \zeta_{k-1})(\alpha_k - \omega_{k-1}) r_{k-\frac{1}{2}}$
                $- \zeta_{k-1} \alpha_k p_k + \eta_k r_{k+\frac{1}{2}}$
    end if
end for
```

Algorithm 3.12: *BiCGStab2.*

3.12 Lanczos' basis for minimization: QMR

The zigzagging convergence behavior of the residual norm in BiCG is due to the fact that BiCG does not try to minimize this residual norm. Convergence in the residual is merely a side effect of the search for a stationary point. A method that minimizes the residual norm over the space spanned by all the BiCG residual vectors (which is $\mathcal{K}_k(A, r_0)$) would be equivalent to GMRES, and have the increasing storage requirement problem of GMRES-like methods.

The residual vectors of BiCG form a basis for $\mathcal{K}_k(A, r_0)$. By normalizing each residual vector, we obtain a non-orthogonal basis for this Krylov subspace. These basis vectors are also called **Lanczos vectors**, as they are generated by Lanczos' biorthogonalization algorithm [Lan50, Gut90b, Gut92].

The **Quasi-Minimal Residual** method (QMR) [FN91] selects the k-th residual r_k as the vector whose norm in the Lanczos basis representation is minimal among all vectors in the Krylov subspace $\mathcal{K}_k(A, r_0)$. Unlike for BiCG, a convergence theorem for QMR exists. The approximations to the solution can be updated through short (though more complicated) recurrences, so that the algorithm has fixed storage requirements. Like BiCG, QMR requires one regular and one transposed matrix-vector product per iteration. Variants of QMR that avoid transposed matrix-vector products or obtain a squaring effect similar to CGS have be presented, but do not seem to be as stable right now [Fre91, CdPvdV91, FGN92].

QMR can be formulated in a way that avoids some of the breakdown conditions of BiCG. We come back to this point in Section 5.3.

3.13 Resource requirements

This section summarizes the resource requirements for the different methods, ignoring preconditioning for now, and assuming that no major timing compromises are made to save storage. Table 3.1 counts the number of occurrences of the main types of operations. Table 3.2 shows the storage overhead in terms of floating-point numbers.

The operation counts per iteration in Table 3.1 have to be multiplied by the number of iterations the method requires to converge. As we have seen, this number of iterations varies significantly from method to method, and

Method	*Matrix-vector products*	*Transposed matrix-vector products*	*Vector dot products*	*Linear operations on vectors*
Jacobi, GS, SOR	(1)	-	-	1
SGS, SSOR	(2)	-	-	2
Steepest descent	1	-	2	4
CG	1	-	2	6
CGNR	1	1	2	6
GCR(ℓ)	1	-	$\frac{1}{2}\ell+2$	$\ell+4$
Orthomin(ℓ)	1	-	$\ell+2$	$2\ell+4$
GMRES(ℓ)	1	-	$\frac{1}{2}\ell+1$	$\ell+3$
BiCG	1	1	2	10
CGS	2	-	2	12
Bi-CGSTAB	2	-	4	12
BiCGStab2	2	-	4.5	18.5

Table 3.1: *Operations per iteration for the methods (ignoring preconditioning). Scalar operations can be neglected, except possibly (when ℓ is large) for the ℓ^2 flops in the back substitution of GMRES(ℓ). For the splitting methods, there are no matrix-vector products, but triangular solves with the same complexity. For GCR(ℓ), GMRES(ℓ), and BiCGStab2, average values are given. The values are smaller for the ℓ first iterations of Orthomin(ℓ) and for the very first iteration of the other methods.*

it depends also on the problem, the preconditioner, and other parameters. Benchmark results are presented in Section 6.13.

Method	*Storage requirements*
Splitting methods	0
Steepest descent	n
CG	$2n$
CGNR	$2n$
GCR(ℓ)	$(2\ell + 1)n$
Orthomin(ℓ)	$(2\ell + 1)n$
GMRES(ℓ)	$\ell n + \frac{1}{2}\ell^2 + O(\ell)$
BiCG	$4n$
CGS	$6n$
Bi-CGSTAB	$4n$
BiCGStab2	$7n$

Table 3.2: *Storage requirements for the iterative methods, expressed in the number of floating-point cells to solve a system of n linear equations. Single scalars are not counted. ℓ is the parameter of some of the methods. It is assumed that the storage for the initial approximation x_0 and the right-hand side b can be overwritten, yielding the final approximation and the final residual on exit.*

Leer - Vide - Empty

4

Preconditioning

4.1 Goal

The number of iterations that a given iterative method needs to solve a system of linear equations $Ax = b$ to a given accuracy depends heavily on the problem under consideration. The conditioning of an unsymmetric linear system for an iterative solver is hard to evaluate (see Section 2.8). If the condition of a problem should be expressed in a single number, the number of iterations a given iterative method needs to reach a given convergence criterion is the only fair choice.

All the methods converge in one single step if the system matrix is the identity matrix. **Preconditioning** transforms the original linear system (1.1) into a **preconditioned system**

$$\widetilde{A}\widetilde{x} = \widetilde{b} \tag{4.1}$$

which should be such that the characteristics (with respect to Section 2.8) of the preconditioned matrix $\widetilde{A}$ are close to the characteristics of the identity.

Assuming there are (isomorphic) operators that transform the original A, x_0, b into the preconditioned $\widetilde{A}$, $\widetilde{x}_0$, $\widetilde{b}$, and the final approximation $\widetilde{x}_k$ to the preconditioned system back into an approximation x_k to the original system, preconditioning could be described as in Algorithm 4.1.

```
.. Transform $A \longmapsto \widetilde{A}$
.. Transform $x_0 \longmapsto \widetilde{x}_0$
.. Transform $b \longmapsto \widetilde{b}$
.. Solve $\widetilde{A}\widetilde{x} = \widetilde{b}$ iteratively (starting with $\widetilde{x}_0$, yielding $\widetilde{x}_k$)
.. Transform $\widetilde{x}_k \longmapsto x_k$
```

Algorithm 4.1: *Preconditioned iterative solution of a linear system.*

One thereby expects that the chosen iterative method is able to solve the preconditioned system (4.1) in fewer iterations than the original system (1.1), even if $\widetilde{A}$ is only a rough approximation of the identity. This approach pays back only if the total time to perform all these transformations and all the needed preconditioned iterations is smaller than the time an iterative method would take on the original system.

Moreover, one preconditioned iteration is usually more expensive than one ordinary iteration, as a matrix-vector product with $\widetilde{A}$ requires more computations than a matrix-vector product with A (see Section 4.7). The choice of one particular preconditioner is therefore a trade-off between cost (in time for the transformations and the preconditioned operations, and also in memory) and effect (in the reduction of the number of iterations).

Ultimately, only the total resource requirements (time and storage) determine the quality of a preconditioner. In Chapter 3, nearly all the important characteristics of different iterative methods could be illustrated on a single linear system that acts as a representative sample for a large number of systems occurring in device simulation. This approach cannot be pursued in this chapter. The quality of different preconditioners depends much stronger on the linear system and the machine architecture in question. Most of the comparisons between preconditioners in this chapter consist of qualitative statements (based on practical experience) and non-numerical data (like idealized costs in memory cells and in flops for single operations). The reader ought to be aware that the quantitative results presented in Sections 4.7.2, 4.7.4, and 4.7.5 should not be misused to draw generalizing conclusions other than those in the text.

For comparison in the next sections, note that costs associated with the original matrix are the following. An $n \times n$ sparse matrix with m nonzero entries requires the storage of m floating-point numbers, plus a significant amount of integer storage for the sparsity structure (see Section 6.6). A matrix-vector product with such a matrix costs m multiplications and $(m - n)$ additions, for a total of $(2m - n)$ flops.

4.2 Position

The preconditioner is usually represented as a preconditioning matrix Q that approximates A, or, more precisely, Q^{-1} approximates A^{-1} in

$$AQ^{-1} \approx I . \tag{4.2}$$

The approximation symbol ($\approx$) in Equation (4.2) is to be interpreted in the sense of Section 4.1.

Equation (4.2) leads to a first form of preconditioning, called **right preconditioning**:

$$[AQ^{-1}]\,[Qx] = b . \tag{4.3}$$

If Equation (4.2) holds, then the matrix $Q^{-1}A$ also approximates the identity in a similar sense, so that **left preconditioning** is another choice:

$$[Q^{-1}A]\,x = [Q^{-1}b] . \tag{4.4}$$

Multiplicative splitting of the preconditioning matrix into $Q = Q_1Q_2$ leads to **split preconditioning**:

$$[Q_1^{-1}AQ_2^{-1}]\,[Q_2x] = [Q_1^{-1}b] . \tag{4.5}$$

The forms (4.3) and (4.4) are in fact special cases of split preconditioning, setting $Q_1 = I$ in (4.5) for right preconditioning and $Q_2 = I$ in (4.5) for left preconditioning [Elm82]. Right preconditioning is sometimes also called **postconditioning**.

4.3 Preconditioned algorithms

The iterative methods presented in Chapter 3 all use the matrix A only as an operator in matrix-vector multiplications (and sometimes transposed matrix-vector multiplications). Therefore, the preconditioned matrix $\widetilde{A}$ is only needed as an operator for preconditioned matrix-vector multiplication. Line 1 of Algorithm 4.1 does not mean that the matrix $\widetilde{A}$ should be computed and stored, but that an operator for preconditioned matrix-vector multiplication should be constructed.

A **preconditioned iterative method** can therefore be implemented just by performing line 4 of Algorithm 4.1 with the selected iterative method, using the operator for preconditioned matrix-vector multiplication instead of the usual operator. In such a straightforward implementation, left preconditioning has the advantage that the **true** approximation of the solution (that is, the approximation to the solution of the original system (1.1)) is always available (since $\widetilde{x}_k = x_k$), whereas right preconditioning keeps the **true** residual always available. The availability of these vectors allows more accurate convergence control.

A trick in the implementation gives right preconditioning both these advantages. Assume the preconditioned residual and approximation are updated as

$$\widetilde{x}_{k+1} = \widetilde{x}_k + \alpha_k \widetilde{p}_k \tag{4.6}$$

$$\widetilde{r}_{k+1} = \widetilde{r}_k - \alpha_k \widetilde{A} \widetilde{p}_k \,. \tag{4.7}$$

For right preconditioning, $\widetilde{A} = AQ^{-1}$, $\widetilde{x}_k = Qx_k$, and $\widetilde{r}_k = r_k$, so that (4.6) and (4.7) become

$$Qx_{k+1} = Qx_k + \alpha_k \widetilde{p}_k \tag{4.8}$$

$$r_{k+1} = r_k - \alpha_k AQ^{-1} \widetilde{p}_k \,. \tag{4.9}$$

Multiplying (4.8) on the left by Q^{-1} yields

$$x_{k+1} = x_k + \alpha_k Q^{-1} \widetilde{p}_k \,. \tag{4.10}$$

The vector $Q^{-1}\widetilde{p}_k$ needed in (4.10) is usually available as an intermediate result in the computation of $AQ^{-1}\widetilde{p}_k$ needed for (4.9).

Except for GMRES, all the methods presented in Chapter 3 contain an update relation similar to (4.6) and (4.7) (although it is not quite obvious in the product methods CGS and Bi-CGSTAB). As an example, Algorithm 4.2 shows the right-preconditioned implementation for Bi-CGSTAB.

A similar trick cannot be used efficiently with left or split preconditioning, as the vector $\widetilde{r}_k$, which then is different from r_k, is needed in the computation of the scalars. Because of the delay before the solution vector is updated, right preconditioned GMRES cannot profit efficiently from this trick.

```
$r_0 := b - Ax_0$
select $\hat{r}_0$
for $k := 0, 1, \ldots$ until convergence do
    if $k = 0$ then
        $p_0 := r_0$
    else
        $\beta_k := \frac{\alpha_{k-1}}{\omega_{k-1}} \frac{\hat{r}_0^T r_k}{\hat{r}_0^T r_{k-1}}$
        $p_k := r_k + \beta_k(p_{k-1} - \omega_{k-1} A Q^{-1} p_{k-1})$
    end if
    $\alpha_k := \frac{\hat{r}_0^T r_k}{\hat{r}_0^T A Q^{-1} p_k}$
    $x_{k+\frac{1}{2}} := x_k + \alpha_k Q^{-1} p_k$
    $r_{k+\frac{1}{2}} := r_k - \alpha_k A Q^{-1} p_k$
    $\omega_k = \frac{r_{k+\frac{1}{2}}^T A Q^{-1} r_{k+\frac{1}{2}}}{(AQ^{-1} r_{k+\frac{1}{2}})^T (AQ^{-1} r_{k+\frac{1}{2}})}$
    $r_{k+1} := r_{k+\frac{1}{2}} - \omega_k A Q^{-1} r_{k+\frac{1}{2}}$
    $x_{k+1} := x_{k+\frac{1}{2}} + \omega_k Q^{-1} r_{k+\frac{1}{2}}$
end for
```

Algorithm 4.2: *Right preconditioned Bi-CGSTAB.*

4.4 A family of incomplete factorizations

The "usual" way of obtaining an operator for A^{-1} is to factorize A into factors that are easily invertible. One practical choice is LDU-factorization, a variant of Gaussian Elimination computing the unit lower triangular matrix L, the diagonal matrix D, and the unit upper triangular matrix U such that

$$A = LDU \; . \tag{4.11}$$

The entries of the factor matrices are computed by

$$\forall i > j: \qquad l_{ij} = \frac{1}{d_{jj}}\left(a_{ij} - \sum_{k=0}^{i-1} l_{ik} d_{kk} u_{kj}\right) \tag{4.12}$$

$$\forall i < j: \qquad u_{ij} = \frac{1}{d_{ii}}\left(a_{ij} - \sum_{k=0}^{j-1} l_{ik} d_{kk} u_{kj}\right) \tag{4.13}$$

$$\forall i: \qquad d_{ii} = a_{ii} - \sum_{k=0}^{i-1} l_{ik} d_{kk} u_{ki} \tag{4.14}$$

The computation of the complete LDU-factorizations of large sparse matrices requires lots of computational and storage resources (This is a main reason to investigate iterative methods, see Section 2.7).

The rationale behind **incomplete factorization preconditioners** is to simplify the factorization rules (4.12) to (4.14) in such a way that the resulting product of factors is still a fair approximation to the matrix A, in the sense of Section 4.1. Increasing simplification should thereby result in reduced resource cost, at the expense of diminishing preconditioner quality.

The application of such a preconditioner of the form

$$Q = LDU \ . \tag{4.15}$$

to a vector (the computation of $\widetilde{v} := Q^{-1}v$) is "cheap" in terms of computation: it involves only two solutions of unit triangular systems and one multiplication of a vector with a diagonal matrix. This amounts to the same number of floating-point operations as would be required for the multiplication of the matrix $(L + D + U)$ with a vector. Since divisions are not as fast as multiplications on many computers, it makes sense to store the diagonal matrix D in inverted form.

Incomplete factorization preconditioners are also well suited for split preconditioning as defined in Equation (4.5). The preconditioner can, for instance, be split into

$$Q_1 = L \qquad \text{and} \qquad Q_2 = DU \ . \tag{4.16}$$

or even

$$Q_1 = L\,\mathrm{sign}(D)\sqrt{|D|} \qquad \text{and} \qquad Q_2 = \sqrt{|D|}\,U \ . \tag{4.17}$$

In the latter case, $\sqrt{|D|}$ should again be stored if the computation of square roots is slower than multiplications.

The remainder of this section describes several choices for this simplification. All these approaches can be characterized as incomplete factorizations. The "classical" incomplete factorization preconditioning [MvdV77] is described in Section 4.4.4. The symbols D_A, L_A, and U_A in the following definitions refer to the diagonal part of A, the strict lower, and the strict upper triangle of A, respectively, yielding the additive splitting $A = L_A + D_A + U_A$.

4.4.1 Jacobi or diagonal preconditioning

The most radical simplification of (4.12) to (4.14) is to approximate A by its diagonal only, substituting the matrices in the right-hand side of Equation (4.11) by $D = D_A$ and $L = U = I$. The preconditioning operation can also be viewed as a single Jacobi iteration (see Section 3.1) with a zero initial vector.

Setting up and storing this preconditioner takes no resources at all (except something insignificant if we want to store the inverse of the diagonal), and the application costs n multiplications.

Although Jacobi preconditioned methods solve a larger percentage of the linear systems in device simulation than unpreconditioned methods, and generally reduce the number of iterations compared to the latter, their performance and success rate is much too low to satisfy the needs.

4.4.2 SSOR preconditioning

The next refinement of incomplete factorization is to ignore the summations in (4.12) to (4.14), substituting $D = D_A$, $L = L_A D_A^{-1} + I$, and $U = I + D_A^{-1} U_A$. The application of this preconditioner can also be viewed as a single iteration of Symmetric Gauss-Seidel (Section 3.1) with a zero initial vector[17].

Applying this preconditioner in the form (4.15) costs as many floating-point operations as a matrix-vector product with the system matrix A. Provided that D_A, L_A, and U_A are stored such that they can be accessed independently (see Chapter 6 for more about matrix data structures), there is no need to compute and store L and U explicitly. Viewing the SSOR preconditioner as

$$Q = (L_A + D_A)\, D_A^{-1}\, (D_A + U_A)\ , \tag{4.18}$$

the application of the preconditioner can be computed by

$$\widetilde{v} = Q^{-1} v = (D_A + U_A)^{-1}\, D_A\, (L_A + D_A)^{-1}\, v\ , \tag{4.19}$$

at only slightly increased cost ($2n$ additional multiplications compared to the form (4.15)).

Distinct savings in computation time can be obtained through a clever combination of the operations for a **preconditioned matrix-vector product**, known as the **Eisenstat trick** [Eis81]. Consider the matrix-vector product for an iterative method with split SSOR preconditioning:

$$\begin{aligned} w &= \widetilde{A} v \\ &= Q_1^{-1} A Q_2^{-1} v \\ &= \left(L_A D_A^{-1} + I\right)^{-1} (L_A + D_A + U_A) \left(I + D_A^{-1} U_A\right)^{-1} D_A^{-1}\, v\ . \end{aligned}$$

---

[17] **Proof.**

$$\begin{aligned} x_0 &= 0 \\ x_{\frac{1}{2}} &= (D_A + L_A)^{-1} (b - U_A x_0) \\ &= (D_A + L_A)^{-1} b \\ x_1 &= (D_A + U_A)^{-1} \left(b - L_A x_{\frac{1}{2}}\right) \\ &= (D_A + U_A)^{-1} \left(b - L_A (D_A + L_A)^{-1} b\right) \\ &= (D_A + U_A)^{-1} \left(b - (D_A + L_A - D_A)(D_A + L_A)^{-1} b\right) \\ &= (D_A + U_A)^{-1} D_A (D_A + L_A)^{-1} b \\ &= \left(I + D_A^{-1} U_A\right)^{-1} D_A^{-1} \left(I + L_A D_A^{-1}\right)^{-1} b \end{aligned}$$

□

Under this form, with $L - I = L_A D_A^{-1}$ and $U - I = D_A^{-1} U_A$ stored explicitly, this whole operation takes $(4m - 2n)$ flops. A clever association of triangular matrices and their inverses[18], however, leads to an algorithm with only $(2m + 4n)$ flops, depicted in Algorithm 4.3. The trick can only partly be applied to left or right preconditioning[19], yielding algorithms with $3m$ flops.

```
$t_1 := (D_A + U_A)^{-1}\ v$
$t_2 := v - D_A t_1$
$t_3 := (L_A + D_A)^{-1}\ t_2$
$w := D_A\ (t_1 + t_3)$
```

Algorithm 4.3: *Split SSOR preconditioned matrix-vector multiplication with the Eisenstat trick.*

SSOR preconditioned methods are again much better than Jacobi preconditioned methods. Since there is no set-up cost, and the number of additional flops for applying this preconditioner is very low due to the Eisenstat trick, SSOR preconditioning is to be preferred over Jacobi preconditioner. A disadvantage lies in the fact that the solution of sparse triangular systems is more difficult to vectorize and parallelize (see Section 6.8), but the lower number of iterations generally outweights the diminution of parallelism.

4.4.3 D-ILU preconditioning

As a next step, the diagonal matrix D is computed using the full formula (4.14), but the summations in (4.12) and (4.13) are still ignored. Of course, this

---

[18] **Proof.**

$$\begin{aligned} w &= \left(L_A D_A^{-1} + I\right)^{-1} (L_A + D_A + U_A) \left(I + D_A^{-1} U_A\right)^{-1} D_A^{-1} v \\ &= D_A\ (L_A + D_A)^{-1}\ ((L_A + D_A) + (D_A + U_A) - D_A)\ (D_A + U_A)^{-1}\ v \\ &= D_A\ (D_A + U_A)^{-1}\ v\ +\ D_A\ (L_A + D_A)^{-1}\ v \\ &\quad - D_A\ (L_A + D_A)^{-1}\ D_A^{-1}\ (D_A + U_A)^{-1}\ v \\ &= D_A\ \left((D_A + U_A)^{-1}\ v - (L_A + D_A)^{-1}\ \left(v - D_A\ (D_A + U_A)^{-1}\ v\right)\right) \qquad \square \end{aligned}$$

[19] For right preconditioning:

$$\begin{aligned} &(L_A + D_A + U_A)\ (D_A + U_A)^{-1}\ D_A\ (L_A + D_A)^{-1}\ v \\ &\qquad = (I + L_A\ (D_A + U_A))\ D_A\ (L_A + D_A)^{-1}\ v \end{aligned}$$

does in no way mean that D is equal to the diagonal matrix of the full LDU-factorization. As with SSOR preconditioning, the triangular matrices $L = \left(L_A D^{-1} + I\right)$ and $U = \left(I + D^{-1} U_A\right)$ do not have to be stored, because the D-ILU preconditioner can be viewed as

$$Q = (L_A + D)\ D^{-1}\ (D + U_A)\ , \tag{4.20}$$

An alternative way to define D-ILU is that Q is the (unique) matrix which can be written in the form (4.20) and satisfies

$$\text{diag}(Q) = D_A\ .$$

The number of flops for the application of the D-ILU preconditioner as well as for preconditioned matrix-vector products are the same as for SSOR preconditioning, since the Eisenstat trick can still be used (see Algorithm 4.4).

```
$t_1 := (D + U_A)^{-1}\ v$
$t_2 := v + (D_A - 2D)\, t_1$
$t_3 := (L_A + D)^{-1}\ t_2$
$w := D\ (t_1 + t_3)$
```

Algorithm 4.4: *Split D-ILU preconditioned matrix-vector multiplication with the Eisenstat trick.*

If the matrix A is structurally symmetric, setting up the D-ILU preconditioner (that is, computing D) takes $\frac{3}{2}(m - n)$ flops.

Note that if the matrix A is tridiagonal (as are the single equation matrices in 1-D device simulation), the D-ILU preconditioner is in fact an exact LDU-factorization of A.

The cost for setting up the D-ILU factorization is negligible, the cost for applying the preconditioner is the same as with SSOR preconditioning, but the number of iterations is generally lower than with the latter. D-ILU is therefore to be preferred over SSOR. Note however that it solves most, but not all the linear systems in semiconductor device simulation [HPWF91]. For really ill-conditioned systems, a stronger preconditioner is needed.

4.4.4 ILU preconditioning

The most common form of incomplete factorization preconditioning involves all three formulae (4.12) to (4.14), but with the additional constraint that entries of L and U whose position in the matrix correspond to zero entries in A are ignored (set to zero):

$$a_{ij} = 0 \Rightarrow l_{ij} = u_{ij} = 0 \, . \tag{4.21}$$

The resulting triangular matrices L and U thus have exactly the same sparsity structure as the corresponding parts L_A and U_A of the original matrix A, and this explains the popularity of such a choice: the part of the data structure that describes the sparsity structure can be reused, thus saving between 20 and 50% of the storage requirements for the factors (see Chapter 6).

The incomplete factorization can be viewed as a matrix splitting

$$A = Q - R \, , \tag{4.22}$$

where all entries of the residual matrix R on positions of nonzero entries of A are zero:

$$a_{ij} \neq 0 \Rightarrow r_{ij} = 0 \, .$$

The summations in (4.12) and (4.13) contain a nonzero term if and only if there is a k such that l_{ik} and u_{kj} are nonzero. With the additional constraint (4.21), nonzero terms in these summations occur only for triplets of distinct indices i, j, and k such that a_{ij}, a_{ik}, and a_{kj} are nonzero. In the undirected graph associated with the sparse matrix, such a triplet is a **triangle**. tensor-product grids, as they are used in finite-difference discretizations (5-point stencils in 2-D and 7-point stencils in 3-D), do not contain any triangles, so that the summations in (4.12) and (4.13) for ILU preconditioning are zero. This completes the proof of the following theorem:

Theorem 4.1 *The ILU and D-ILU preconditioners are identical for single-variable finite-difference discretizations*[20].

---

[20] Many authors confuse these two variants of ILU even for matrices whose graphs include triangles. Since the version of Section 4.4.4 corresponds to the original description in [MvdV77] and in [Elm82], I decided to introduce the name D-ILU in [HPWF91] for the version of Section 4.4.3 to emphasize the distinction.

To save multiplications in the incomplete factorization process, one of the triangular factors is usually chosen not to have a unit diagonal (storing for instance L as defined by (4.12), but (DU) instead of U). The minimum number of flops for this set-up of the ILU preconditioner cannot be determined solely from the size and density of the matrix, as it depends on the number of triangles in the graph of the matrix. If nonzeros are distributed more or less equally over all the rows of the matrix (which is the case for the discretization grids normally used in device simulation), the effort for the factorization in a reasonable data structure is $O(m^2/n)$ (including integer operations)[21].

If the matrix A is symmetric, the triangular factors satisfy $L = U^T$, so that only one of them needs to be stored. For many classes of symmetric positive definite matrices, the diagonal matrix D of the ILU-factorization will be positive (see [GvL83, Man79] for cases where this is not true). Writing $C = (L + I)\sqrt{D}$, the matrix $Q = CC^T$ is called the **incomplete Cholesky factorization** of A. It makes sense to use split preconditioning with $Q_1 = C$ and $Q_2 = C^T$ in this case, since the preconditioned matrix $\widetilde{A} = C^{-1}AC^{-T}$ is still symmetric and positive-definite, so that the Conjugate Gradients method can be applied. This combination is known as **Incomplete Cholesky preconditioned Conjugate Gradients** or **ICCG(0)** [MvdV77].

The number of iterations with ILU preconditioning is usually lower than with D-ILU. The cost is much higher, though. Setting up the ILU factorization takes as many flops as several iterations and is more difficult to parallelize and vectorize (see Section 6.9.2). As the Eisenstat trick cannot be used, a split ILU-preconditioned matrix-vector multiplication takes almost twice as long as with D-ILU. The higher number of fast D-ILU iterations usually takes less total time than the slightly smaller number of slower ILU iterations. Furthermore, the matrices L and U have to be stored for ILU, taking as much memory as the numerical values of A. In consequence, D-ILU is to be preferred over ILU.

ILU does not solve a significant number of those ill-conditioned linear systems that D-ILU cannot solve, so ILU cannot be regarded as a more robust alternative preconditioner, either.

---

[21] **Proof.** (See also Section 6.9.4.3). Assuming the nonzeros are uniformly distributed over the matrix, so that the probability that a_{ij} is nonzero is m/n^2, let the average time to retrieve a nonzero entry in the factorization be s. The average complexity of the inner loop of the factorization is then $O(n \cdot (m/n^2) \cdot s) = O(ms/n)$, that of the middle loop is $O((ms/n) \cdot n \cdot (m/n^2)s) = O(m^2s^2/n^2)$, and that of the outer loop is $O((m^2s^2/n^2) \cdot n) = O(m^2s^2/n)$. Some data structures (e.g., those that store L by columns and U by row, see Chapter 6) achieve $s = O(1)$. □

4.4.5 Positional dropping

Since the exact LDU-factorization generates fill in the factors, we may expect that the quality of an incomplete factorization could be improved by allowing **some limited amount of fill** in the L and U factors. Several strategies to select the fill entries have been proposed. Most of these selection criteria can be characterized as **positional dropping criteria**, as the fill entries are selected by their position in the matrix, in the graph of the matrix, or in the elimination tree of the factorization process.

All these positional dropping criteria define **levels** of fill. The zeroth level, ILU(0), is identical to no-fill ILU as described in Section 4.4.4. ILU(k) includes all the fill entries of the levels 0 through k, thus a superset of the set of fill entries of ILU($k-1$). ILU(n) is a full decomposition. Some of the proposed definitions for the levels are listed below:

a. **Position in the matrix** [MvdV77]. ILU(k) includes all the fill occurring either in the original sparsity pattern of A or in the k co-diagonals on both sides of the main diagonal.

b. **Position in the elimination graph of the matrix** [Saa89]. The level of a fill entry is defined as one plus the minimum sum of fill entries of lower levels that give a nonzero contribution in (4.12) and (4.13).

c. **Position in the elimination tree of the factorization process** [BS87]. An implementation of the bordering factorization technique [Ort88] as a sparse direct solver [BS87, BR90] identifies fill generated by the entry a_{ij} as the set of graph entries obtained by chording the edge of (i, j) along the **elimination tree** [Liu90] of the factorization. ILU(k) is defined by exiting this chording loop in the algorithm after at most k iterations.

In all these variants, the factorization can be split into a symbolic factorization step and a numerical factorization step, just as in full sparse direct solvers. This saves time, and the data structure can be arranged for vectorization and parallelization if many matrices with the same sparsity structure have to be factorized. The co-diagonal version (a) was designed for matrices with a regular or banded structure, and the other versions extend the idea to irregular sparsity structures.

The residual matrix R from Equation (4.22) is by construction zero in those positions where L or U are nonzero:

$$l_{ij} \neq 0 \text{ or } u_{ij} \neq 0 \Rightarrow r_{ij} = 0 \, .$$

The number of iterations for ILU(k)-preconditioned methods usually decreases monotonically with increasing level of fill k. The cost generally increases too strongly to make higher levels of fill attractive. In particular, the storage requirements increase drastically with higher values of k.

4.4.6 Numerical dropping

A dropping criterion determines which entries in the factors can be considered as unimportant for the approximate factorization. Positional dropping criteria ignore the value of an entry when deciding its importance. A **numerical dropping criterion** looks at the magnitude of the value of a nonzero entry to decide whether it has to be dropped or not.

The **drop tolerance** τ controls the accuracy of the approximation, hereafter abbreviated as ND(τ). ND(0) should lead to a full factorization, ND(∞) to one of the incomplete factorizations above. The **numerical dropping criterion** decides what happens with finite and positive values for τ. It decides whether an entry l_{ij} or u_{ij} is important enough with respect to τ.

An ideal numerical dropping criterion would be one like *"an entry is important if dropping it would make the condition number of $\tilde{A}$ larger than $(1 + \tau)$"*. Unfortunately, a global criterion like this is too complicated to implement. A criterion like *"an entry is important if its magnitude is larger than τ"* is much easier to handle. Such a criterion can be called local absolute. A local relative criterion like *"an entry is important if its magnitude is larger than τ times the norm of its row"* can give more control without increased cost.

Standard roundoff analysis [GvL83] is of little help for specifying the details of the dropping criterion. For drop tolerances leading to acceptable storage requirements, τ^{-1} is usually smaller than quantities like n, $|||A|||$, and $\kappa(A)$, which are used in the error bounds.

A **dropping strategy** specifies *when* entries should be dropped. Unimportant entries have to be sieved out during the factorization process, not only at the end of a factorization. One strategy would be to clean each row as soon as it is completed. However, practically each entry that is dropped earlier is missed in later stages of the approximate factorization and thus modifies the values obtained therein.

Efficiency considerations in the design of the approximate factorization algorithm predominate the choice of the numerical dropping strategy, rather than numerical considerations. The details of the algorithm used in the experiments below, along with justifications for the choices, are discussed in Section 6.9.4. For now, note that the algorithm does not vectorize or parallelize, which gives it a major handicap in performance comparisons on fast architectures.

The use of drop tolerances to save storage in sparse direct solvers is not uncommon [DDSvdV91, AJ89]. The idea of using approximate factorizations with numerical dropping to precondition iterative methods has been put forward by several authors [AJ84, Saa88, YMJ^{+}89, GSZ90], but none of these approaches appears to be really efficient on large systems computed in-core.

A smaller drop tolerance τ leads to more fill and thus higher cost in storage and time. The number of iterations are also reduced monotonically. The ratio between cost (in terms of storage) and effect (in terms of total solution time) is generally much better than with positional dropping preconditioners. On sequential computers, numerical dropping often comes close to the efficiency of D-ILU. Because of the low parallelism, however, numerical dropping is slower than no-fill factorization preconditioners on high-performance architectures.

On very ill-conditioned linear systems, sequential ND(τ) preconditioning is often faster than vectorized or parallelized D-ILU or ILU preconditioning. In particular, numerical dropping was found to solve all the systems that D-ILU and ILU are unable to solve [PF91a], and this with moderate storage requirements (usually around three to five times the number of memory words an ILU preconditioner would take). Numerical dropping is a robust preconditioner and should be used as an alternative to D-ILU when the latter fails.

4.5 Nested iterative solvers

According to Equation (4.2), $Q^{-1}v$ should be an approximation to $A^{-1}v$. In other words, Q^{-1} is an approximate operator to the exact operator A^{-1}. Observe that any iterative solver can be seen as an operator Q^{it} that approximates A^{-1} within a certain tolerance, and inside some (Krylov or polynomial) subspace. Why should we not use any iterative solver to precondition an iterative method ?

Such an approach is called a **nested iterative solver**. In the remainder of this section, the iterative solver used as preconditioning operator Q^{it} is referred to as the **inner method**, and the iterative method that is preconditioned by Q^{it} as the **outer method**. Nested iterative solvers using variants of GCR as outer method and GMRES as inner method have been introduced and analyzed under the name GMRESR by van der Vorst and Vuik [vdVV91]. Other variants for the inner method were investigated in [Vui92]. Rutishauser's cgT-method is a nested iterative solver (with CG as outer method and a Chebychev method as inner method) for symmetric positive-definite systems, requiring some limited knowledge about the spectrum [EGRS59]. We examine below under which circumstances other methods can be used to play the outer and the inner roles.

There is a caveat to the nested iterative solver approach. In the theoretical derivations of Chapter 3, A was always assumed to be a nonsingular matrix. Preconditioning (on the right, for now) with an iterative solver, the operator $\widetilde{A} = AQ^{\mathrm{it}}$ qualifies only if Q^{it} is such a nonsingular matrix itself.

If the matrix-vector products occurring in the outer method involve only the system matrix $\widetilde{A}$ itself (and not a splitting of it, like in the methods presented in Section 3.1, nor the transpose $\widetilde{A}^T$, like in CGNR, BiCG, and QMR), all an operator needs to resemble a matrix is **linearity**. Some iterative methods are known to be scaling invariant, but the approximations $Q^{\mathrm{it}}v$ and $Q^{\mathrm{it}}w$ will in general not sum up to $Q^{\mathrm{it}}(v + w)$. The following simple coding rule in the implementation of the outer method makes the inner method look like a linear operator:

> *Avoid all matrix-vector products that can be replaced by linear combinations of already computed vectors.*

Note that performance considerations already guided us to avoid unnecessary matrix-vector products most of the time in Chapter 3. Now we have to avoid them by all means, so that, for instance, an implementation of GCR without using Equation (3.33) is out of question.

With the coding rule above, the result of the i-th matrix-vector product arising in the iteration (which occurs at the $(i-1)$-th iteration in CG and the GMRES-like methods, and at the $\lfloor i/2 \rfloor$-th iteration in the CGS-like methods) is always the first vector contained in $\mathcal{K}_{i+1}(\widetilde{A}, \widetilde{r}_0) - \mathcal{K}_i(\widetilde{A}, \widetilde{r}_0)$ [22] and is used to extend the basis for the sequence of Krylov subspaces. If this vector happens accidentally to be linearly dependent of the previous matrix-vector products, then the outer method appears to be in an invariant subspace of $\widetilde{A}$. The outer method then breaks down and needs to be restarted in the usual way (see Section 5.3).

For the special nested solver combination called GMRESR, with GCR (or Orthomin) as outer method and GMRES as inner method, it is proven in [vdVV91] that breakdown can only occur in the case where the inner GMRES process does not reduce its relative residual. It is also shown that the method cannot break down at all if, in the event of such a stagnating inner iteration, the preconditioner Q^{it} is replaced by A^T (for one outer iteration). Since CG "preconditioned" with A^T is nothing but CGNR, this iteration is just a single iteration of CGNR, or LSQR. The breakdown prevention technique is therefore called **LSQR-switch**. The authors suggest using the LSQR-switch not only if the final relative residual norm of the inner method is exactly one, but also if it is just slightly smaller than one.

For preconditioning by an iterative method, the transposed matrix operations cannot be defined, the preconditioner cannot be split, and $\left(Q^{\text{it}}\right)^{-1}$ cannot be used in line 5 of Algorithm 4.1. Among the iterative methods from Chapter 3, only the right-preconditioned versions of GCR, Orthomin, CGS, Bi-CGSTAB, and BiCGStab2 can therefore be used as outer methods with nested preconditioning.

For most linear systems in device simulation that can be solved with a fast D-ILU preconditioned Bi-CGSTAB solver, nested solvers do not seem to bring any improvement. For very ill-conditioned systems, they may be an alternative to numerical dropping preconditioning, especially on architectures relying on high parallelism (see Section 6.13.4).

---

[22] The initial computation of $\widetilde{r}_0 := \widetilde{b} - \widetilde{A}\widetilde{x}_0$ is also nothing but extending the zero-dimensional $\mathcal{K}_0(\widetilde{A}, \widetilde{r}_0) \equiv \{0\}$ into $\mathcal{K}_1(\widetilde{A}, \widetilde{r}_0) \equiv \operatorname{span}\left\{\widetilde{r}_0\right\}$.

4.6 Other preconditioners

Besides incomplete factorizations and nested iterative solvers, there are of course other ways to approximate A^{-1} that can be used for preconditioning. This section reviews briefly a few other choices.

Incomplete LQ-factorizations have been studied by Saad [Saa88]. Incomplete explicit representations of the inverse matrix A^{-1} itself have been analyzed by Kolotilina, Nikishin, and Yeremin [KNY91].

Polynomial preconditioners approximate the inverse by a low-order polynomial in A. The coefficients are usually obtained through some knowledge about the spectrum of A. These preconditioners usually lead to higher iteration numbers than incomplete factorizations, although the preconditioning operation requires much more computation. In spite of this, they have received considerable attention in the last few years for use on vector-computers and particularly on massively parallel computers, as they rely only on sparse matrix-vector products, which are much easier to parallelize than the solution of sparse triangular systems required in each application of an incomplete factorization preconditioner (see also Section 6.8). Note that using a nested iterative solver as preconditioner is essentially the same idea, but with a polynomial that varies from iteration to iteration.

Sparse matrices in finite-element applications are in fact the sum ("assembly") of a large number of small dense matrices. Each of these small dense matrices describes the discretization inside a single element, and has entries for each pair of corner points of this element. Element-by-element preconditioners construct an approximation for the inverse of the big sparse matrix by combining the exact inverses of the element matrices.

Block preconditioners try to group strongly dependent variables into blocks and construct an approximation of the inverse that uses the inverses (obtained through dense, direct sparse, or iterative solvers) of such strongly coupled blocks. Alternate block factorization preconditioning (ABF) is a block Jacobi preconditioner for applications where several unknowns residing on the same grid point of the discretization are blocked together [BCCS89].

4.7 Comparison

4.7.1 Criteria

The differences between preconditioners can be stated in the following criteria:

1. The storage requirements.
2. The time for setting up the preconditioner (this occurs once for the solution of one linear system).
3. The time for applying the preconditioner, computing $\widetilde{v} := Q^{-1}v$ for a vector v (this occurs once or twice in every iteration).
4. The effect on the condition of the matrix, expressed by the number of iterations to achieve the desired accuracy with a given method.

Criterion 1 is always a hard condition: either the preconditioner fits into the available memory, or it does not[23]. Criterion 4 can be as hard on some ill-conditioned systems: a given preconditioner may not lead to convergence at all, no matter how many iterations are tried. In less stringent cases, Criteria 2, 3, and 4 boil down to one soft question: How long does the solution process take ?

Criterion 4 depends largely on the numerical properties of the linear system in question. For well designed preconditioning methods, the number of iterations usually decreases with increasing storage and CPU cost. There is a trade-off between the number of iterations and the time per iteration.

The optimum answer to this trade-off usually varies with the machine on which the solver is running. The relative weights of Criteria 2, 3, and 4 in the execution time may differ considerably for different machine architectures. See Sections 6.8, 6.9, and 6.13.

---

[23] The criterion is still practically as hard on virtual memory machines. At each application of the preconditioner, every entry of it is referenced exactly once. If the preconditioner fits into virtual memory, but not into physical memory, a page brought into physical memory at one application of the preconditioner will be replaced before it is reused in the next preconditioner application. The program will page fault over the entire memory allocated to the preconditioner. This results in thrashing [SP88]: The program spends most of its time paging, and (wall clock) execution time increases dramatically (by up to a factor of 50).

For different iterative methods, the variation is minor, the general effect of the preconditioner is roughly the same on all the methods considered.

4.7.2 Spectral analysis of preconditioned matrices

Definition. *The scalar λ is an* **eigenvalue** *of the matrix A if and only if there is a nonzero vector v, called* **eigenvector** *of A, such that*

$$Av = \lambda v \ . \tag{4.23}$$

The set of eigenvalues is called the **spectrum** *of A.*

As Krylov subspace methods access the matrix only in matrix-vector products, the spectrum of the system matrix has to play a major role in any theoretical analysis of such iterative methods. Knowledge of the spectrum alone, however, is insufficient to explain the convergence behavior of an iterative method on a linear system. The distribution of eigenvectors in the right-hand side vector (and in the dual initial residual $\hat{r}_0$, for BiCG and variants) determines the sequence of iterates together with the spectrum.

According to Section 4.1, preconditioning should make the matrix "resemble" the identity matrix as much as possible. The only eigenvalue of the identity is 1. A good preconditioner can thus be expected to squeeze the spectrum together around unity.

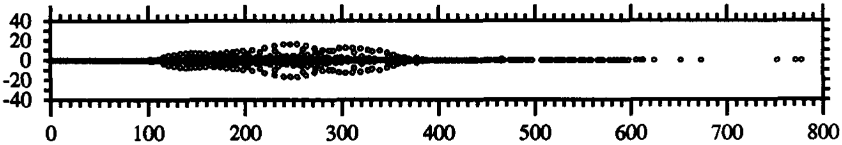


Figure 4.1: *Spectrum of a sample matrix without preconditioning. Please read the warnings about the validity of this plot in the text of Section 4.7.2.*

Figure 4.1 shows the spectrum of a matrix occurring in the plug-in solution of a hole continuity equation in a 2-D simulation on a grid with 2674 points inside the 2-D device simulator GENSIM [Bür90]. The plot represents the distribution of eigenvalues in the complex plane. Figures 4.2 and 4.3 show the spectrum of the preconditioned matrix with different preconditioners (in split position). Note that the scale in Figure 4.1 is different from that in Figures 4.2 and 4.3. Table 4.1 lists the extremal eigenvalues.

precondi-tioner	$\mathrm{Re}(\lambda)$ $= \min$	$\mathrm{Re}(\lambda)$ $= \max$	$\lvert\mathrm{Im}(\lambda)\rvert$ $= \max$	over-head	itera-tions
none	0.029	778	$254 \pm 16.6i$	0%	887
Jacobi	0.0036	1.97	$0.93 \pm 0.14i$	0%	118
SSOR	0.0095	$1.01 \pm 0.0054i$	$0.95 \pm 0.027i$	0%	59
D-ILU	0.013	1.89	$0.98 \pm 0.043i$	15%	57
ILU	0.018	1.90	$0.98 \pm 0.038i$	100%	55
ILU(1)	0.022	1.81	$1 \pm 0.022i$	155%	51
ILU(2)	0.023	1.77	$0.98 \pm 0.021i$	195%	46
ND(0.1)	0.040	1.31	$1 \pm 0.0011i$	106%	19
ND(0.01)	0.22	1.22	$1 \pm 0.00014i$	185%	8

Table 4.1: *Extremal eigenvalues of a matrix with different preconditioners. The second and the third column list the eigenvalues with minimum and maximum real part; the fourth column lists the pair of conjugate eigenvalues with extremal imaginary part; the fifth column lists the relative storage overhead, expressed as the ratio of the amount of floating-point numbers to represent the preconditioner over the number of nonzero entries in the original matrix; and the sixth column lists the number of Bi-CGSTAB iterations to achieve a relative accuracy of* 10^{-10} *on the entire linear system when this preconditioner is used in split position.*

The approach in this section has several serious weaknesses:

- The distribution of eigenvector components in the right-hand side is ignored (see the beginning of this section).
- I do not know how typical the spectral distribution is. The spectrum of matrices in device simulation is always symmetric with respect to the real axis (because the matrices have no imaginary components), and the real part of most eigenvalues is positive (although negative real eigenvalues do occur), but the spectral analysis of larger matrices or of a larger set of samples would take a huge amount of computational resources and was therefore not undertaken.
- The displayed values are, in fact, only ε-pseudo-eigenvalues [Tre], that

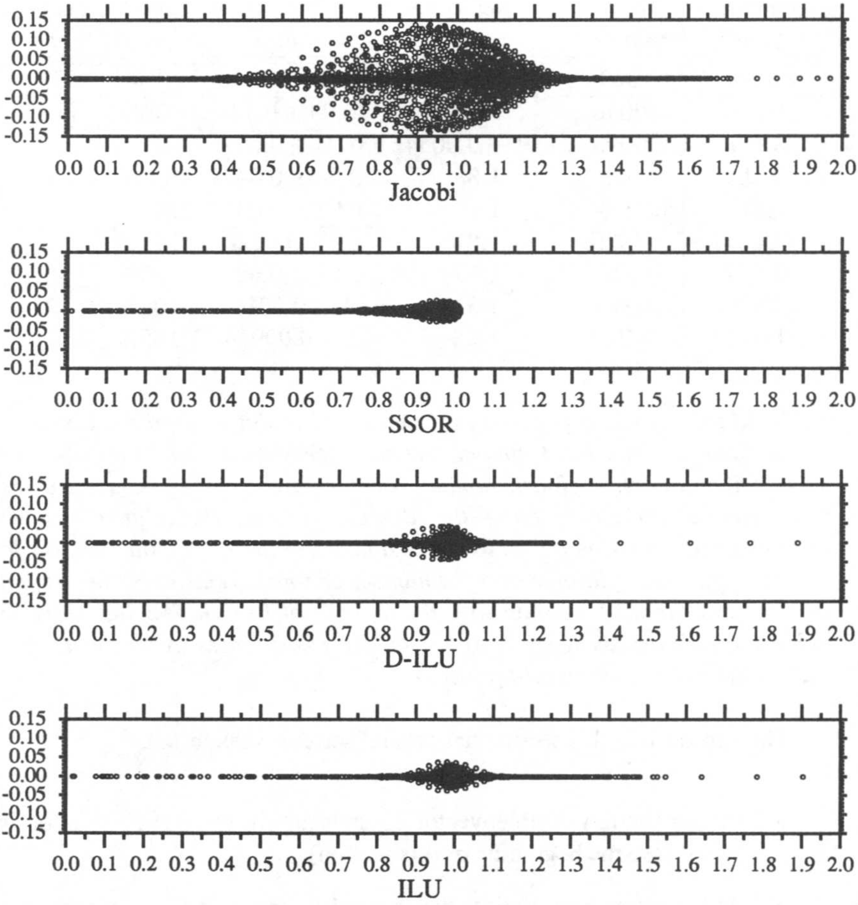


Figure 4.2: *Spectra of the matrix of Figure 4.1 with different preconditioners based on no-fill factorizations.*

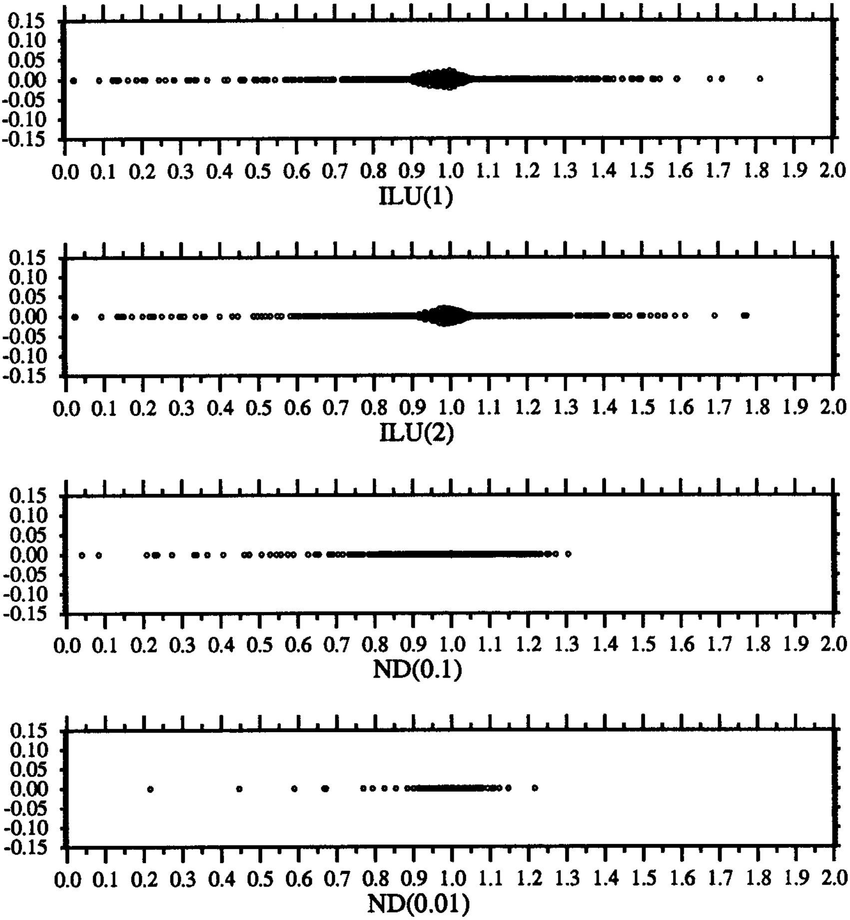


Figure 4.3: *Spectra of the matrix of Figure 4.1 with different preconditioners based on factorizations with parameter-controlled fill, using positional dropping [ILU(k)] or numerical dropping [ND(τ)].*

is, the eigenvalue Equation (4.23) should be relaxed to

$$\|(\lambda I - A)^{-1}\| \geq \epsilon^{-1} .$$

The minimum value of the tolerance ε depends on the floating-point accuracy of the EISPACK [SBD+76] implementation used to determine the eigenvalues. Our problem seems to be at the limits of the numerical stability of these routines, as the Cray Y/MP library implementation failed on some the matrices, but the experiments could be carried out completely on a Convex C-220 (which has one and a half digits more accuracy than Cray machines). Note that iterative eigenvalue solvers rely on the same basic algorithms as iterative linear system solvers and would deliver only a very biased basis for analyzing the latter.

For these reasons, the spectra displayed in Figures 4.1, 4.2, and 4.3 should be regarded only as illustrations, and not be misused for generalizing conclusions. The following sections summarize experience with preconditioners on many linear systems that occurred in the application of several device simulators over almost two years.

4.7.3 Factorizations without fill

For the preconditioners described in Sections 4.4.1 to 4.4.4, Criteria 1, 2, and 3 can be (mostly) estimated by the sparsity structure of system only. Table 4.2 expresses these costs in terms of the number of unknowns n and the number of nonzeros m in the original matrix. An application of the preconditioner is usually performed together with a matrix-vector multiplication, so the time for this combined operation is listed as well.

For most cases, the number of iterations needed with the preconditioners in Table 4.2 decreases from left to right, as expected. Because of the applicability of the Eisenstat trick and the moderate set-up cost, D-ILU generally wins. D-ILU was observed to be the most efficient preconditioner for the large majority of the linear systems occurring in device simulation.

4.7.4 Factorizations with limited fill

The approximate factorizations with parameter-controlled limited fill presented in Sections 4.4.5 and 4.4.6 exhibit the anticipated monotonicity: the more fill

resource cost	*Preconditioner*			
for ...	Jacobi	SSOR	D-ILU	ILU
storage	0	0	n	m
set-up time	0	0	$\frac{3}{2}m$	$O(m^2/n)$
time for $Q^{-1}v$	n	$2m$	$2m$	$2m$
time for $AQ^{-1}v$	$2m$	$3m$	$3m$	$4m$
time for $Q_1^{-1}AQ_2^{-1}v$	$2m$	$2m$	$2m$	$4m$

Table 4.2: *Resource requirements for different preconditioners, for a linear system with* n *unknowns and* m *nonzero entries in the matrix. Storage is expressed by the number of floating-point cells, time by the number of floating-point operations. The storage cost for one vector is* n *and for the original matrix is* m*. One linear operation on vectors costs* n *time units, one unpreconditioned matrix-vector multiplication costs* $2m$*. Minor terms are ignored, and* $m \gg n$ *is assumed.*

allowed (i.e., the more storage is used), the longer the factorization takes, and the fewer iterations are needed.

On not too small and not too ill-conditioned systems where D-ILU preconditioned methods converge in a reasonable number (like $O(\sqrt{n})$) of iterations, these limited-fill preconditioners are not competitive. We therefore concentrate here on ill-conditioned cases were no-fill preconditioners converged very slowly or not at all.

Figure 4.4 plots the solution time in function of the storage requirements for such a resilient case. The linear system occurs in the solution of the hole continuity equation in the simulation of an UV diode on a very large 2-D grid with more than 20k points. All preconditioners presented in Section 4.4 could be run to convergence, but except for numerical dropping, they all needed several thousand Bi-CGSTAB iterations to converge. Jacobi, SSOR, and positional dropping (based on the elimination tree) took too long to fit in the plotting area.

Numerical dropping was the only preconditioner that solved in a reasonable time all the linear systems that occurred in the daily use of three device simulators over the last year. For ill-conditioned systems that the other preconditioners were able to solve, but needed several hundreds or thousands of iterations to converge, numerical dropping was often faster.

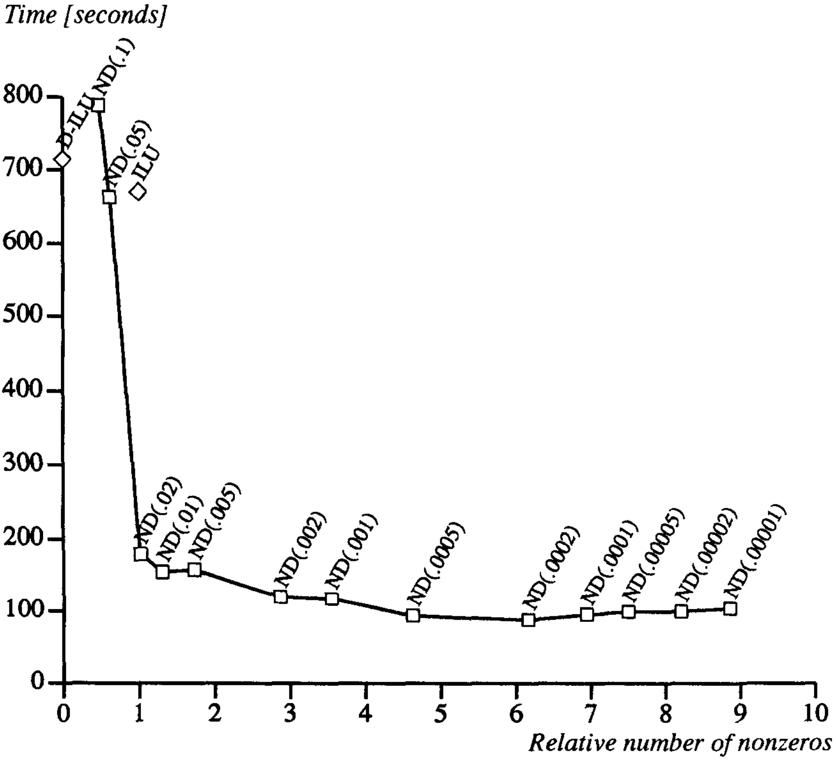


Figure 4.4: *Storage size (number of off-diagonal nonzeros relative to those in A) and total solution time (in seconds on a Convex C-220) for Bi-CGSTAB with different preconditioners on a very ill-conditioned linear system.*

With smaller drop tolerances, the density of the factors increases, and the number of iterations decreases. The increase of the number of nonzeros in the factors implies also an increase of the factorization time and the time per iteration. This is why the total solution time as a function of the number of nonzeros increases again (from ND(0.0002) on in Figure 4.4) after the initial decrease. The drop tolerance τ for optimum time efficiency was generally between 10^{-2} and 10^{-3}. The number of nonzeros in the ND(10^{-2}) preconditioner was usually between one and three times the number m of nonzeros in the original matrix A.

The results of positional dropping based on the elimination tree were disappointing; positional dropping was never more efficient than no-fill preconditioners and did not solve a higher number of ill-conditioned cases. Results with other variants of positional dropping are not much more encouraging [Saa89].

4.7.5 Nested iterative solvers

A particular nested iterative solver is defined by its outer method (and possibly its back-vector parameter ℓ), its inner method (and possibly its parameter), the maximum number of inner iterations, the tolerance on the inner solver, the preconditioner to the inner method and its position, and the stagnation handling strategy. Moreover, this is only the list for one level of nesting; the idea of nested preconditioning can be applied recursively to the inner method.

I did not search exhaustively over this huge parameter space, but I could not find a nested solver combination that would beat the fastest non-nested solver (usually split D-ILU preconditioned Bi-CGSTAB) on a set of moderately ill-conditioned linear systems, although they came close within a factor of two in speed. See also Section 6.13.4.

Since preconditioned matrix-vector products which involve several inner iterations are much more expensive than any linear operations, and since the number of outer iterations is rather low, it makes sense to use an outer method that is economical in matrix operations. GCR and Orthomin proved to be the best choice for the outer method. The convergence speed of the inner iteration usually degraded after a few outer iterations. The erratic convergence behavior of CGS made the latter a bad choice for the inner method.

Nested solvers were sometimes faster than D-ILU preconditioned methods on very ill-conditioned systems. Figure 4.5 shows some experiments with GCR(∞) preconditioned by Bi-CGSTAB with split D-ILU preconditioning on the same linear system and the same machine that was used for Figure 4.4.

For the experiments in Figure 4.5, the tolerance on the inner solver and the maximum number of inner iterations were varied. The different line styles and mark shapes in Figure 4.5 indicate the inner tolerance $\varepsilon_{\text{inner}}$, that is, the factor by which the inner solver should reduce the inner residual norm. The tiny numbers inside the marks in Figure 4.5 indicate the maximum number of inner iterations $i_{\max}$. The preconditioning operator Q^{it} applied to a vector v delivers

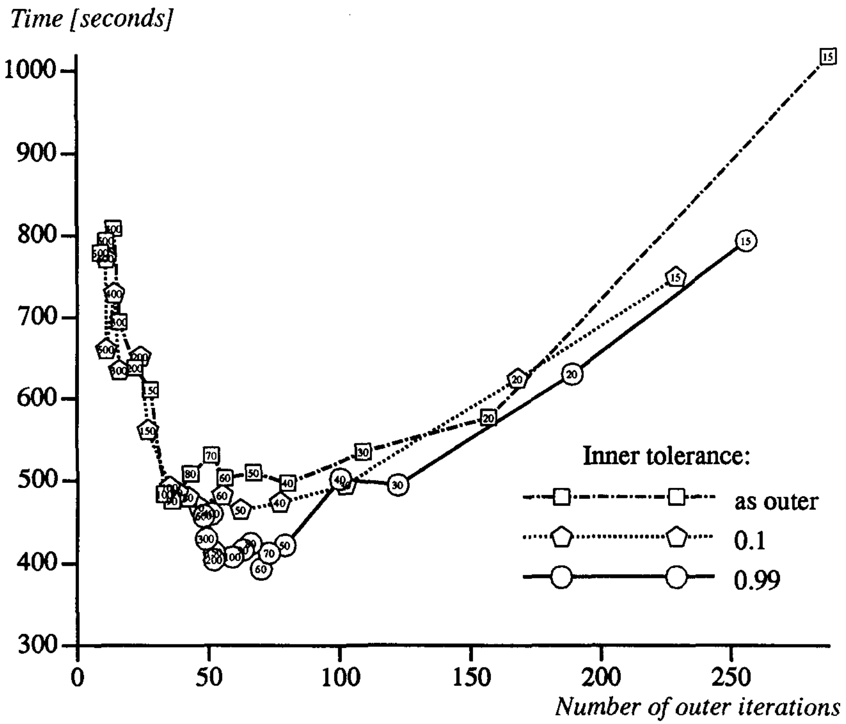


Figure 4.5: *Time (vertical axis, in seconds of a Convex C-220) and number of outer iterations (horizontal axis) for different choices of the inner tolerance (line style) and the maximum number of inner iterations (tiny numbers in the marks) in a nested iterative solver. The outer method was GCR without restarting, the inner method was split D-ILU preconditioned Bi-CGSTAB, and the linear system was the same as in Figure 4.4.*

a preconditioned vector $\widetilde{v}_i = Q^{\text{it}}v$, where the number of inner iterations i is selected such that

$$\frac{\|v - A\widetilde{v}_i\|}{\|v\|} \leq \varepsilon_{\text{inner}} \qquad \text{or} \qquad i = i_{\max} \,.$$

As expected, the number of outer iterations decreases when the number of inner iterations (the tiny numbers inside the marks in Figure 4.5) is increased. The back-vector storage size is proportional to the number of outer iterations in GCR(∞) (note therefore the analogy between Figures 4.5 and 4.4.). The best performance is obtained when the number of inner and outer iterations are roughly the same.

The nested solver converges well even though on many of the outer iterations the inner solver does not reduce, but increase its relative norm. It happens often that no residual in any iteration of the inner Bi-CGSTAB is smaller in norm than the right-hand size. It is thus not surprising that convergence is also obtained if the tolerance $\varepsilon_{\text{inner}}$ of the inner solver is not the same as the outer tolerance (as suggested in [vdVV91]), but only slightly smaller than one (like 0.1 or even more), so that the inner solver reduces its residual norm only by a factor of ten or less. Such a weak inner tolerance usually increases the number of outer iterations, but also improves the overall performance. On our example, this can be seen in Figure 4.5 by comparing the different lines corresponding to different inner tolerances.

Leer - Vide - Empty

5

Convergence Behavior and Control

5.1 Optimality for unsymmetric systems

Unrestarted GMRES constructs in its k-th iteration the optimal solution (the solution minimizing the Euclidean norm of the residual) in the k-dimensional Krylov subspace $\mathcal{K}_k(A, r_0)$. It uses k matrix-vector products, which is the minimum number to construct a basis in $\mathcal{K}_k(A, r_0)$. Theorem 5.1 follows.

Theorem 5.1 *GMRES(∞) uses the minimum number of matrix-vector products to construct a solution that minimizes the residual in* $\mathcal{K}_k(A, r_0)$.

Moreover, the memory consumption and the number of other operations (besides matrix-vector products) is lower for GMRES(∞) than for most other methods having the same minimization property (like GCR(∞)). GMRES is therefore often treated as the optimal iterative method for unsymmetric systems.

One should be very aware of the fact that GMRES(∞) is optimal only in the sense of Theorem 5.1. There are two reasons why GMRES cannot be called the "fastest" or even "best" method in general:

1. The memory requirements increase by one vector in each iteration. After a certain number of iterations, the temporary storage (for the orthonormal

basis vectors) used in the solver algorithm exceeds considerably the storage needed to describe the problem (the matrix, its structure, and the right-hand side).

2. In each iteration the search direction is orthogonalized to all the previous directions, so that each iteration takes more time than the previous one. After a certain number of iterations, linear operations on vectors dominate over matrix-vector products.

It should be clear now that methods with limited resource requirements have to be preferred over GMRES(∞) in some cases, even if they require more matrix-vector products. GMRES(∞) may not be the method of choice if one of the following is true:

- A matrix-vector product takes no more time than a moderate number (a few dozen) of linear operations.
- The number of GMRES(∞) iterations is high (higher than a few dozen).
- Memory requirements are critical (it is inconvenient or impossible to store more than a few dozen vectors).

5.2 Experienced optimality

The imprecision at the end of the previous section was deliberate. There is no hard criterion to predict which method and which preconditioner will be appropriate for a given case. For those methods where theoretical bounds on the convergence speed exist, these bounds involve quantities that are unknown or even more expensive to obtain. Furthermore, designing good preconditioners for systems without any nice properties is more an art than a science, and is somewhat reminiscent of black magic. Until iterative solver design passes beyond the age of alchemy, experience takes the place of knowledge.

Experience with iterative solvers in device simulation [BCD$^+$89, Pin90, HPWF91, PF91a, HSST92] shows that the time for linear operations cannot be neglected completely, iteration numbers are high, and memory is tight, so GMRES(∞) is often not optimal. The loss when restarting or truncating GMRES-like methods is substantial, and a good value for the back vector parameter ℓ is hard to find [VvdV92]. Biconjugate Gradients and variants are generally recognized to be better in device simulation. Bi-CGSTAB has recently emerged as the leading method in this class.

Incomplete factorizations, ILU in particular, are almost the only type of preconditioner used for iterative linear solvers in device simulation. D-ILU appears to give the best compromise between execution time per iteration and number of iterations, for most of the cases. There were some very ill-conditioned cases that could not be solved at all until recently, but they have now been solved using numerical dropping.

5.3 Breakdown control

Two scalar divisions appear in Conjugate Gradients (lines 6 and 8 in Algorithm 3.2 on page 33). Because A is assumed to be positive definite, the denominators $r_{k-1}^T r_{k-1}$ and $p_k^T A p_k$ cannot be zero unless the algorithm has already converged to the exact solution.

As was mentioned in Chapter 3, some of the methods for unsymmetric systems are not safe from zero denominators. Such a situation, where a zero denominator occurs even if the exact solution is not yet found, is called a **breakdown**. The following types of breakdown can be identified in the methods presented:

1. **Coincidence of minima.** The approximation vector that minimizes the residual in $\mathcal{K}_k(A, r_0)$ also minimizes the residual in $\mathcal{K}_{k+1}(A, r_0)$, so that $\alpha_k = 0$ in GCR and Orthomin. The next search direction p_{k+1} is then selected to be the zero vector, so that the computation of α_{k+1} fails.
2. **Invariant subspace.** $\mathcal{K}_k(A^T, \hat{r}_0)$ is A^T-invariant, so that the new dual residual $\hat{r}_k$ in BiCG is a linear combination of all the previous dual residuals, and thus orthogonal to r_k^{BiCG}. The denominator in the computation of the line search parameter α_k is zero, in BiCG as well as in all biorthogonalization methods derived from it.
3. **Orthogonal dual residual.** The residuals of the primal system (1.1) and the dual system (3.45) are orthogonal to each other. The biorthogonalization methods break down at a division by zero in the computation of the biorthogonalization scalar β_{k+1}.

4. **No one-dimensional minimization needed.** The one-dimensional minimum starting from $r_{k+\frac{1}{2}}$ in Bi-CGSTAB and BiCGStab2 coincides with $r_{k+\frac{1}{2}}$, so that the line search parameter ω_k is zero. The degree of the polynomial $\Omega_k(A)$ is only $k-1$ and not k, so that it cannot be used to compute the leading coefficient of the k-th order polynomial Φ_k. The methods break down at the computation of the biorthogonalization scalar β_{k+1}.

5. **No two-dimensional minimization needed.** The two-dimensional minimum starting from $r_{k+\frac{1}{2}}$ in BiCGStab2 coincides with the one-dimensional minimum along $(r_{k+\frac{1}{2}} - r_{k+\frac{1}{3}})$. The coefficient η_k is zero, the polynomial $\Omega_k(A)$ has too low degree, and breakdown occurs at the computation of β_{k+1}.

Breakdown through coincidence of minima (type 1) occurs quite often when GCR or Orthomin are applied to ill-conditioned systems, especially when truncated or restarted algorithms are used (see, for instance, the example in Section 3.6 and Figure 3.3). GMRES, which avoids breakdowns, thus has a true advantage in this respect.

Breakdowns in biorthogonalization methods (types 2 to 5) are far less frequent. In the last one and a half years, the daily use of Bi-CGSTAB (and the occasional use of CGS, BiCG, and BiCGStab2) brought up no breakdown case of type 4 or 5 and only one breakdown of type 2. Breakdown by accidental orthogonality (type 3) does occur from time to time, but certainly not as frequently as type 1 does in variants of GCR.

In exact arithmetic, breakdowns of the types 2 and 3 would occur at the same iteration in all biorthogonalization methods, even if different polynomials are used. If the dual initial residual $\hat{r}_0$ is selected equal to the primal initial residual r_0 (as suggested in Equation (3.60) on page 51), a breakdown of type 2 or 3 may manifest itself already in the one-dimensional minimization step of Bi-CGSTAB and BiCGStab2, leading to $\omega_k = 0$, or in the two-dimensional minimization step of BiCGStab2, leading to singularity for the matrix $(S_k^T S_k)$.

All breakdowns that occurred in practice were fixed by restarting the algorithm, taking as starting iterate the approximation from before the breakdown.

Most of the breakdown conditions can also be circumvented by reformulating the algorithm. GMRES(ℓ) is algebraically equivalent to GCR(ℓ), except that it cannot break down as type 1. Breakdown of type 2 in biorthogonalization methods is **incurable**, the linear system just cannot be solved with this choice of $\hat{r}_0$. Several reformulations of BiCG, CGS, and QMR that avoid breakdown of type 3 have been suggested, based on formal orthogonal polynomials (unnormalized Lanczos algorithms [Gut90b], look-ahead Lanczos [Gut92, FGN90, Jou90] and BiCG [Gut90a] algorithms, QMR with look-ahead [FN91], methods of recursive zoom: MRZ [BZS91] and MRZS [BS91]). The look-ahead approach extends to **near-breakdown**, where denominators are small but nonzero, but the set of tolerance parameters to decide when a quantity is "small" needs more tuning to cope with ill-conditioned systems as those in device simulation. Substituting a small nonzero value for ω_k or η_k would not hurt the Bi-CGSTAB and BiCGStab2 methods, but be sufficient to avoid the breakdown types 4 and 5.

5.4 Convergence criterion

A **convergence criterion** decides whether the accuracy of the current approximation is sufficient, so that the iteration can be terminated. The client application for the iterative solver usually prescribes a **convergence tolerance** on the approximation error. The way this tolerance is interpreted defines the convergence criterion. Variants include what type of norm is to be used, whether a norm should be absolute or relative to another norm, and whether the residual $r_k = b - Ax_k$ or the solution error $\epsilon_k = x^\star - x_k$ should be used.

A tolerance on the solution error ϵ_k specifies how close to the exact solution the approximation x_k should be. A relative criterion is given as the quotient of the error norm over the norm of the solution.

A tolerance on the residual r_k specifies how well the approximation x_k should solve the system. A relative criterion may be defined as the quotient of the final residual over the initial residual norm, or over the norm of the right-hand side. If the initial approximation to the solution is $x_0 = 0$, these definitions are equivalent.

Convergence criteria based on the residual are much more popular than solution error criteria. The reason is that the value of the residual can be computed at any time (through Equation (1.5)), while some norm of the solution error must be estimated.

5.4.1 Using the residual

For most iterative methods, the residual vector is available in every iteration, as it gets updated along with the solution vector. The evaluation of a residual-based convergence then requires calculation of the norm of this vector, which is a vector dot product for Euclidean norms.

GMRES does not keep an up-to-date residual vector, but the Euclidean norm of the residual is directly available through Equation (3.44). For other residual norms, the residual vector has to be constructed at considerable cost through Equation (3.41). Therefore, the convergence criterion should not be evaluated at each iteration, even if this wastes a few iterations before convergence is confirmed at the end. It is advantageous to evaluate the convergence criterion at each restart, where the residual has to be computed anyway.

5.4.2 Using the solution error

Since the exact solution error vector ϵ_k is unknown, we have to find an estimate for it.

Provided that the current approximate x_k is available in each iteration (which is again not the case in GMRES), the latest solution update $(x_k - x_{k-1})$ leads to a cheap estimator (at $5n$ flops) for the relative solution error which is

$$\frac{||x^\star - x_k||}{||x^\star||} \approx \frac{||x_k - x_{k-1}||}{||x_k||}\,. \tag{5.1}$$

As Figure 5.1 shows, this estimate is much too optimistic, especially in phases where the iteration converges slowly (this is independent of the choice of the method). The exact correlation between the convergence on the solution error and the residual is too expensive to evaluate (it depends on how much which eigenvector components of the residual are damped in which iteration). Still, we can use the behavior of the residual norm as a cheap hint (at $2n$ flops) for the convergence speed in the solution error norm. Our improved estimator now samples the approximation to the solution only if the residual norm has changed by more than a given factor α.

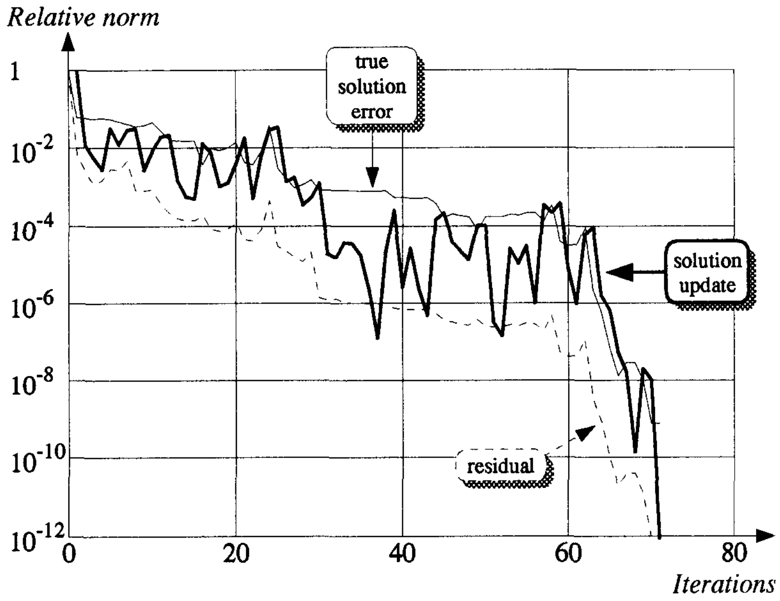


Figure 5.1: *The relative solution update norm* $\frac{\|x_k - x_{k-1}\|}{\|x_k\|}$ *(thick line) used as an estimate for the true solution error* $\frac{\|x^\star - x_k\|}{\|x^\star\|}$ *(thin line). The relative residual norm* $\frac{\|r_k\|}{\|b\|}$ *(dashed line) is given as an indication. Split D-ILU preconditioned Bi-CGSTAB was used to solve the same problem as in Chapter 3 (see also Figure 3.7).*

More precisely, assume the last estimate for the error norm dates from iteration k. The next estimate is then

$$\frac{\|x^\star - x_{k+j}\|}{\|x^\star\|} \approx \frac{\|x_{k+j} - x_k\|}{\|x_{k+j}\|} \tag{5.2}$$

at iteration $(k+j)$, where j is selected such that $\alpha\|r_{k+j}\| \leq \|r_k\|$. To deal with diverging iterations, an estimate is also produced if $\|r_{k+j}\| \geq \alpha\|r_k\|$. Fast zigzagging is flattened through a minimum distance between estimation iterations, $j \geq j_{\min}$, and a maximum $j \leq j_{\max}$ avoids loss of control. An estimation strategy based on Equation (5.2) is thus characterized by the parameter triplet $(\alpha, j_{\min}, j_{\max})$. This class includes our first estimator (5.1) with parameter set $(1, 1, 1)$.

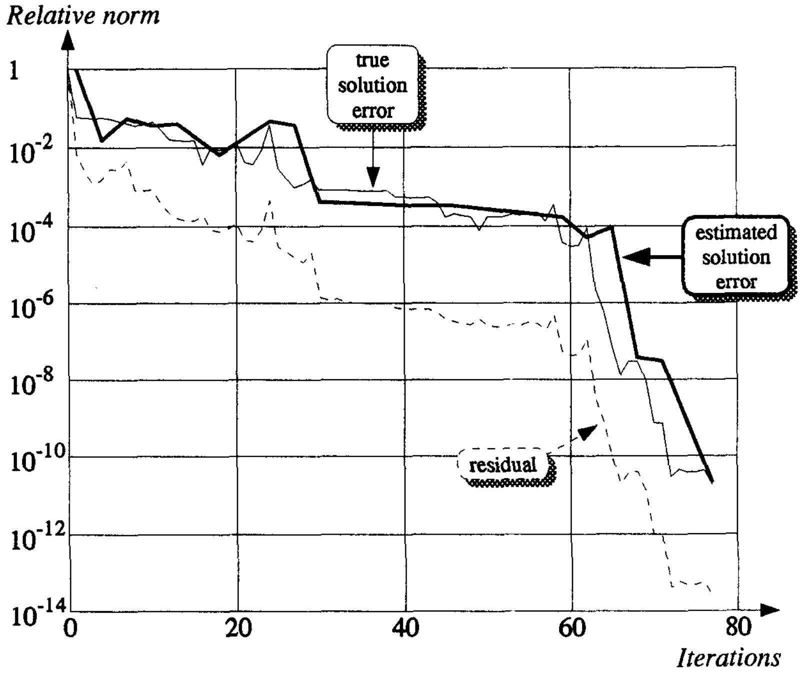


Figure 5.2: *A better estimate (thick line) for the true solution error (thin line). The strategy is guided by the relative residual norm (dashed line), and is characterized by the parameter set* $(\alpha, j_{\min}, j_{\max}) = (2, 3, 20)$ *(see text).*

On a small set of test examples (where the exact solution $x^\star$ was computed with a direct method or with a different preconditioner), as well as in experiments with the GIANT nonlinear solver [Deu90] (see also Section 2.4) inside the device simulator SIMUL [KMFW91], the parameter set $(2, 3, 20)$ appeared to be a good compromise between estimation accuracy and computation overhead. Figure 5.2 shows the quality of this estimator on our standard test example.

5.5 Zigzagging and cancellation

The zigzagging convergence behavior of BiCG and CGS can cause cancellation effects. A needle-shaped peak in the convergence history is nothing else but one or more iterations where large updates are added to the residual vector and the approximation to the solution vector, followed by one or more iterations where the almost opposites of these updates are added. This is the classical case of cancellation: the accuracy of lower-order digits gets lost.

On our standard test example from Chapter 3, we have to select an (unrealistically) tighter convergence tolerance to discern these cancellation effects in Figure 5.3. Knowing that the experiments have been run on a machine with 64-bit IEEE arithmetic and a unit roundoff error of $2 \cdot 10^{-16}$, the BiCG solver has lost the two last digits of accuracy, and the CGS solver has lost four digits. The accuracy of the smoother Bi-CGSTAB solver is of the order of the unit roundoff.

Observe now that the highest peak of the BiCG residual is two orders of magnitude higher than the initial residual (at iteration 24, see Figure 3.5 on page 52), and the highest peak for CGS is four orders higher (also at iteration 24, see Figure 3.6 on page 57).

Cancellation can be avoided by **restarting** the algorithm whenever the residual norm exceeds the right-hand side norm by a small factor (like 10) [vdV89, HPWF91]. The restarted method still yields a well converging process, typically at the same convergence speed as the original algorithm. Figure 5.4 shows how the restarted algorithms (BiCG was restarted at iteration 24, CGS at iteration 7) are not affected by cancellation anymore.

A high peak in the convergence history may also be regarded as a near-breakdown (see Section 5.3). Restarts on large relative residual norms is then just another breakdown avoidance strategy, and the restart condition factor delimits the range of near-breakdowns.

5.6 Rounding error sensibility

Roundoff errors in the computation of vector dot products affect the convergence behavior in Krylov subspace methods, as the scalars ensuring orthogonality are quotients of such dot products. Machine-dependent vari-

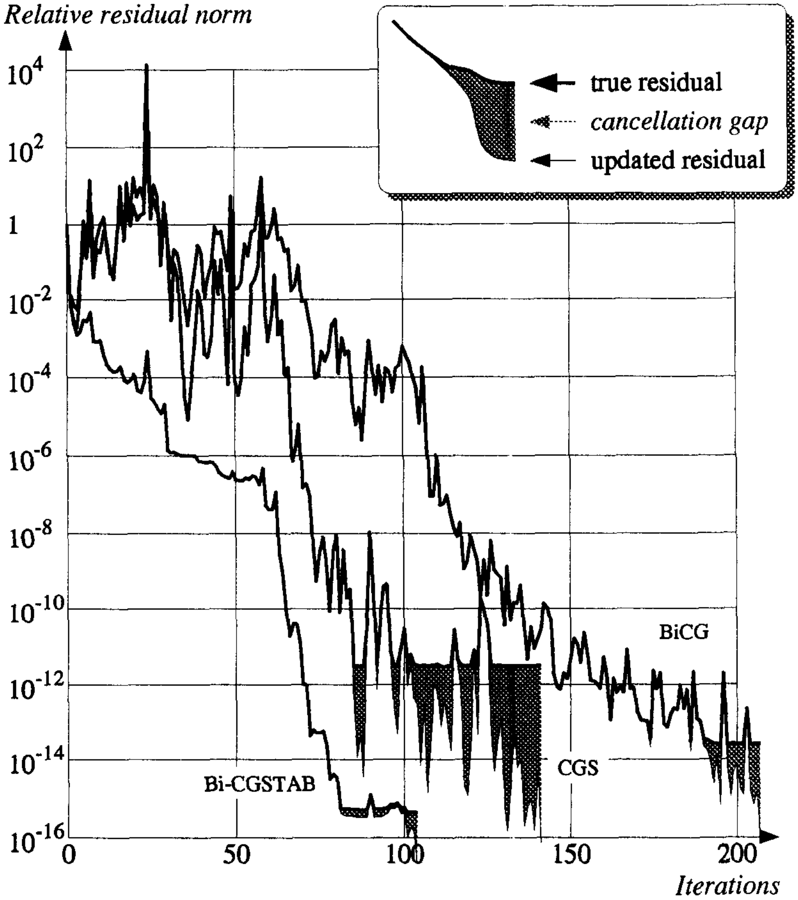


Figure 5.3: *Cancellation effects through zigzagging in BiCG and CGS. For each algorithm, the thin lower line plots the norm of the updated residual vector, and the thick upper line plots the norm of the true residual, computed by plugging the current approximation to the solution into Equation (1.5). The gray area between the two curves shows the cancellation error. The curves for Bi-CGSTAB are given as an illustration.*

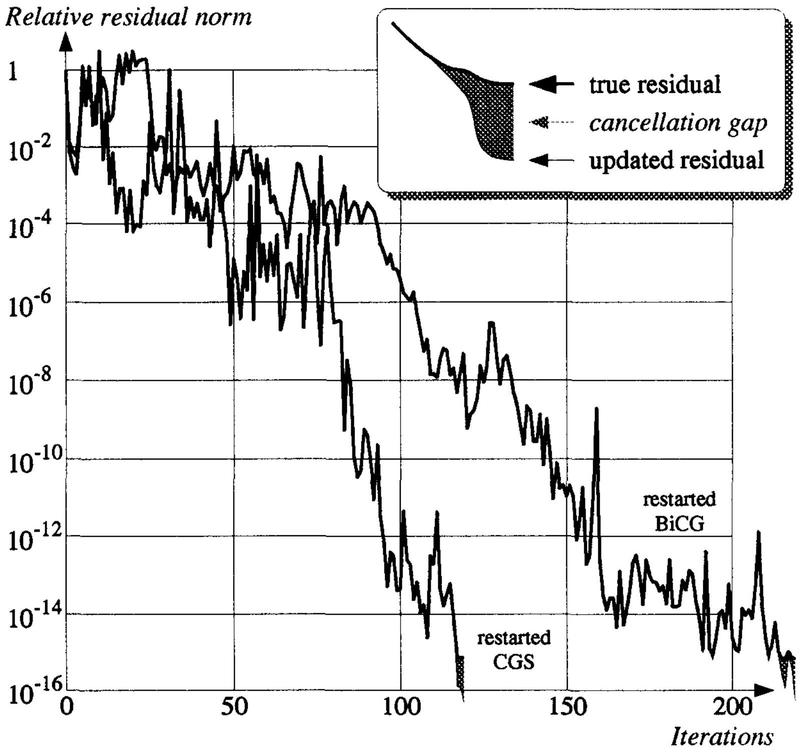


Figure 5.4: *Avoided cancellation by restarting BiCG and CGS.*

ations in the floating-point formats, the arithmetic method, the individual rounding, the format for temporary results, and the evaluation order lead to differences in the roundoff for vector dot products.

As a result, iterative solvers behave differently on different machines (and even in different runs on the same machine, if the evaluation order is not deterministic, see Section 6.5). Usually, the iteration process just walks a slightly different path to the solution. Often, the solver requires one iteration more or less to meet the convergence tolerance. In some critical cases, the differences in the number of iterations is bigger, or the solver converges on one machine and does not on another. Exact breakdown can often only be reconstructed on the same machine, or on a machine with exactly the same arithmetic.

The fact that methods converge anyway, even if they show some sensibility to rounding errors by generating different iterates, is a sign of robustness of the approach. Among the methods in Chapter 3, Bi-CGSTAB and BiCGStab2 are particularly sensitive to roundoff in dot products. Even cases where reduced precision in dot products had a positive effect of Bi-CGSTAB's convergence have been observed [DvdV91].

The residual norm estimate of GMRES, Equation (3.44), is also affected by rounding errors accumulated over the iterations. This is usually not critical as an exact residual norm is computed at each restart. The effect can be seen in the 20-th iteration of GMRES(20) in Figure 3.4. Observed cases where the theoretical upper bound on the residual norm in QMR was smaller than the true residual norm may also be due to accumulated roundoff errors.

5.7 Automatic adaptation

We have seen that their is no single fastest iterative solver for all possible linear systems. For a given linear system, determining the fastest iterative solver can usually only be done a posteriori, or costs more time than this solver would take itself. Generally, the solver will be used that appeared to be the fastest on the majority of systems within the same class. If this solver is not optimal for a few of the many systems to be solved in one application run, but within a small factor (up to an order of magnitude in time) in comparison to the best solver for these particular systems, it does not really matter.

The situation is more complicated if one system cannot be solved at all by the fastest solver, or if the fastest solver happens to be very slow on this system. Using a stabler solver that can cope with this particular system may cost huge amounts of time on the other linear systems. The usual way to deal with this situation is to use the faster solver anyway, but stop it after a maximum number of iterations, even if the desired accuracy has not been achieved. Sometimes, additional outer Newton iterations in the nonlinear solver or even smaller time steps to solve the transient problem can overcome insufficient accuracy in the linear solver. This approach does not always succeed [HPWF91], however, and if it does, it is quite inefficient as well.

A better approach is to switch to the stable solver after the fast solver has exceeded a maximum number of iterations without achieving the required accuracy. Figure 5.5 shows this automatic adaptation on an example. The

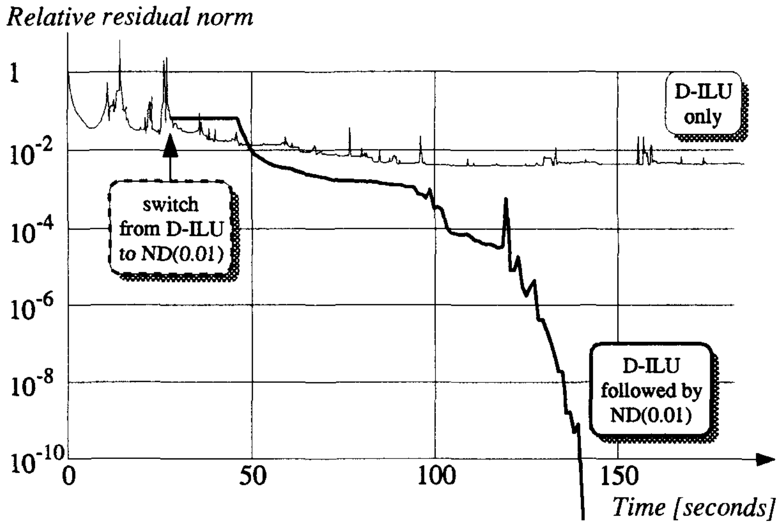


Figure 5.5: *Automatic switching from one iterative solver (fast D-ILU preconditioned Bi-CGSTAB) to another (stable ND(0.01) preconditioned Bi-CGSTAB), as the first solver is incapable of solving the system.*

fast solver, using D-ILU preconditioning only (thin line in Figure 5.5) cannot solve this particular linear system. ND(0.01) preconditioning does solve this system. The solver package switches to stable numerical dropping preconditioning after 150 fast D-ILU preconditioned iterations (arrow in Figure 5.5), using the 150-th approximation of the fast solver as initial guess for the stable solver (thick line in Figure 5.5).

The iteration number at which the switching should take place depends on the problem size and the relative performance of the fast and the stable preconditioners. In practical experiments, it appeared that this number is best chosen to be such that the total time for the D-ILU iterations before the switching is about twice the factorization time for the numerical dropping preconditioner.

Of course, this is only a simple approach to automatic adaptation. An a priori decision on which solver to use would be desirable. See Section 7.3 for possible future realizations.

5.8 Termination control

A **termination control mechanism** evaluates the convergence criterion (see Section 5.4) and terminates the iterative solver when it has converged. Also, the solver is usually terminated if the convergence criterion is not fulfilled after a given maximum number of iterations. The control mechanism should also take care of restarting, to save storage (see Sections 3.5 and 3.7), to avoid breakdown (see Section 5.3), or to avoid cancellation (see Section 5.5).

The mechanism controls the iterative solver in the sense that it makes it compute residual vectors, approximations to the solution, and corresponding norms only when they are needed, thus managing the computational resources efficiently. In an experimental environment, the mechanism should also provide monitoring features.

6

Implementation

It is straightforward to take one of the iterative methods presented in Chapter 3, one of the preconditioners presented in Chapter 4, and one of the termination control mechanisms presented in Chapter 5 and implement them on a computer. In any decent programming language, such an implementation including the preconditioner set-up, the preconditioned method, and the supporting routines for sparse matrix-vector multiplication and solution of sparse triangular systems could be done in three or four pages of code.

However, an efficient implementation which exploits architectural features such as vector operations and parallelism in a portable way and offers the flexibility requirements formulated in Chapter 2 is much more difficult, especially for use with very irregular sparsity structures. The present chapter details some of the non-trivial aspects of such an implementation.

The previous chapters contain already several implementation hints and tricks. In this chapter, we consider the mathematical description of the iterative solver as fixed, and concentrate on executing it in a most efficient way.

The chapter is organized as follows. Section 6.1 explains the symbols used in data structure and memory access pattern illustrations. Section 6.2 analyzes the operations performed in iterative solvers, and Section 6.3 highlights the main architectural features of target computers. Each of the Sections 6.4 to 6.9 details implementation aspects for one of the operations mentioned in

Section 6.2, distinguishing between the machine types listed in Section 6.3 where necessary. Sections 6.10 and 6.11 raise some more general flexibility and portability issues in iterative solver implementations, and Section 6.12 presents PILS, a package of iterative linear solvers based on and used for the findings in this thesis. The final Section 6.13 gives benchmark results, varying separately iterative method, preconditioner, machine, problem size and complexity, and storage requirements, and explains the performance variations by memory-related aspects.

6.1 Figure legends

Sparse matrix **data structures** are one of the most important topics of this chapter. Although all the schemes are described textually, the major data structures are also illustrated in graphical form in Figures 6.2, 6.8, 6.9, 6.10, 6.11, and 6.18. Table 6.1 explains the symbols used in these illustrations. All the illustrations display the storage of the sparse matrix shown in Figure 6.1 (or the lower triangle of it).

$$A = \begin{bmatrix}
a_{00} & a_{01} & 0 & a_{03} & 0 & 0 & a_{06} & 0 & a_{08} \\
a_{10} & a_{11} & 0 & 0 & 0 & a_{15} & 0 & a_{17} & 0 \\
0 & 0 & a_{22} & 0 & a_{24} & 0 & a_{26} & a_{27} & a_{28} \\
a_{30} & 0 & 0 & a_{33} & 0 & 0 & a_{36} & 0 & a_{38} \\
0 & 0 & a_{42} & 0 & a_{44} & a_{45} & 0 & 0 & 0 \\
0 & a_{51} & 0 & 0 & a_{54} & a_{55} & a_{56} & a_{57} & 0 \\
a_{60} & 0 & a_{62} & a_{63} & 0 & a_{65} & a_{66} & 0 & a_{68} \\
0 & a_{71} & a_{72} & 0 & 0 & a_{75} & 0 & a_{77} & a_{78} \\
a_{80} & 0 & a_{82} & a_{83} & 0 & 0 & a_{86} & a_{87} & a_{88}
\end{bmatrix}$$

Figure 6.1: *The sparse matrix whose storage is illustrated in Figures 6.2, 6.8, 6.9, 6.10, 6.11, and 6.18. As the diagonal here is entirely nonzero, it is stored separately and is not considered in the illustrations.*

Memory access patterns on matrices and vectors during certain operations play another important role in this chapter. Seeing which parts are read, written, or updated, or are not touched during a given (outer) iteration of the algorithm, and which parts are new or old, gives an insight why certain parts of an operation can be executed in parallel. Figures 6.12, 6.15, 6.16, and 6.17

▩	A gray square stands for one floating-point value.
●→	A black oval with a thick arrow stands for a pointer (which may also be just an index into one or more arrays).
$\boxed{2_{14}}$	A white square with a big number stands for a (column or row) index, the big number showing the value of this index.
$\boxed{_{7}}$	A white square without a big number stands for a dummy index (referring to an explicit zero fill entry).
→	A thin arrow shows the storage sequence in an array. The small numbers in the right bottom corner of data field symbols mark the position of a field inside an array.
$\boxed{7_{19}}$	Two data field symbols on top of each other stand for entries of two related, synchronous arrays. The example here represents the 19-th entry of an index array and the 19-th entry of a floating-point array. The value of the 19-th entry of the index array is 7.
↑ $\boxed{8_{13}}$	These are three symbols on top of each other, representing the 13-th entry of three synchronous arrays: An index array (with value 8 at this entry), a floating-point array, and a pointer array (the arrow displaying where the 13-th entry points to).

Table 6.1: *Graphical symbols used in the data structure illustrations in Figures 6.2, 6.8, 6.9, 6.10, 6.11, and 6.18.*

illustrate the memory access patterns by drawing certain parts of the matrices and vectors with different shadings. Table 6.2 explains the meaning of these shadings.

□	zero	▦	old, not read
▩	being written or updated		
▨	new, read	▧	new, not read
■	unchanged, read	░	unchanged, not read

Table 6.2: *Shadings for the memory access pattern drawings in Figures 6.12, 6.15, 6.16, and 6.17.*

6.2 Operation breakdown

One of the major reasons for the attractiveness of Krylov subspace methods is that they involve the system matrix A only in the form of an operator for matrix-vector multiplication. In practical preconditioned Krylov subspace methods, however, this is only partly true, as

- Some methods also need A^T for the transposed matrix-vector multiplication.
- The matrix A is used in a non-trivial way to set up the preconditioner.
- The cost for each application of the preconditioner may be higher than the actual matrix-vector product.

A single linear operation (scaling or addition) on vectors of size n takes n flops, and a dot product of two vectors takes $(2n - 1)$ flops. A matrix-vector multiplication with an $n \times n$ sparse matrix having m nonzero entries takes $(2m - n)$ flops. Recall from Table 4.2 on page 93 that the effort to apply certain fast preconditioners is of this order of magnitude.

These flops counts are uneven with respect to memory access. Vector-vector operations access one or more contiguous sections of memory sequentially. Sparse matrix operations (on matrices with irregular sparsity structure) involve at least one indirect access (through a pointer or index) per multiply-add pair. Most features of high-performance architectures favor regular access over indirect addressing.

As we saw in Section 2.6, the average number of nonzeros per row (thus the number $d_{\text{aver}} = m/n$) lies between 7 and 20 for the matrices in device simulation. Let R be the ratio of the number of flops spent in preconditioned matrix-vector products over the number of flops spent in vector-vector operations. This ratio is $R = d_{\text{aver}}/5$ for split D-ILU preconditioned Bi-CGSTAB. The ratios for the other parameter-free methods in Chapter 3 have similar values (see Table 3.1). The ratio is $R = d_{\text{aver}}/(2\ell+5)$ for GMRES(ℓ), and lower for GCR(ℓ) and Orthomin(ℓ).

The ratio R is correlated with the ratio of indirect memory accesses over sequential accesses. With the numbers above, we see that indirect accesses tend to dominate in one category of methods (in particular BiCG and its derivatives), while sequential accesses are (relatively) more frequent in another class (with increasing frequency with the parameter ℓ in the GMRES-like methods). However, none of the two types of memory accesses can be neglected in either class of methods.

As the preconditioner has to be constructed only once at the beginning of a solver, the CPU cost for this operation is less important than that of the repeated operations. If the latter are highly parallelized, however, some attention to preconditioner set-up cost is needed, especially for more complicated preconditioners (e.g. factorizations with some fill).

6.3 Target architectures

This chapter concentrates on efficient implementation for high-performance computers, and distinguishes between three types of architectures: Vector computers, shared-memory multiprocessors, and distributed-memory multicomputers. Even if computer architecture offers a much larger set of variations [HP90], these three classes are sufficient to clarify how parallelism can be exploited in iterative solvers. The features and problems that are important in our context are the following:

- **Vector computers** obtain their computing power from pipelining computations over vectors. A vector operation like $u[0 : n-1] := \alpha v[0 : n-1]$ takes

$$t_{\text{vector}}(n) = t_{\text{start}} + n t_{\text{init}} \qquad (6.1)$$

time [HP90], where the initialization rate t_{init} is much smaller (typically more than one order of magnitude) than the time t_{scalar} for a single ("scalar") multiplication, and the start-up time t_{start} is low enough such that the vector operation is faster than n scalar multiplications:

$$t_{\text{vector}}(n) < nt_{\text{scalar}} . \tag{6.2}$$

Equation 6.1 holds only up to a maximum vector length n_{mvl}, the size of the vector registers. For larger vectors, $n > n_{\text{mvl}}$, the right-hand side of Equation 6.1 has to be replaced by a more complicated expression. With techniques like strip-mining and bottom loading, the performance usually still increases with multiples of n_{mvl}. Chaining of multiple vector pipes makes some combined operations like $u[0 : n-1] := \alpha v[0 : n-1] + w[0 : n-1]$ run at the same initiation rate as a simple vector operation.

- **Shared-memory multiprocessors** consist of a set of parallel processors sharing a common memory. Processors communicate by writing to and reading from the same addresses in this memory and synchronize using semaphores or similar primitives.
- **Distributed-memory multicomputers** (or distributed-memory parallel processors, DMPPs) consist of a set of parallel processors with local memories, and an interconnection network. Processors communicate by sending or receiving explicitly messages or raw data. The network does not provide direct connections between all pairs of processors, but aligns processors in a regular structure (like a ring, a torus, a hypercube, a (fat) tree), in which only nearest neighbors are connected. The time to send a message to a non-neighboring processor then depends on the number of hops in the network. Message transmission time generally increases with message size, but message latency is often significant as well. The ratio of the transmission time for a single short message over the time to perform a single floating-point operation is an important parameter that determines the effectivity of a DMPP on fine grain parallel applications.

Memory bandwidth and latency are crucial factors in all three approaches. Vector computers have to rely on multiple memory banks to feed the vector registers at a sufficient rate. The memory bus is the main bottleneck in shared-memory machines; sophisticated cache designs have to alleviate this problem. Accessing data from another processor's memory is considerably more expensive than local memory access in a DMPP.

Data dependencies and irregularly indexed memory access inhibit most of the performance gains in all the three types of machines. Most of the transformations presented in this chapter attempt to control and reduce data dependencies and to increase the regularity of memory access.

Real machines cannot be assigned unequivocally to one of the classes above. Fortunately, the techniques presented in this chapter can mostly be combined to exploit the features of such hybrid machines. Unfortunately, not all the techniques could be evaluated quantitatively, because of lack of access to certain types of machines, lack of time to implement variants that intuitively appear to be less efficient, or lack of time to squeeze out the last few percents of performance improvement.

6.4 Linear operations

Addition ($u := v + w$) and scaling ($u := \alpha v$) of vectors are the easiest operations to vectorize and parallelize, as they do not involve any data dependencies and have a completely regular structure. Rather than splitting expressions into these basic components, it is important to group more complicated expressions and sequences of expressions together in order to reduce the number of memory accesses and to enable vector chaining. Some apparent vector operations can be replaced by scalar operations after expanding expressions. Common subexpressions should be identified. Table 6.3 illustrates such modifications on an example.

On a DMPP, as long as all vectors are distributed similarly (that is, a processor "owns" the same components of the operand vectors and the result vector), these operations can be executed without any communication. When vectors of length n are distributed over p processors, optimum load balance is achieved if no processor is responsible for more than $\lceil n/p \rceil$ vector components.

6.5 Vector dot products

Vector dot products ($\alpha := v^T w$) can only be parallelized using a different order of evaluation than the usual (sequential) implementation. As floating-point addition is not associative, rounding errors may lead to differing results in the

computation	*R*	*W*	*F*	*modification*
$v_k:=r_k+\beta_k(v_{k-1}-\alpha_{k-1}Aw_{k-1})$ $w_k:=v_k+\beta_k(v_{k-1}-\alpha_{k-1}Aw_{k-1})+\beta_k^2 w_{k-1}$	7	2	10	literally as in Algorithm 3.10
$t:=\beta_k(v_{k-1}-\alpha_{k-1}Aw_{k-1})$ $v_k:=r_k+t$ $w_k:=v_k+t+\beta_k^2 w_{k-1}$	7	3	7	common subexpression elimination
$t_1:=\alpha_{k-1}Aw_{k-1}$ $t_2:=v_{k-1}-t_1$ $t_3:=\beta_k t_2$ $v_k:=r_k+t_2$ $t_4:=v_k+t_2$ $t_5:=\beta_k^2 w_{k-1}$ $w_k:=t_4+t_5$	11	7	7	splitting into addition and scaling
$w_k:=\ (t:=\beta_k(v_{k-1}-\alpha_{k-1}Aw_{k-1}))$ $+(v_k:=r_k+t)$ $+\beta_k^2 w_{k-1}$	4	2	7	grouping and chaining (note: t is not copied to memory)

Table 6.3: *Saving memory accesses in code lines 9 and 10 of Algorithm 3.10 (CGS) by grouping linear operations. The columns labeled "R", "W", and "F" list the number of memory reads, writes, and flops per vector entry.*

parallel and in the sequential implementation. Krylov subspace methods use vector dot products to construct orthogonalities and are usually very sensitive to such minor differences. These minor variations accumulate from one iteration to another, so that an iterative solver based on parallel dot products walks through a different set of iterates than one based on sequential dot products. This frequently results in a difference in the number of iterations required for convergence between a sequential and a parallel implementation, even if the underlying hardware supports exactly the same floating-point representation and (individual) rounding. See also Section 5.6.

On multitasking multiprocessors (e.g. Convex and Cray machines), the parallelizing compiler often implements reduction operators asynchronously in order to improve load balance. The vectors are partitioned into p or more chunks. Whenever a processor is available, it grabs a chunk and computes a partial dot product on this chunk. When it has finished, it locks a semaphore, adds its partial result to a global summation variable, and unlocks the semaphore. This approach leads to differences in the summation order that depend undeterministically on the current load of the machine. Ultimately, this means that two runs for exactly the same input data on exactly the same parallel machine can give different results. I experimented with one case where the parallel implementation of split D-ILU preconditioned Bi-CGSTAB on a Convex C-220 converged in 75 percent of the runs, and diverged or broke down in the rest of the runs.

On a DMPP, every processor first computes a partial dot product of its local vector components. If n vector components are equally distributed over p processors (as in Section 6.4), every processor has to perform $2n/p$ flops. For any network, at least $\log_2 p$ stages of parallel data exchanges are then needed to form the global sum. Communication dominates performance, and we have a **high-latency network** in this context, if sending a single (short) message takes more than $2n/(p \log_2 p)$ flops. Note that the larger the number of processors, the lower the message latency should be to keep communication and computation balanced in this operation.

This global synchronization induced by vector dot products become a serious bottleneck [AÖ87, MvdV87] on massively parallel machines ($p \approx n$) or other DMPPs with high message latency. Variations of iterative methods that delay dot products over several iterations have been suggested, but they seem to suffer from serious stability degradation [AÖES88, CG89]. As long as the partial iterations between such synchronization points are well balanced, vector dot products do not induce any problems on DMPPs with systolic raw-data communication [APR89, PAF92].

6.6 Sparse matrix-vector multiplication

The classical way to store matrices with an irregular sparsity structure uses **compressed rows**: All the d_i nonzero entries of row i are stored in a dense vector. The corresponding column indices are listed in a similar dense integer vector. All the compressed rows of the matrix are assembled in one long vector (of length m), and their column indices come along in a similar (synchronous) integer vector. A row start index vector points to the start of each row in the two other arrays. The row start vector has length $n + 1$, the last entry being a sentinel. Figure 6.2 illustrates the data structure[24].

---

[24] In FORTRAN implementations the row start pointer vector is usually named `IA`, the column index vector `JA`, and the vector with the nonzero entries `A`. Because of this naming, the compressed row data structure is also often referred to by **(IA,JA,A)-form**.

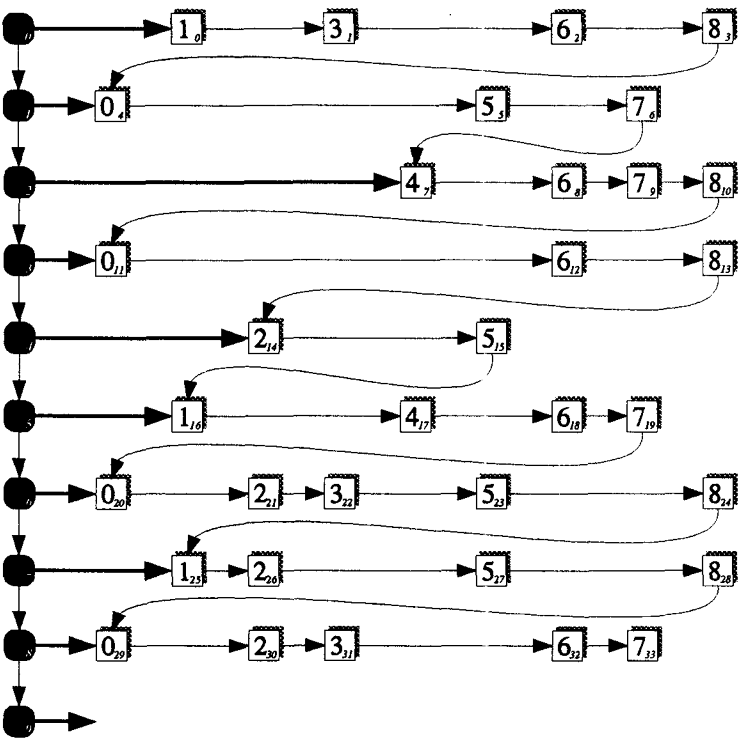


Figure 6.2: *Compressed rows to store the sparse matrix of Figure 6.1.*

Several variations of compressed row storage are in use, like compressed column storage. If the diagonal is entirely nonzero (which is generally the case for PDEs), it should be stored as a dense vector, separate from the nonzero structure of the off-diagonals. Structural symmetry can be exploited to halve the integer overhead by storing the upper triangular part by rows and the lower triangular part by columns [BCD$^+$89]. The latter variant also reduces memory references in sparse matrix-vector multiplication, using one compressed column index vector to multiply one row of the upper triangular part and one column of the lower triangular part at the same time.

For the purpose of extracting parallelism for sparse matrix-vector multiplication, row-oriented storage schemes are generally to be preferred over their column-oriented counterparts, as the latter require the result vector to be accessed indirectly. Each vector access through column indices (in a row-oriented scheme) requires one **gather** operation to read from the argument vector. Each vector access through row indices (in a column-oriented scheme) requires one **gather** operation to read the previous values from the result vector and one **scatter** operation to write the updated values to the result vector.

Other common data structures for sparse matrices are reviewed in [Saa90].

Any kind of structural properties of a sparse matrix can and should be exploited to improve the performance of matrix-vector multiplication in iterative solvers.

6.6.1 Shared-memory multiprocessors

On compressed rows, shared-memory parallelization is trivially done by partitioning the components of the result vector among the processors, ensuring that each processor gets the same amount of work to do. For a sufficiently uniform distribution of nonzeros among rows (which is the case for PDE discretizations), this load balance is obtained by assigning every processor at most $\lceil n/p \rceil$ components.

Although this partitioning minimizes the synchronization overhead on shared-memory machines, the data structures presented below for vector processors are considerably efficient as well.

6.6.2 Distributed-memory multicomputers

On a DMPP, some communication between processors is needed. A row-wise partitioning of the matrix, storing in each local memory those rows that correspond to the local vector components (as selected in Section 6.4), is appropriate to separate communication from computation and to achieve maximal grain size in the communication. Before starting the actual computation, every processor has to fetch from other processors the non-local components its rows refer to. In terms of graphs, the computation for one vertex requires data from all its adjacent vertices. In addition to balancing the computational load (as above), the following points have to be addressed:

1. Adjacent vertices should, if possible, be assigned to the same processor, to minimize the communication volume.
2. Adjacent vertices not assigned to the same processor should be assigned to processors that are close to each other in the interconnection network, as communication time increases with the number of hops[25].
3. If two or more adjacent vertices of a vertex v cannot be assigned to v's processor, they should be assigned to the same other processor, so that v's value has to be transmitted only once.
4. Communication should be balanced, so that every processor has approximately the same number of messages or data items to send.
5. On DMPPs with efficient manual message forwarding (like iWarp [BCC$^+$88] and some transputer-based machines), if vertex v_1 on processor P_1 is adjacent to v_2 on P_2 and to v_3 on P_3, and if P_2 lies on a shortest path from P_1 to P_3, then P_1 should send the value of v_1 only to P_2, which should forward it to P_3.
6. On DMPPs with large message latencies (like iPSC, nCUBE, most transputer machines, PARAGON, and CM-5), all the values to be sent from one processor to another should be packed together, and every processor should only talk to a few other processors.
7. On SIMD architectures (like CM-2 or MasPar), every processor should have approximately the same number of rows with a given number of nonzeros, to reduce idle looping.

Points 1, 2, 3, 5, and 6 are achieved by **mapping** the locality of the problem graph into the locality of the interconnection topology. While minimizing the communication overhead, such a mapping still needs to ensure equal load balance as specified above.

This is easy to accomplish for regular (e.g., finite difference) graphs, but finding an optimal mapping of a complex structure—like our finite element graphs—is NP-complete. For problem sizes of several tens or hundreds of thousands, even low order complexities (like $O(n^2)$) are too high. In [PAF92],

---

[25] Some machines appear to offer a homogeneous network, where the communication between any pair of processors is almost equally fast. Still, many simultaneous communications over long distances can cause network contention.

we have suggested various mapping heuristics with a "quasi" linear complexity (less than $O(n^{1+\varepsilon})$ for any $\varepsilon > 0$).

There are two basic ways of identifying locality in the problem graph: **topology** and **geometry**. *Minimum path lengths define a topological distance* concept in the associated (undirected) graph. The distance to a selected set of vertices S defines a single coordinate for every vertex. A vertex partioning based on topological distance to a single vertex or a vertex set is usually called **level structure** (see Figure 6.7)[26]. The pair of distances to two distinct sets S and T form a two-dimensional coordinate system on the graph, or a **double level structure** [PAF92].

For sparse matrices resulting from the discretization of PDEs, the physical coordinates of the grid points give a **geometric** source of information to detect locality. Geometric distance between grid points does not directly tell whether the corresponding vertices are connected by an edge, but close points are more likely to be connected than far distant points. A two-dimensional coordinate system based on geometric distances can consist of a pair of two geometric coordinates of a point, or any projection onto two distinct directions.

A good mapping for a one-, two-, or three-dimensional DMPP interconnection network can now be obtained by sorting vertices according to their topological or geometric coordinates. On networks with higher dimensionality or large message latencies (these two properties are usually correlated [Vit88, APR89, Dal90]), it is preferable to use a one- or two-dimensional target topology in order to keep the number of individual messages low (optimizing points 3 and 6).

Both types of mappings can be improved by selecting carefully the starting sets (in the case of topological heuristics) or the projection axes (in the case of geometric heuristics).

To illustrate the effects of these mapping strategies on irregularly refined discretization grids, we will now consider a small example. The two-dimensional grid in Figure 6.3 should be mapped to sixteen processors, connected in a four-by-four mesh as shown in Figure 6.4.

Figure 6.5 shows the result of geometric mapping heuristic with a 2-D target topology, and Figure 6.6 that of a topological mapping heuristic with a 2-D

---

[26] Many sparse matrix reordering algorithms for reducing the fill in a full factorization (like nested dissection or reverse Cuthill-McKee [GL81]) and detecting [Saa89] or constructing [MR88, PAF92] parallelism in preconditioners (see also Section 6.8) use level structures.

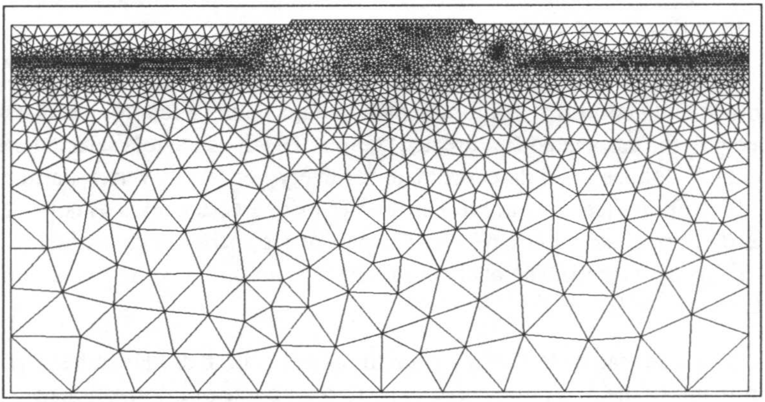


Figure 6.3: *The 2-D triangular grid used to analyze topological and geometric mapping heuristics in Figures 6.5, 6.6, and 6.7 and coloring heuristics in Figures 6.13 and 6.14. The grid consist of 2674 points.*

target topology. Observe in particular the strongly refined region in the upper fifth of the grid. In this region, there are much more points along the horizontal direction than there are along the vertical direction. Geometric mapping adapts to this by assigning long narrow areas to processors. Topological mapping creates a larger number of almost "round" areas in this region. The topological heuristic appears to reduce communication volume by assigning to processors areas with a better aspect ratio than those generated by the geometric heuristic. On the other hand, topological mapping often assigns unconnected areas (f.i., the assignment to processor #11 in Figure 6.6 is split in three pieces). Only a quantitative evaluation can decide which mapping is better.

These mappings were done with two of the heuristics suggested in [PAF92]. All heuristics were evaluated for K2, a planned 64-processor DMPP with low-latency and high-bandwidth communication in an 8 × 8 torus raw-data interconnection network [AFNV90, AzBF$^+$90], which was simulated using the K9 simulator [BPA89]. An extract of the results of this study is given in Section 6.13.3. The general conclusion was that geometric mapping heuristic give better results than topological heuristics.

A typical characteristic of level structures on irregularly refined grids can be seen in Figure 6.7. No matter where the levelization is started, levels tend

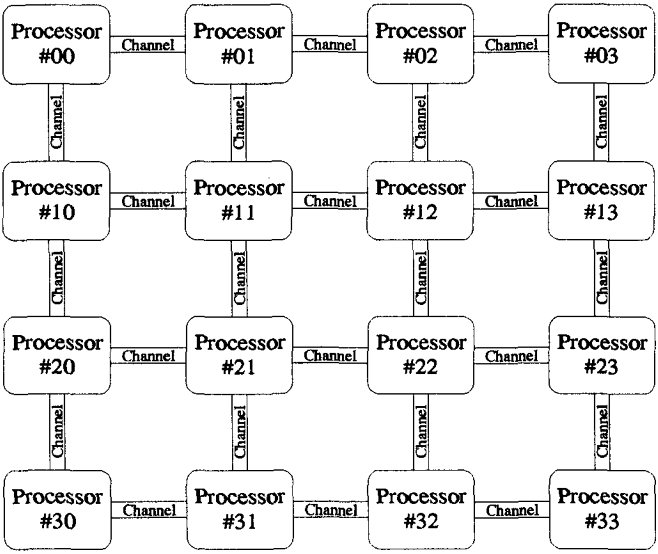


Figure 6.4: *A 16-processor DMPP with a* 4 × 4 *mesh interconnection topology. Figures 6.5 and 6.6 show how the grid of Figure 6.3 is mapped on this mesh of processors.*

to become long in the direction of refinement.

The best of the mappings from [PAF92], 2-D geometric mapping, works as follows on a $p_v \times p_h$ torus or mesh of processors (possibly embedded in a hypercube through Gray coding): First, vertices are sorted according to their x-coordinate in the physical discretization grid. This sorted list of vertices is partitioned by $(p_v - 1)$ vertical cut lines into p_v sets of n/p_v vertices each, and each of these sets is assigned to one column of the mesh of processors. Then each set is sorted again according to the y-coordinate and partitioned by $(p_h - 1)$ horizontal cut lines into p_h sets, and each of these sets is assigned to one processor of this column. The time complexity of this algorithm is $O(n \log n)$ [PAF92].

The size of the messages exchanged in sparse matrix-vector multiplication is low, especially in 3-D applications. Consider for instance a tensor-product grid of size $n^{1/3} \times n^{1/3} \times n^{1/3}$ mapped to a three-dimensional $p^{1/3} \times p^{1/3} \times p^{1/3}$ mesh of processors. This interconnection structure minimizes

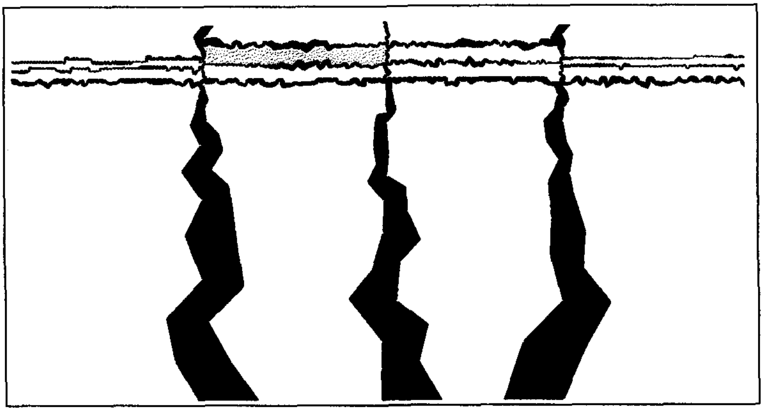


Figure 6.5: *Two-dimensional geometric mapping of the grid in Figure 6.3 to the* 4×4 *processor interconnection network shown in Figure 6.4. All elements (triangles) whose corners are assigned to different processor are drawn in black. White lines crossing black strips correspond to graph edges that require communication. As an example, the gray area shows the part of the grid assigned to processor #11.*

the communication volume for this grid, and with a perfect mapping, every inner processor will have to exchange six messages of size $(n/p)^{1/3}$ with its direct neighbors. Even with a huge grid size $n = 1,000,000$ and a moderate number of processors $p = 1,000$, messages will consist of only 100 numbers. Larger message sizes are obtained with more than one unknown per grid point and lower dimensionality interconnections, but at the expense of increased communication volume. As long as $2m/p$, the number of flops per processor if the workload of this operation is perfectly balanced, is much higher than the time to send all these small messages, the DMPP will still be quite effective. On massively parallel machines (where $p \approx n$), communication will dominate the operation.

6.6.3 Vector computers

Sparse matrices resulting from regular discretizations (e.g., finite differences, see Section 2.5) are the easiest case to handle on vector (and parallel)

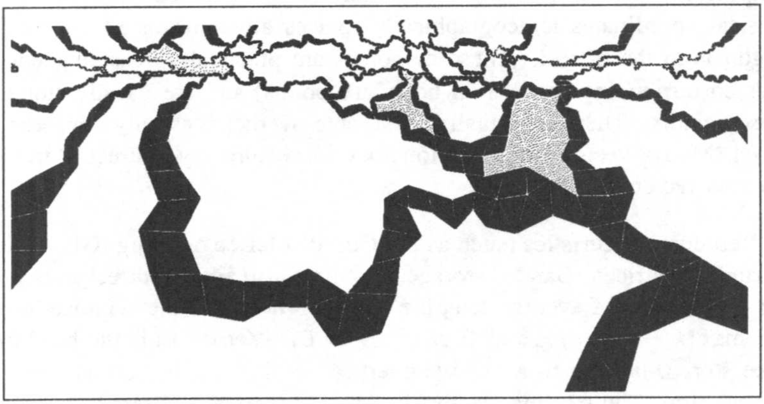


Figure 6.6: *Two-dimensional topological mapping of the grid in Figure 6.3 to the* 4×4 *processor interconnection network shown in Figure 6.4. All elements (triangles) whose corners are assigned to different processor are drawn in black. White lines crossing black strips correspond to graph edges that require communication. As an example, the gray area shows the part of the grid assigned to processor #11.*

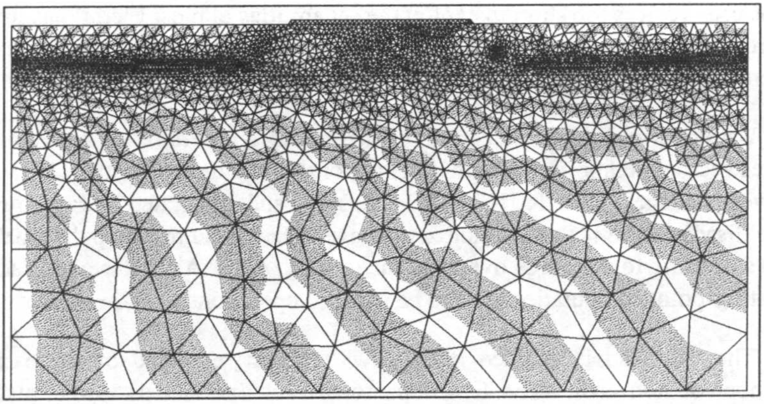


Figure 6.7: *A level structure on the grid of Figure 6.3. Gray areas correspond to odd numbered levels, white areas to even numbered levels. The starting set (level zero) consists of the points on the left boundary of the grid.*

computers. Natural ordering of the unknowns (obtained by sorting their physical coordinates lexicographically) places all nonzeros in only a few diagonals in the matrix. These diagonals are practically dense (except for rows corresponding to external boundary points) and are usually stored in dense vectors. There is virtually no storage overhead, as only nonzeros are stored. Matrix-vector multiplication does not require any indirect addressing and uses vectors of size n.

Reordering heuristics (such as the Cuthill-McKee ordering [GL81]) lead to banded matrices. Banded storage structures also avoid indirect addressing and use vectors of average length $n - b/2$, where b is the semibandwidth: $b = \max\{k \mid \exists i : \ a_{i,i+k} \neq 0 \text{ or } a_{i,i-k} \neq 0\}$. Zeros inside the band have to be stored, leading to a relative overhead of $2b/d_{\text{aver}}$ in both storage and computation. The minimal bandwidth is $O(\sqrt{n})$ for 2-D discretizations, and $O(n^{2/3})$ for 3-D, which is clearly much larger than d_{aver} (see Section 2.5). Envelope reorderings (obtained e.g. by Reverse Cuthill-McKee) are exploited in skyline formats. Their storage requirements are less than those of banded forms, but asymptotically differ only by a constant, and the obtained vectors are much shorter.

For sparsity structures that are too irregular for the tricks above, compressed rows alone are still a bad choice. Because the number of nonzero entries per row is so low (see Sections 2.5 and 2.6), vectorizing only over one row at a time uses vectors too short to exploit the capabilities of most vector machines. Hardware modifications that would avoid the vector unit startup latency have been suggested, but not realized [TRM91].

Using a two-dimensional array (of size $n \times d_{\text{max}}$) for the compressed rows gives vector operations of length n. This storage scheme is used in the iterative solver package ITPACK [KRYG82, KOY89] and illustrated in Figure 6.8. The disadvantage is that explicit zeros have to be stored to fill up the smaller rows, leading to a storage overhead factor $d_{\text{max}}/d_{\text{aver}}$. A similar overhead in floating-point computations has to be paid for operations with the zeros.

The overhead is avoided if the rows are first reordered by decreasing number of nonzeros. The first nonzero of each (permuted) row is placed in a dense array, and there is a similar, but usually shorter dense array for the second nonzero of each row, and so on. The average vector length for these so-called **jagged diagonals** [Saa89, Saa90] is $(d_{\text{aver}}/d_{\text{max}})\, n$. Of course, the reordering of the rows should not involve accessing the argument vector indirectly, indexed through the corresponding permutation. The vector should

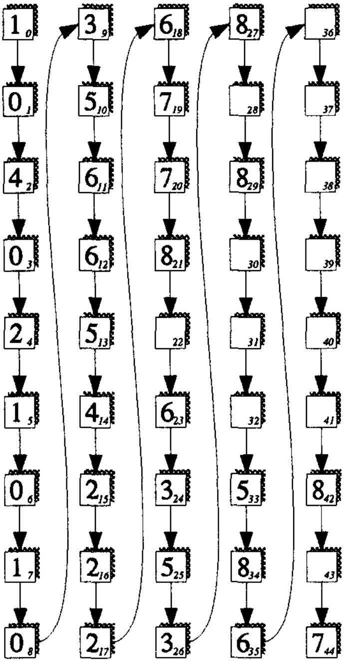


Figure 6.8: *ITPACK scheme to store the sparse matrix of Figure 6.1.*

first be copied into a permuted vector, which is then accessed directly inside the multiplication operation. Even better, all the vectors should be held in permuted form during the entire solver algorithm. Figure 6.9 illustrates the data structure.

Algorithm 6.1 shows the code for matrix-vector multiplication with this data structure. The loops on code lines 1 and 5 are parallel. The loop on line 5 can be vectorized efficiently only on a machine with support for gather operations. For each two flops (one multiplication and one addition), line 6 needs five memory accesses: one integer index and three floating-point values are read, and one floating-point number is written.

Caches are of little help for improving the memory performance in this case. For large problem size n, all cached values will be overwritten before they are reused, except for a few occasional hits on v (whose frequency

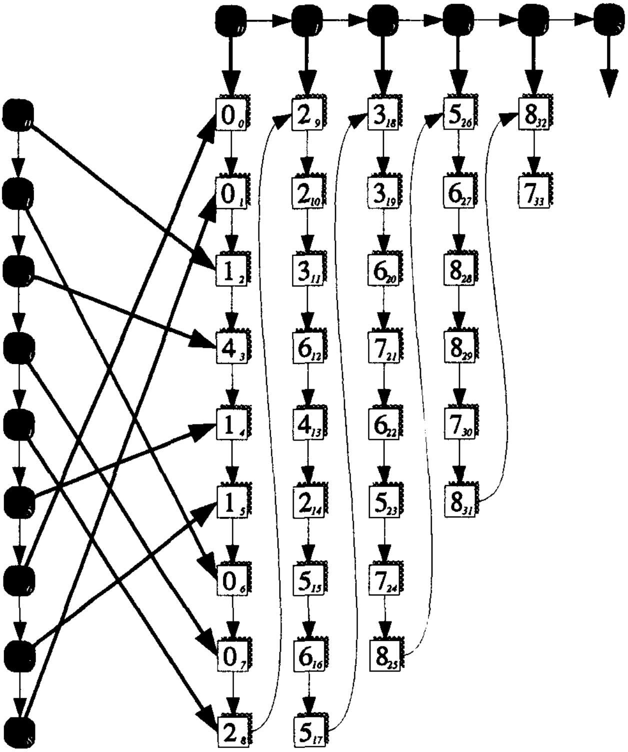


Figure 6.9: *Jagged diagonals to store the sparse matrix of Figure 6.1.*

depends again on the matrix bandwidth).

By exchanging the nesting of the loops on lines 4 and 5 in Algorithm 6.1 and vectorizing over the outer loop, most of the read and write accesses on w are avoided. While this is simple for the ITPACK data structure, the varying vector lengths in jagged diagonals do not allow this trick. Another modification allows the savings without storage overhead: Each set of n_d rows with an identical number of nonzeros d are packed in a compact two-dimensional array of size $n_d \times d$. This reduces the average vector length to $n/d_{\max}$, which limits the performance improvement.

```
foreach $i$ in $[0 : n - 1]$ do
  $w[i] := \texttt{JD.d}[i]v[i]$
end foreach
foreach $k$ in $[0 : \texttt{JD.b} - 1]$ do
  foreach $i$ in $[0 : \texttt{JD.c}[k] - 1]$ do
    $w[i] := w[i] + \texttt{JD.a}[k,i]\ v[\texttt{JD.j}[k,i]]$
  end foreach
end foreach
```

Algorithm 6.1: *Matrix-vector multiplication using the jagged diagonal data structure, performing* $w := Av$*. In this variant, the diagonal is assumed to be nonzero and stored in the dense vector* $\texttt{JD.d}[\star]$*. The* k*-th jagged diagonal is named* $\texttt{JD.a}[k,\star]$*, the corresponding column indices* $\texttt{JD.j}[k,\star]$*. There are* $\texttt{JD.b} = d_{\max}$ *such jagged diagonals, their lengths are* $\texttt{JD.c}[k]$*. All vectors are assumed to be reordered (see text). By replacing all* $\texttt{JD.c}[k]$*'s by* n*, this algorithm can also be used with the ITPACK data structure.*

6.7 Transposed matrix-vector products

Besides applying "regular" matrix-vector multiplication of the form $w := Av$, some of the iterative methods presented in Chapter 3 also use products with the transposed matrix in expressions of the form $w := A^T v$ (this operation can also be viewed as multiplication on the left, writing $w^T := v^T A$). Many sparse matrix data structures that support high efficiency in the regular operation are considerably less favorable for transposed matrix-vector multiplication.

We saw in Section 3.9 how the Biconjugate Gradients method can be reformulated to avoid the transposed operation. The resulting Algorithm 3.9, which is algebraically equivalent to the original formulation in Algorithm 3.8, trades one transposed product for two regular products, plus some linear operations. Still, this formulation will be faster on some machines, and for some of the data structures.

This section discusses why certain data structures behave so differently, and introduces fixes to some of them. These modifications generally improve the performance of the transposed operation at the expense of the regular operation or of storage overhead.

The easiest fix, which does not affect the performance of the regular multiplication, is to duplicate the matrix, storing A and A^T as if they were unrelated. This does not necessarily mean that the whole storage for A is doubled. If the full diagonal is stored apart, it can be reused. If the matrix is structurally symmetric, the whole description of the sparsity structure of A also describes A^T, and should not be duplicated. The modifications suggested below are less *efficient than duplication and should not* be used *in the case of* memory abundance. Unfortunately, the latter occurs very rarely.

6.7.1 Shared-memory multiprocessors

A row-oriented schemes behaves in the transposed operation as its column-oriented twin does on the regular operation. An indirect read access of a vector on the right-hand side of an assignment turns into an indirect write access to a vector on the left-hand side. Parallel indirect writes pose the problem of memory conflict: If two iterations of the same loop update the same component of a vector and if these two iterations overlap in a parallel execution, the resulting value may be wrong. Hardware solutions to assure atomicity in the updates require a lot of synchronization overhead. More efficient software solutions are similar to those for vector computers, discussed in Section 6.7.3.

6.7.2 Distributed-memory multicomputers

With a row-wise partitioning of A on a DMPP, the operation with A^T is not much more complicated than multiplication with A itself. Every processor first computes its local updates to its own local and to non-local components of the result vector. It then sends the non-local updates to the respective processors, and receives updates to its own components, which it adds to the intermediate local value.

Several updates for the same component may come from several processors, but this does not cause any problem here. A disadvantage lies only in the fact that communication and computation cannot be separated as cleanly as in regular matrix-vector multiplication.

6.7.3 Vector computers

The data structures for regular or banded systems usually support the transposed multiplication efficiently. The situation is less favorable for the structures for irregular sparsity presented above.

In the transposed multiplication operation, as depicted in Algorithm 6.2, the column index vectors in the ITPACK and jagged diagonal data structures are used in indirect write accesses, on the left-hand side of line 6.

```
1: .. foreach $i$ in $[0 : n-1]$ do
2: .... $w[i] := \mathtt{JD.d}[i]v[i]$
3: .. end foreach
4: .. foreach $k$ in $[0 : \mathtt{JD.b} - 1]$ do
5: .... foreach $i$ in $[0 : \mathtt{JD.c}[k] - 1]$ do
6: ...... $w[\mathtt{JD.j}[k,i]] := w[\mathtt{JD.j}[k,i]] + \mathtt{JD.a}[k,i]\ v[i]$
7: .... end foreach
8: .. end foreach
```

Algorithm 6.2: *Transposed matrix-vector multiplication with the jagged diagonal data structure, performing* $w := A^T v$*. The same comments as for Algorithm 6.1 apply.*

The loop on line 5 is only vectorizable if no index in each integer vector $\mathtt{JD.j}[k, 0 : \mathtt{JD.c}[k] - 1]$ occurs more than once. In the associated bipartite graph of the matrix, the set of edges corresponding to one jagged diagonal has to be a **matching**[27]. In general, the initial distribution of nonzeros among jagged diagonals does not consist of matchings only. Some additional reorganization is needed to construct the set of all jagged diagonals as a partitioning of the edges consisting only of matchings.

Note that the goal here is different from the usual matching problem, which consists of finding a single matching of maximum size or weight. We are concerned here with a **partitioning by matchings**: The union of the disjoint matchings $M_0, M_1, \ldots, M_{\mathtt{JD.b}-1}$ gives the complete set of edges. To yield jagged diagonals, the matchings have to fulfill the following condition:

---

[27] A matching in a graph is an edge subset such that no vertex is incident to more than one edge of this subset [PS82b].

> *The matchings being ordered by non-increasing size,* $|M_k| \geq |M_{k+1}|$, *a row vertex matched in* M_{k+1} *may not be exposed in* M_k.

Defining

$$\begin{aligned} E_0 &= E \, , \\ E_{k+1} &= E_k - M_k \, , \text{ and} \\ V_k &= \{v \in V | v \text{ is not isolated in } (V, E_k)\} \, , \end{aligned}$$

the two conditions above are equivalent to requiring that each M_k is a perfect bipartite matching in (V_k, E_k). Figure 6.10 shows such a **partitioning by perfect matchings.**

The usual algorithm to find a maximum matching in a bipartite graph works by augmenting alternating paths [PS82b] and has complexity $O(nm)$. If it exists, a partitioning by perfect matchings can thus be found in $O(m^2)$ time.

Unfortunately, not every graph allows such a perfect partitioning. An exposed vertex in M_k which is matched by $M_{k+\ell}$ requires the storage of an explicit zero in the k-th jagged diagonal. To include non-perfect partitionings with minimum explicit zero fill, the condition should be weakened to:

> *The matchings being ordered by non-increasing last matched row vertex,*
> $\max_i\{v_i \text{ matched in } M_k\} \geq \max_i\{v_i \text{ matched in } M_{k+1}\}$,
> *the total number of exposed row vertices with index smaller than the last matched row vertex of the respective matching should be minimum.*

In practice, the $O(m^2)$ complexity is too high. Less complicated heuristics approaching the minimum of the weak condition should be used instead. Figure 6.11 shows a partition by matchings obtained through a heuristic of time complexity $O(mn)$. Since many matrices with the same sparsity structure occur in a simulation run, this data structure initialization has to be done only once, so that the timing is acceptable.

In the ITPACK data structure, the matchings do not need to be perfect. The ideal condition is then:

> *The number of matchings should be* $d_{\max}$.

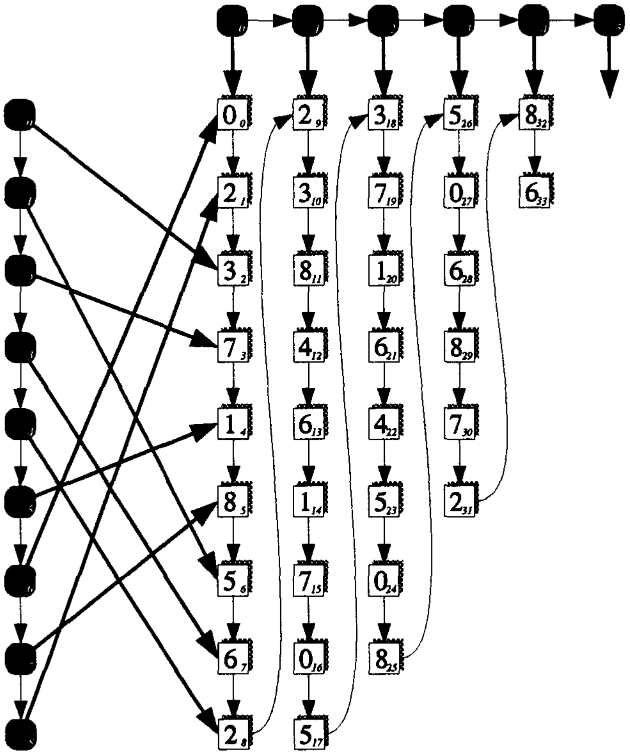


Figure 6.10: *Jagged diagonals using a partitioning by perfect matchings to store the sparse matrix of Figure 6.1.*

Again, not every graph can be partitioned into as many matchings as its maximum degree. The weaker condition is then:

> *The number of matchings should be minimum.*

6.8 Sparse triangular solvers

At each application of a preconditioner which is based on an approximate factorization (see Section 4.4), two sparse triangular systems have to be solved.

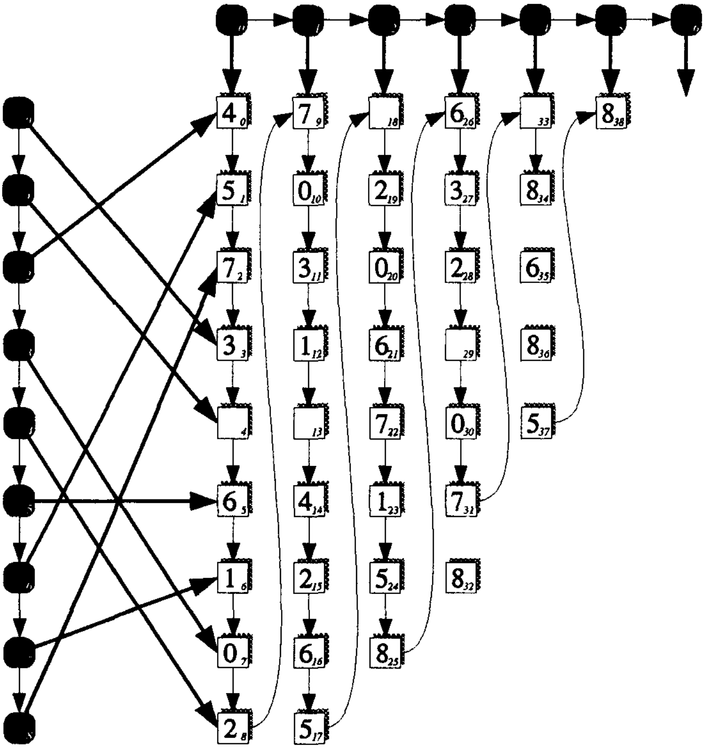


Figure 6.11: *Jagged diagonals using a heuristically obtained partitioning by matchings to store the sparse matrix of Figure 6.1.*

If the triangle including the diagonal has $(m + n)/2$ nonzero entries (as it is the case for the SSOR, D-ILU, and ILU preconditioners on a structurally symmetric matrix with m nonzeros), the solution of a triangular system by substitution (Algorithm 6.3) requires m flops.

The outer loop on line 1 in Algorithm 6.3 incurs data dependencies on the solution vector x. The inner loop on line 3 is perfectly parallel. The average number of iterations in this inner loop, however, is only $d_{\text{aver}}/2$, which is much too small to be exploited efficiently.

```
for $i := 0$ to $n-1$ do
    $x_i := b_i$
    foreach $j$ in $0 : i-1$ do
        $x_i := x_i - \ell_{ij} x_j$
    end foreach
    $x_i := x_i / \ell_{ii}$
end for
```

Algorithm 6.3: *Forward substitution by rows to solve the lower triangular system* $Lx = b$.

6.8.1 Vector and shared-memory parallel computers

Parallelism in the outer loop can be constructed by exploiting the sparsity. The chunk $x_{s:t}$ can be computed in parallel if the entries from x_s to x_t are mutually independent in $Lx = b$. This means that the block $\ell_{s:t,s:t}$ in L must be diagonal. Viewing the matrix as a block matrix in which each block on the diagonal is a diagonal matrix, the outer loop of Algorithm 6.3 can be executed in parallel inside each block row. Substitution by columns is parallelized similarly. Figure 6.12 shows the memory access pattern during one block of outer iterations.

Every block substitution involves the product of a block row of L with the updated part of x. This is again a sparse matrix-vector multiplication that has to be vectorized or parallelized as described in Section 6.6. For jagged diagonals, this means that every block row has to form a separate data structure.

There are two basic approaches to obtain a blocking as above:

1. **Identifying** parallelism by detecting independent variables (entries of x) in the computation of $x := L^{-1}b$, and grouping them together into blocks. This is nothing but standard data dependency analysis (DDA), as parallelizing compilers do it statically, but using the dynamic information given by the sparsity structure. The data dependency graph is the digraph associated with the triangular factor L, which is acyclic by construction. A different grouping may be found for U.

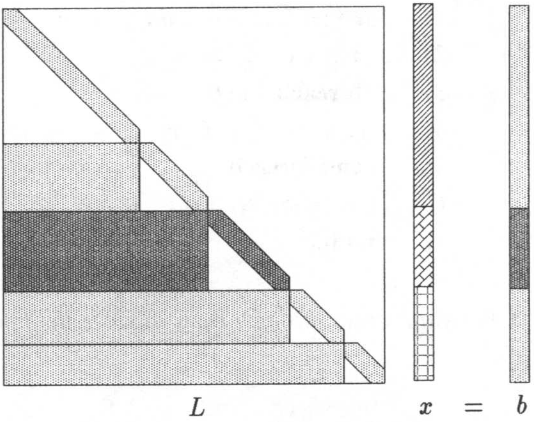


Figure 6.12: *Memory access pattern for one block in the parallel solution of a blocked sparse triangular system.*

2. **Constructing** parallelism by partitioning the associated undirected graph of the entire matrix A into sets of independent vertices. These sets are called "colors", and the partitioning is a **coloring**. The rows and columns of A are reordered by colors **before** the incomplete factorization. By construction, the blocking induced by the coloring has the desired diagonal property as required above, for both L and U.

The main difference between the two approaches is that DDA preserves the evaluation order of the incomplete factorization, while coloring modifies it. The obtained DDA preconditioner Q is just a permutation of the preconditioner without the reordering. Coloring leads to a truly different preconditioner.

As a result, the convergence behavior of the preconditioned algorithm is modified by the coloring. As they do not contain any vertex triangles (see Section 2.5), tensor-product grids (5-point stencils in 2-D and 7-point stencils in 3-D) can be colored with only two colors. This **red-black coloring** is notorious for deteriorating the convergence of iterative algorithms, leading to slower convergence than the natural ordering. The problem seems to be alleviated by using more than two colors, so-called **multicoloring**.

For irregular sparsity structures, there is no "natural" ordering generally recognized as delivering better convergence than others. In particular, there is no ordering which would be best for several numerical instances of matrices with the same sparsity structure, as we have it in device simulation. Since almost every ordering gives a different convergence behavior, there is almost certainly a particular ordering which is faster than the colored one, but in the absence of a cheap way to find this perfect ordering, the colored ordering is just as good (or as bad) as any random ordering.

Coloring then has the advantage that the number of colors is fairly small and depends only on the grid density and dimension, but not on the size of the grid. This leads to a considerably higher degree of parallelism than the DDA approach.

The algorithm used for the DDA approach is **level scheduling** [Saa89]: The first level comprises all of the vertices with no incoming arc in the digraph of L. A given vertex belongs to the k-th level if all its incoming arcs come from vertices in lower levels, and at least one comes from a vertex in the immediately preceding $(k-1)$-th level. Each level corresponds to one block in the reordered matrix.

A coloring can be obtained by the **greedy coloring** heuristic, as described in [AHU83] and depicted in Algorithm 6.4 [BCD+89, HPWF91, PAF92]. For each color, it sweeps once through the whole set of vertices (in the order they are numbered). If a vertex is not yet colored and is not adjacent to any vertex having this color, it is assigned this color. The maximum number of colors is equal to one plus the maximum degree of any vertex, but usually fewer colors are needed to color all the vertices of the graph. The typical number of colors after greedy coloring is around 7 for our 2-D grids and around 12 for our 3-D grids. The outer loop (on line 2) in Algorithm 6.4 is executed c times (where c is the number of colors), the second loop (on line 5) is executed n times on the whole, and the inner loop (on line 10) is executed m times on the whole. The set assignment on line 4 takes at most $O(n)$ time, so the complexity of the basic greedy heuristic is $O(m + cn)$. Figure 6.13 shows one color (the first color C_0 generated by Algorithm 6.4) on a 2-D grid.

Melhem and Ramarao [MR88] have described an heuristic that is guaranteed to find a 6-coloring for any planar graph. Comparing the Sun-3 execution times given in [MR88], it appears that this heuristic is much slower than greedy coloring. An extension of the method allows to find a 4-coloring for planar triangular graphs. The famous Four Color Conjecture [Har72, AH76] asserts that such a 4-coloring exists for any planar graph.

```
$R := V$
for $c := 0, 1, \ldots$ until $R = \emptyset$
  $C_c := \emptyset$
  $I := R$
  while $I \neq \emptyset$ do
    select a vertex $v \in I$
    $C_c := C_c \cup \{v\}$
    $I := I - \{v\}$
    $R := R - \{v\}$
    foreach $w \in \mathrm{Adj}(v)$ do
      if $w \in I$ then
        $I := I - \{w\}$
      end if
    end foreach
  end while
end for
```

Algorithm 6.4: *Greedy coloring, constructing in* $\{C_0, C_1, \ldots\}$ *a partitioning of the set of vertices* V.

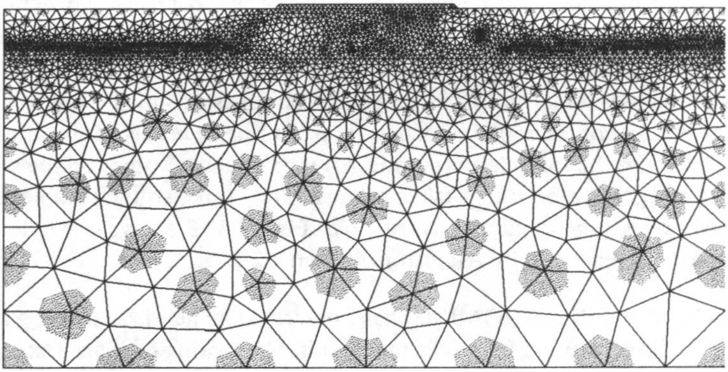


Figure 6.13: *A single color on the 2-D grid from Figure 6.3. Vertices belonging to this color are underlaid with gray.*

6.8.2 Distributed-memory multicomputers

Distributed parallelism can be identified through DDA or constructed through coloring in the same way as above. Each color block is distributed to all the processors. Processors handle the same block concurrently. Between the processing of two blocks, the needed values from the previous block are exchanged. If c is the number of blocks, a total of $(c-1)$ such synchronizations are required to solve the triangular system.

These "synchronizations" should not be seen as single points in time where all processors wait for each other. In a practical implementation on a MIMD computer solving an irregularly structured problem, each processor completes its part of a color at a different time. The processor sends then its local values to those other processors that need them. As soon as it has received all the data it needs itself, it can proceed with the next color. At a given moment, some processors still work on color C_i, others are actively communicating, waiting for incoming data, or working already on color C_{i+1}.

Coloring for DMPPs differs from coloring for shared-memory parallelization or vectorization in the following aspects:

1. Dependent vertices allocated on the same processor can be processed in the same color since no communication is required. The edges between local vertices need not be considered for the coloring of the graph.
2. The number of colors must be as small as possible to reduce the synchronization overhead.
3. Coloring and mapping (Section 6.6.2) should be combined to ease the coloring task and to potentially reduce the number of colors that are required. We can define three categories of coloring strategies, depending on their relationship with the mapping strategy:
    (a) Mapping and coloring are done **independently** of each other. In this case, the above property is not exploited, and all vertices in one color must be independent of each other regardless of their assignment to processors. We can thus use the same coloring heuristics as above for vector computers.
    (b) Coloring is done **after** mapping. After the mapping heuristic has assigned vertices to processors, the edges between vertices mapped to the same processor can be identified and ignored in the

coloring heuristic. The first requirement for mapping strategies mentioned in section 6.6.2 ensures that as many edges as possible can be deleted this way.

(c) Coloring is done **before** mapping. Pairs of adjacent vertices that have received the same color must be assigned to the same processor in the later mapping stage. The mapping scheme has to be adapted to satisfy this additional requirement.

Every block substitution (in Figure 6.12) consists of a sparse matrix-vector multiplication. The blocks of L and U are distributed, and their assignment to processors has to respect similar communication-related criteria as the seven points defined at the beginning of Section 6.6.2. This discards variant 3a above: mapping and coloring are closely interrelated. It is imperative that each color is distributed equally to the processors [PAF92].

Algorithm 6.5 shows **balanced greedy coloring**, which constructs a suitable coloring for DMPPs after mapping (for variant 3b above). This heuristic tries to balance over all processors the number of vertices of each color. At each iteration of the loop starting on line 11, every processor donates at most one vertex to the color. As long as the *flag* is not set, every processor donates exactly one vertex to the color. The **maximum coloring imbalance** parameter $i_{\max}$ controls by how many vertices the size of each color assignment may vary among processors. The clause $(q \neq p)$ in line 22 allows the same color for adjacent vertices mapped to the same processor (aspect 1). On machines with additional vectorization capabilities on each processor, this clause should be omitted.

The number of colors generally increases with more restrictive values for $i_{\max}$, so that the better balance gets outmatched by more synchronization for the colors. The value $i_{\max} = 2$ appears to be optimal in many cases. With such a small $i_{\max}$, the complexity of balanced greedy coloring is the same as for absolute greedy coloring. The number of colors is in the same range or lower for the balanced coloring, profiting from aspect 1 mentioned above.

Figure 6.14 shows the first color (C_0 in Algorithm 6.5) of balanced greedy coloring (with $i_{\max} = 2$) on the grid of Figure 6.3, using the mapping of Figure 6.5. This color set is large, as it contains most, but not all of the inner vertices (that is, vertices requiring no communication). The heuristic assigned some of the inner vertices to the second color to respect the maximum coloring imbalance.

```
foreach processor $p$ do
  $R_p := V_p$
end foreach
while $\bigcup_p R_p \neq \emptyset$ do
  $C_c := \emptyset$
  $i := 0$
  $flag := 0$
  foreach processor $p$ do
    $I_p := R_p$
  end foreach
  while $i < i_{\max}$ do
    foreach processor $p$ do
      if $I_p = \emptyset$ then
        $flag := 1$
      else
        select a vertex $v \in I_p$
        $C_c := C_c \cup \{v\}$
        $I_p := I_p - \{v\}$
        $R_p := R_p - \{v\}$
        foreach $w \in \mathrm{Adj}(v)$ do
          find $q$ such that $w \in V_q$
          if $(q \neq p)$ and $(w \in I_q)$ then
            $I_q := I_q - \{w\}$
          end if
        end foreach
      end if
    end foreach
    $i := i + flag$
  end while
end while
```

Algorithm 6.5: *Balanced greedy coloring for a maximum coloring imbalance* $i_{\max}$. *Each* V_p *stands for the set of vertices mapped to processor* p.

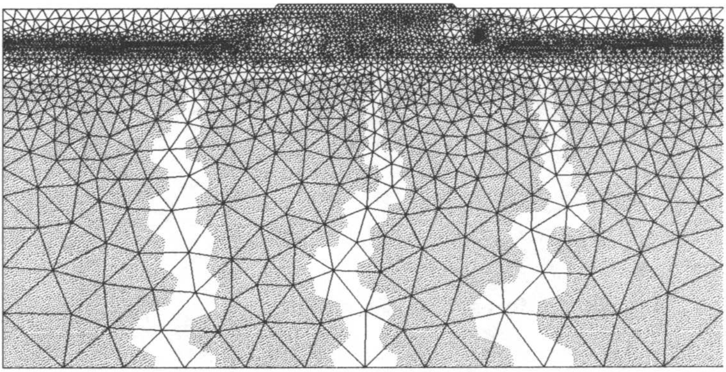


Figure 6.14: *The first color generated by the balanced greedy coloring Algorithm 6.5 on the 2-D geometric mapping (Figure 6.5) of the 2-D grid in Figure 6.3 on the* 4×4 *processor mesh of Figure 6.4. Vertices belonging to this color are underlaid with gray.*

The solution of the sparse triangular systems with L and U in the block structure obtained through coloring with c colors is now performed by $(2c-2)$ matrix-vector products with an average of $m/(2c-2)$ nonzeros in each block. On the $n^{1/3} \times n^{1/3} \times n^{1/3}$ tensor-product grid mapped to $p^{1/3} \times p^{1/3} \times p^{1/3}$ processors considered as an example at the end of Section 6.6.2, average message sizes would now be $\frac{n^{2/3}}{p^{2/3}(2c-2)}$. For $n = 1,000,000$ and $p = 1,000$, the average message size would definitely be smaller than 10 numbers, and a significant number of messages would consist of a single number. Message latency is the most important machine characteristic determining the performance of incomplete factorization preconditioners on DMPPs. Solving sparse triangular systems can be practically as efficient as sparse matrix-vector multiplication on machines where the time to send a single message is of the order of one or a few floating-point operations. Networks with higher latency are practically useless for iterative solvers with incomplete factorization preconditioners, no matter what the peak bandwidth may be.

6.9 Setting up the preconditioner

The storage requirements and the operation counts for the different preconditioners are discussed in Chapter 4. In this Section, we concentrate on how to set up the preconditioner **efficiently**.

There is not much to say about the Jacobi and SSOR preconditioners. All the information is already available. The inverse of the diagonal should be computed in advance, to trade divisions for multiplications when applying the preconditioners. If split preconditioning of the form (4.17) is used, square roots and their inverses should be computed in advance as well.

The use of nested iterative solvers for preconditioning is mainly a problem of organization, and is treated in Section 6.10.

True incomplete factorizations are more critical, both because of their higher algorithmic complexity and because their flops count is in the order of magnitude of one or several iterations. Moreover, non-numerical operations start dominating the execution time.

Not all the ideas for parallelizing sparse direct solvers can be transferred to incomplete factorization. The major source of parallelism in direct methods comes from the fact that some submatrices in the computation are or can be treated as dense matrices. This density results here from (partly even deliberately included) fill, and is thus not available in factorizations without or with little fill.

The **elimination tree** is another source of parallelism exploited in direct solvers [DDSvdV91, Liu90]. The elimination tree is a graph that defines a partial order for the elimination of the rows of a matrix: If v_i is an ancestor of v_j in the tree, then row i must be eliminated after row j. Independent subtrees, however, may be processed in parallel. For increased parallelism, the matrix is reordered such that its elimination tree becomes flatter. This idea is crucial for the parallel approximate factorizations suggested below.

6.9.1 D-ILU factorization

The diagonal matrix D for D-ILU factorization is computed through

$$d_{ii} := a_{ii} - \sum_{k=0}^{i-1} a_{ik} d_{kk} a_{ki} \ . \tag{6.3}$$

Equation (6.3) is looped over the index i, so that there is a data dependency between d_{ii} and all the previous d_{jj}'s. **Coloring** delays this data dependency from one color to the next color: if v_i and v_j are within the same color, then $a_{ij} = a_{ji} = 0$, so that d_{jj} does not need to be used. An entire color can therefore be handled in parallel. See Figure 6.15 as an illustration.

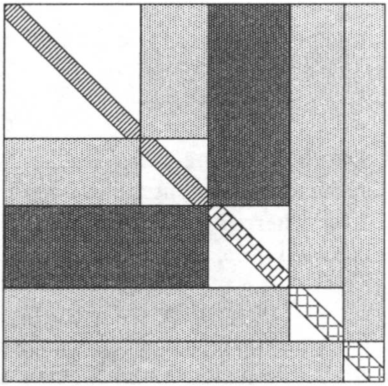


Figure 6.15: *Memory access pattern for colored D-ILU factorization, with the upper triangle stored by row blocks, and the lower triangle stored by column blocks.*

This is really efficient on a (shared- or distributed-memory) parallel computer only if the lower triangle of A can be accessed by rows and the upper triangle of A can be accessed by columns. On a vector computer, long vectors can be obtained again with the ITPACK storage scheme or with jagged diagonals. Row oriented jagged diagonals (as described in Section 6.6.3) are used for the block columns of the upper triangle, and column oriented jagged diagonals for the block rows of the lower triangle.

6.9.2 ILU factorization

Blocking by colors is also the strategy to be followed in parallelizing ILU factorization. Every single block, the intersection of a block row and a column block, is stored as an independent sparse matrix. The parallelizable operation is then the multiplication of two single blocks of the already computed parts of L and U and their addition to the block currently being updated (see Figure 6.16).

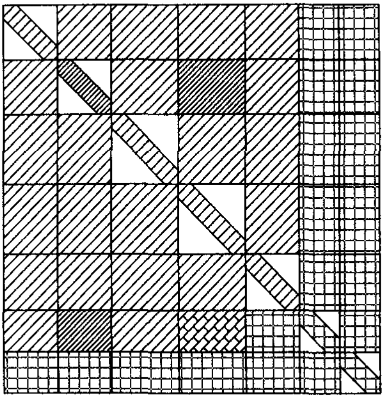


Figure 6.16: *Memory access pattern for colored ILU factorization.*

Since we are writing now into a sparse matrix (whose structure is known) and not into a dense vector as in the previous section, jagged diagonals and partitionings by matchings are not sufficient to avoid data dependencies without leading to quadratic complexity.

ILU factorization can be parallelized efficiently using a **distance-two coloring**, i.e., the colors are selected such that no vertex within the distance of two edges has the same color. This entails that every single block, that is, the set of edges connecting the vertices of two given colors, is a single matching. Every single block update from two finished blocks, as shown in Figure 6.16, can now be performed completely in parallel.

By replacing "$w \in \mathrm{Adj}(v)$" in line 10 of Algorithm 6.4 by "$w \in \mathrm{Adj}(v) \cup \mathrm{Adj}(\mathrm{Adj}(v))$", we have a **greedy distance-two coloring heuristic**, with a complexity of $O(m^2/n + cn)$. Note that the number of distance-two colors will be higher than for a simple (distance-one) coloring.

6.9.3 Positional dropping factorization

Factorizations with some limited fill are far more difficult to parallelize. For positional dropping strategies, the procedure can be subdivided into a symbolic factorization that determines the structure of the resulting matrix, and a numerical factorization that computes the values and puts them into the positions predetermined in the previous step. Just as in the case of direct solvers without numerical pivoting, the symbolic factorization is valid for all matrices with the same sparsity structure, so more effort can be applied to obtain a parallelizable structure.

For the sequential implementation, refer to [MvdV77, Saa89, BS87]. Depending on the dropping strategy, extensions to the graph coloring and matching concepts as presented in the previous Sections 6.9.1 and 6.9.2 can lead to higher parallelism.

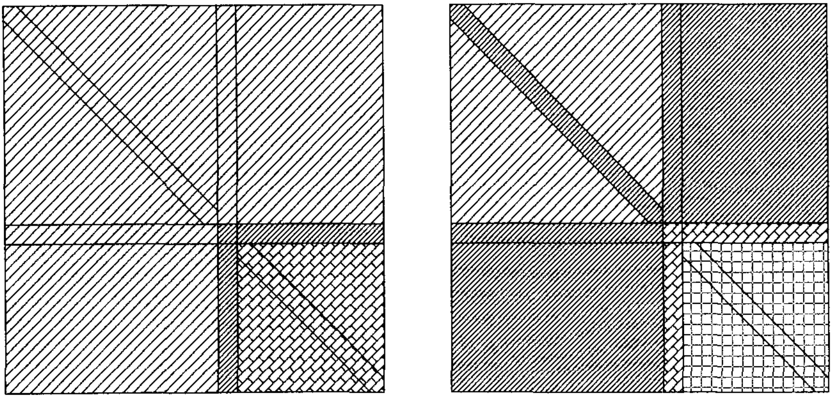
6.9.4 Numerical dropping factorization

Numerical dropping does not permit a separation of the work as with positional dropping. Symbolic factorization, that is, the selection of which entries are nonzero and in what order they have to be processed, must be done along with the numerical factorization, where the values are actually computed.

We would like to be able to choose the dropping strategy and the dropping criterion freely, relying only on numerical considerations. Unfortunately, most such freely chosen combinations do not allow efficient implementations of the approximate factorization algorithm. Therefore, we start with the factorization algorithm and then justify the choice of the dropping strategy and the dropping criterion, based on efficiency, memory consumption, numerical, empirical, or inspirational arguments.

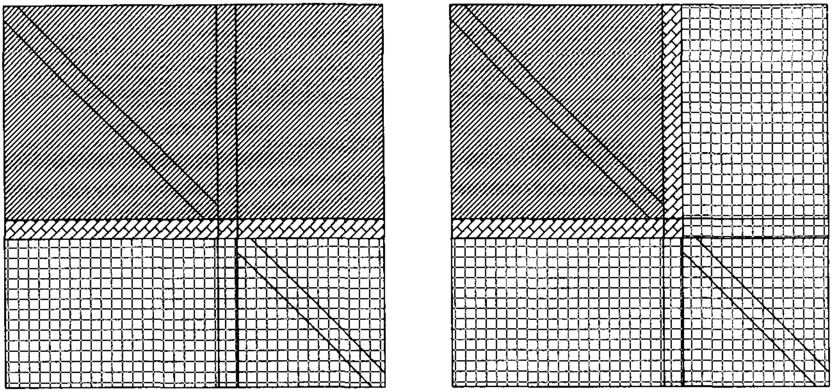
6.9.4.1 Choice of the base algorithm

Depending on the relative nesting of the three loops over the indices in i, j, and k in Equations (4.12) to (4.14) on page 74, and the order by which parts of L and U are constructed, several variants of LDU-factorization can be distinguished. These variants differ by their access pattern on the matrix.



(a) Gaussian elimination (kij-form)

(c) Crout (jki-form)



(b) Doolittle (jki-form)

(d) bordering (ikj-form)

Figure 6.17: *Memory access patterns for different formulations of LDU-factorization.*

Figure 6.17 shows the memory access patterns of several variants of LDU-factorization (see also [Ort88, DDSvdV91]). The classical form of Gaussian elimination (kij-form [GvL83]) is badly suited for numerical dropping. At the k-th iteration of the outer loop, all the values in the block below the k-th row and right of the k-th column of the matrix get updated. The temporary storage to keep the values of these entries, which are not the final values, can easily become almost as large as the temporary storage for a complete factorization. Storage requirements can be reduced by dropping small partial values of an entry whose final value is not known yet, but this induces another source of inaccuracy that could be avoided.

A formulation that updates a smaller part of the matrix, like a single line of L and U, is to be preferred. This line can be cleaned of unimportant entries at the end of each outer iteration, so that temporary storage never exceeds $O(n)$. The variants known as Doolittle, Crout, and bordering have this property. I selected the **bordering algorithm** ([Ort88, BR90], Figure 6.17(d), Algorithm 6.6), as the basic factorization algorithm for numerical dropping.

6.9.4.2 The dropping strategy

The dropping strategy specifies *when* in the factorization process *which* entries of the factors should be presented to the dropping criterion. In this section, the application of the dropping criterion is symbolized by the operator Censor(l_{ij}). This operator checks the dropping criterion on the value of l_{ij}, and annihilates it (i.e., sets $l_{ij} := 0$) if it is not important enough. The censoring operator can take a list of arguments, each one of them being examined separately.

Algorithm 6.6 shows four locations where the censoring operator could be placed in the factorization algorithm. Each of these locations corresponds to a different **dropping grain size**:

1. Line 11 examines single contributions (summation terms) to an entry.
2. Line 7 examines each entry as soon as its final value is known (after all the contributions have been added).
3. Line 18 examines a line of each factor as soon as this line has been computed entirely.
4. Line 20 examines the entire factor matrices at the end of the factorization.

Since the censor is a data-dependent operation, a fine grain size corresponds to finely dispersed data-dependencies. The coarser the grain size, the more opportunities for an efficient and parallelizable implementation arise. On the

```
for $i := 0$ to $n-1$ do
  $l_{i,0:i-1} := a_{i,0:i-1}$
  $u_{0:i-1,i} := a_{0:i-1,i}$
  for $k := 0$ to $i-1$ do
    $l_{ik} := l_{ik}/d_{kk}$
    $u_{ki} := u_{ki}/d_{kk}$
    Censor($l_{ik}, u_{ki}$)
    foreach $j$ in $[k+1 : i-1]$ do
      $l_{ij} := l_{ij} - l_{ik}d_{kk}u_{kj}$
      $u_{ji} := u_{ji} - l_{jk}d_{kk}u_{ki}$
      Censor($l_{ij}, u_{ji}$)
    end foreach
  end for
  $d_{ii} := a_{ii}$
  foreach $k$ in $[0 : n-1]$ do
    $d_{ii} := d_{ii} - l_{ik}d_{kk}u_{ki}$
  end foreach
  Censor($l_{i\star}, u_{\star i}$)
end for
Censor($L, U$)
```

Algorithm 6.6: *The bordering factorization algorithm, with possible locations for the censoring operator for numerical dropping.*

other hand, a coarser grain size also leads to a higher number of unimportant contributions and entries that have to be stored intermediately and processed to contribute to other entries.

Dropping only at grain size 4 would require a full factorization, which is out of the question. For smaller grain sizes, storage size is not a problem with single-line factorization variants, as was mentioned in Section 6.9.4.1.

Experiments with an earlier variant of the current implementation showed that dropping after entire lines (grain size 3) is too late. The uncensored lines were by a considerable factor denser than the final lines after dropping unimportant entries.

Switching to single entry dropping (grain size 2), the time spent evaluating the dropping criterion starts dominating the entire factorization algorithm. For this reason, dropping individual contributions (grain size 1) is counterproductive in timing.

6.9.4.3 Complexity observations

All of the inner loops in Algorithm 6.6 are sparse loops. They all have to be supported by fast access data structures in order to keep the complexity of the factorization low. The following observations do not constitute a complete complexity analysis of the approximate factorization algorithm, but they point out the important aspects to help in the choice of the data structures and the dropping criterion.

The complexity of the full sparse bordering factorization has been studied in detail by Bank and Rose [BR90]. Unfortunately, their elimination tree-based sparse bordering algorithm could not be adapted efficiently to numerical dropping[28]. The two major complexity issues identified in [BR90], the **intersection problem** and the **sorting problem**, have a similar importance in sparse bordering with or without numerical dropping.

Let m_L be number of nonzeros kept in the lower triangular factor, and let $\check{m}_L$ be the number of entries inspected (submitted to the grain size 2 censor on code line 7). Let m_U and $\check{m}_U$ be similarly defined quantities for the upper triangular factor. Obviously, $n \leq m_L \leq \check{m}_L \leq n^2$ and $n \leq m_U \leq \check{m}_U \leq n^2$ hold.

A dense matrix implementation of Algorithm 6.6 would step through all the indices for each loop, so that the total number of iterations of the middle loop (starting at code line 4) is $n^2/2$, and there are a total of $n^3/6$ inner

---

[28] This algorithm [BS87, BR90] identifies each nonzero u_{kj} (on code line 9) by chording each cycle formed by the path from vertex v_i (which is currently being eliminated, in the i-th outer loop iteration) to a leaf v_k in the elimination tree and the corresponding backedge (v_k, v_i). In an implementation with numerical dropping, a chord corresponding to a dropped entry u_{kj} does not require any floating-point computation, but the chord still has to be traversed to ensure the proper functioning of the elimination tree algorithm. Thus, the non-numerical complexity of such an approximate factorization stays the same as for an exact factorization.

iterations (in the loop starting at code line 8)[29].

The censor is applied to $\check{m}_L$ entries in the lower triangle and to $\check{m}_U$ in the upper triangle. It makes sense at this point to split the middle loop in two, one computing $l_{i\star}$ and one computing $u_{\star i}$. Code lines 5 and 7 (that is, Censor(l_{ik}) in the split loop) are executed $\check{m}_L$ times (over all outer iterations). The inner loop, however, is started only m_L times. Assuming a uniform distribution of nonzeros over the entire matrices L and U, the average number of inner iterations updating entries of L is $2m_U m_L i^2/n^4$. In summary, the factorization performs $\frac{2}{3}m_L m_U/n$ times either of the lines 9 and 10.

It is clear that this inner loop (which constructs the **intersection** of the nonzero sets of $l_{i\star}$ and $u_{\star k}$) is a candidate for dominating the timing of the algorithm, and so it should be made as efficient as possible. To keep the overall complexity of all the inner loops at $O(m_L m_U/n)$, all operations must be $O(1)$.

The inner loop is not the only thing that can dominate the timing of the algorithm. The number of (split) middle iterations, as we saw above, is $(\check{m}_L + \check{m}_U)$. If the dropping rate $\check{m}_L/m_L$ is high (higher than the average number of nonzeros per row m_U/n), lines 5–7 are carried out more frequently than lines 9–11.

The execution order of the iterations of the middle loop poses another problem. The data dependency between l_{ij} and l_{ik} in line 9 imposes a partial order on the index k in line 4. New fill entries are generated in a more or less random order in line 9. This partial **sorting** problem cannot be solved constructively via the elimination tree, as in full factorizations or factorizations with positional dropping [BS87, BR90]. The average number of candidates in $l_{i\star}$ for the next index in line 4 is $\frac{2}{3}\check{m}_L/n$.

The index search for the middle loop thus appears to be a major bottleneck and requires special attention. A naive implementation with a linear search would lead to a complexity of $O(\check{m}_L^2/n)$, which is the highest complexity we found up to now. The access characteristics of the set of unprocessed indices are those of a **priority queue**. The fastest implementations of this abstract data type have logarithmic complexity for each access, so that the overall complexity of the index search can be lowered to $O(\check{m}_L \log(\check{m}_L/n))$.

---

[29] To be precise, there are $(n^2/2 - n/2)$ middle iterations and $(n^3/6 - n^2/2 + n/3)$ inner iterations.

In summary, the overall complexity of the numerical dropping factorization is

$$O\left(\underbrace{\frac{m_L m_U}{n}} + \underbrace{\breve{m}_L + \breve{m}_U} + \underbrace{\breve{m}_L \log \frac{\breve{m}_L}{n} + \breve{m}_U \log \frac{\breve{m}_U}{n}} \right) . \tag{6.4}$$

Only the first term and the second term involve floating-point operations. The second term appears to be shadowed by the third term. It is included here to emphasize that if line 7 has a non-constant complexity, then the complexity of this censoring operator comes in as a multiplicative factor to the second term in (6.4).

Each of the terms in Expression (6.4) can dominate—the complexity of the dropping criterion and the dropping rate decide. The dropping rate can be expected to behave similarly to the fill factor of a full factorization (which increases with the dimensionality of the problem, see Section 2.7).

6.9.4.4 Data structures to handle sparsity

In the following, the data structures to store the lower triangular factor L is described in detail. The upper triangle U is stored in similar structures, whose construction is symmetrical (that is, the roles of rows and columns are inversed).

To keep the first term in (6.4) low, line 10 has to find l_{jk} in constant time. In other words, the inner loop accesses entries from the **previous rows** of L by columns. Still, it is not possible to store single columns of L in a contiguous compressed array, as entries are generated by columns. Linear lists for each column are used instead. Figure 6.18 shows a data structure that allows row-wise writing and column-wise reading with constant complexity per access (an elementary operation on a simply linked list) and minimum possible storage overhead (one row index and one pointer per nonzero entry).

The values of the **current row** of L are stored in an expanded vector of full length n. Their set of indices is accessed as a priority queue, for which a **heap** [AHU83, Sed88] is the appropriate data structure. A heap does not provide a fast way to tell whether a given index is present or not, so that an additional boolean array is used to indicate whether the corresponding entry of the current row is zero or not.

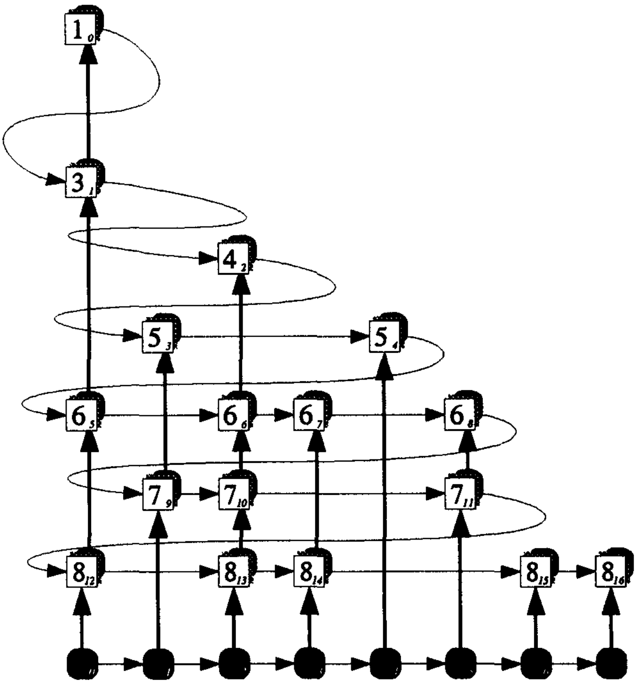


Figure 6.18: *Data structure for the L factor during numerical dropping factorization.*

Because of the sequential (row by row) construction of the data structure for kept entries of L, it is easy to convert this data structure (Figure 6.18) into a compressed row structure (Figure 6.2). All that remains is to replace row indices by column indices, which is done in $O(m_L)$ time using the column pointers. Column-wise access is not needed to apply the preconditioner, so the column-pointer array can be deallocated and reused for other purposes.

6.9.4.5 The dropping criterion

The implementation considerations above exclude several variants for the dropping criterion. I selected a local row-relative criterion on L: The row censor (on code line 18) keeps an entry l_{ij} if $|l_{ij}| \geq \tau \| l_{ij} \|_{\max}$. In order to drop unimportant entries even earlier, the single entry censor (on code line 7) maintains a running maximum, keeping l_{ij} if $|l_{ij}| \geq \tau \| l_{i,0:j} \|_{\max}$. A similar local column-relative criterion is used on U.

A contribution censor (on code line 11) is not used because it would dominate the time. A full matrix censor (on code line 20) is also not implemented, as it would not save any storage and make it impossible to convert the data structure into a compressed row format.

Except for a local absolute criterion, other possible choices for criteria would either cost more than $O(1)$ per censor application, or take more intermediate storage.

6.9.4.6 Parallelism

The implementation of the numerical dropping factorization presented above is entirely sequential. The main operations walk through pointer lists and offer practically no parallelism. The number of operations between successive censor applications is only $O(m_L/n + \log(\breve{m}_L/n))$, so that even if more parallelizable data structures (with a higher total operation and storage count) are chosen for the intersection problem and the sorting problem, the amount of parallelism between the (data dependent) censor calls is very low. On the other hand, delaying the censor rapidly increases the overhead.

A higher amount of parallelism might appear if the matrix is reordered by a nested dissection [GL81] or the mapping heuristics presented in Section 6.6.2. Knowing that the current implementation (using still other implementation tricks to increase performance, like automatic inlining) exceeds already a dozen pages of code, such a parallel version will be quite complex.

6.10 Flexibility

6.10.1 Solver types

An **iterative solver** is a combination of an iterative solution algorithm, a preconditioner, a position of the preconditioner, a termination control mechanism, and some other parameters. Such a combination can be called a **solver type**. A solver package implements several such solver types. Ideally, it should provide all orthogonal combinations of the parameters, or at least those that make sense.

The solver type that appeared to be fastest on most device simulation problems is, as pointed out before, split D-ILU preconditioned Bi-CGSTAB. The most robust type was GMRES(∞) preconditioned on the right with numerical dropping.

6.10.2 View of the client application

A client application wants to see a linear solver only as a single function that is called with a matrix and a right-hand side as input parameters and returns the solution on output. This is called a **black box solver**.

Reality is more complicated. The solution may not be representable in a floating-point format. Even the closest representable solution may not be computable by any algorithm (even by direct solvers). Our black box needs more input knobs to quantify what should be done.

The requested accuracy and the type of convergence criterion are two of these additional parameters for an iterative solver. Requesting maximum achievable accuracy wastes time if only a few digits are needed. The client may know more about a particular problem, like a good initial guess x_0, definiteness of the matrix, or other information that may suggest that one particular combination of an iterative method and a preconditioner is faster than another.

Thus, the solver package should select a default solver type that serves most of the cases efficiently, but the client application should have a way to manipulate the parameters to obtain a solver type that suits its needs.

6.10.3 Automatic adaptation

Closer to the idea of a black box solver would be a package that selects automatically the fastest solver type for each linear system. Unfortunately, there is no easy way of determining beforehand which type is the most appropriate.

An approximation to this is to try first the solver type that is fastest in most cases. If this type does not attain convergence in a given maximum number of iterations, the package switches to a slower, but more robust solver type. The initial guess of this second solver type can be the final approximation of the first solver type.

6.10.4 Experimenting

It should be clear at this point that many properties of iterative solvers are still poorly understood. More theoretical framework is needed and many experiments have to be conducted (and had to be conducted, for instance, before this thesis could be written).

Many researchers in iterative methods write a small experimental code fragment, concentrating only on one aspect (e.g. the method, the preconditioner, the parallelism) and ignoring the others. They examine the behavior of their code only on a set of model problems, or at most on a few problems from a collection. This approach is acceptable for making first statements about a new idea, but insufficient to predict its effectiveness inside a real application.

On the other hand, experimenting only inside complete applications is often too expensive, and leads to large measurement errors through interactions with other effects. There have to be ways of extracting a snapshot of a real situation (like one single matrix together with the right-hand side, the initial guess, the convergence tolerance) and doing more exhaustive experiments with a stand-alone solver tool. Conversely, set-ups investigated on a single test problem should be easily and immediately transferable to the solver library that is integrated in an application.

As a consequence, experimentation tool and fully-fledged application library in such an evolving field as iterative solvers should be only one package.

6.10.5 Expansions

Research in iterative solvers is currently a very active field. Major new ideas, such as new methods, new preconditioners, new implementations (motivated by new architectures), new uses (through new algorithms or models in the client application), have been invented or put into practice only very recently, and will continue to emerge. Flexibility in an implementation includes the ability to insert even unconventional ideas (like, for instance, recursively nested iterative solvers, see Section 4.5).

6.10.6 Configuration control

Application programs usually control numerical software libraries through an additional "parameter array" supplied along with a function call. Each entry of this array contains a value that selects one particular option. The most important switches may be supplied as individual function parameters, some of them even as individual function calls. Options in the application itself are usually (at least in device simulators) selected in a control file, using some simple command language. For an experimentation tool, control information should be given interactively (through keyboard input or menus) or through command line options. Common settings for a longer series of experiments are best specified by a control file.

In light of Section 6.10.4, all control information to a solver package should have the same format. In light of Section 6.10.5, this format should be easy to extend to new features. In light of Section 6.10.2, default settings should be in the same format, and client applications should be able to generate this format. More complicated set-ups such as automatic adaptation (Section 5.7) or nested solvers (Section 4.5) ask for a less static specification than parameter arrays.

One solution to this is to use a simple **declarative language** to specify the options. Defaults are specified in a constant string interpreted at the first invocation of the solver package. Applications modify settings by passing a character string (like "`method=cgs`") to the package. The control file of the application may contain sections with plain solver control declarations. In other applications, the solver package can also be controlled by an extra solver control file. Reading this file at each solver call allows dynamic control over the behavior of the solver, even without intervention of the client application. Command line options, keyboard input, and file input to the experimentation tool use the same declarative language. If the language interpreter is written using a standard parser generator (like the UNIX tools YACC and LEX), the language can be extended very easily.

6.10.7 Object-oriented package design

At a given point of each Krylov subspace method, a preconditioned matrix-vector product has to be computed. Depending on which preconditioner is currently being used, one particular function[30] is called. How should this be implemented ? Many numerical packages would just use a list of conditionals or case-statements to select the path to go. Some would even use reverse communication [AS90, Her91], and exit the iterative algorithm with a return code that tells the caller function to apply the preconditioner to a return vector and reenter the algorithm later.

A cleaner software engineering approach identifies the application of the preconditioner as an operator of an abstract data type, or as a **method** of the **object** instantiating the preconditioner. An object-oriented view sees the (concept of a) preconditioner as a **virtual class** [BDMN79], and each particular type of preconditioner (such as ILU, ND, nested solver) is a subclass of the general preconditioner class. The iterative method calls a virtual method of the general preconditioner, and dynamic binding [Mey88] substitutes the correct instance method of the selected preconditioner.

In addition to the preconditioner, other virtual, dynamically exchangeable objects turn up, like the matrix, the iterative method, the termination control mechanism, and the breakdown and restart handlers. Figure 6.19 presents a simplified extract of the code organization in an object-oriented approach for an iterative solver package. The termination control does not see which iterative methods it controls, the iterative method activates a matrix-vector multiplication function without knowing whether the matrix is preconditioned or not, and which preconditioner is used.

Note that the object-oriented concept for an iterative solver package does not necessarily require an object-oriented programming language for its implementation. Furthermore, at the object grain size suggested in this section, efficiency is not critical.

---

[30] In order not to get involved in concepts specific to one particular programming language at this point, programming language concepts are named as in [Set89].

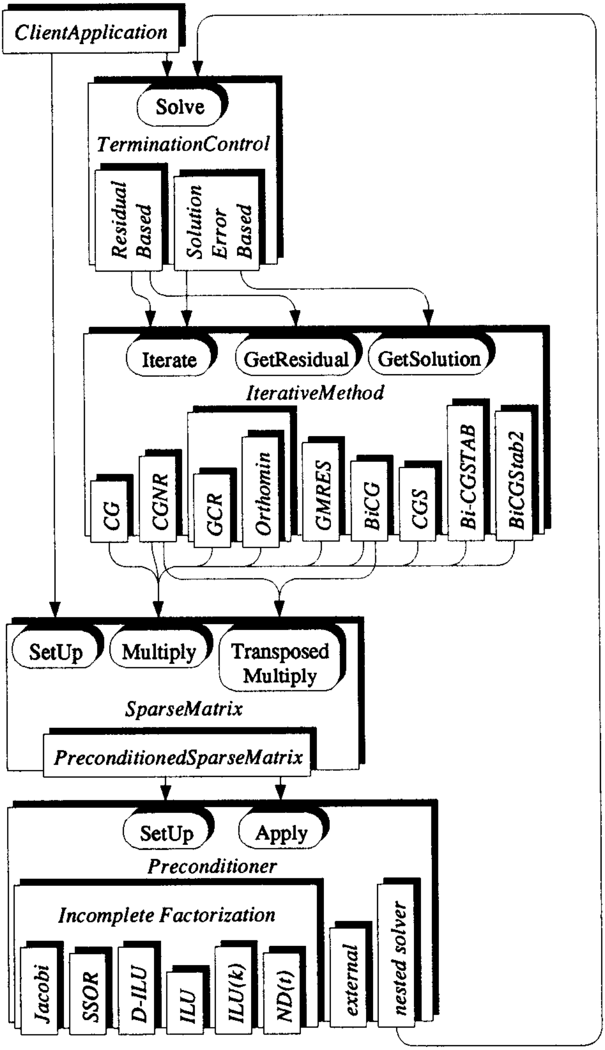


Figure 6.19: *An extract of the class structure for an object-oriented implementation of an iterative solver. Boxes stand for classes, boxes on boxes for subclasses, rounded buttons for methods, and arrows for accesses.*

6.11 Portability

The resource requirements for the client applications of an iterative solver are such that it should be able to run on the fastest supercomputers available. On the other hand, software development, as well as smaller application runs, are more convenient on smaller platforms, like workstations. In certain situations, minisupercomputers are more economical.

The application must therefore be highly portable, as users will not accept working with different interfaces each time they switch to another machine. Although it may be acceptable for a client application implementor to use different, machine-specific and manufacturer-provided tuned packages on each machine, it is also preferable to have the numerical software packages portable.

As we know that the linear solver dominates the computation time in device simulation, the solver package should even be the most portable part of all. It can then be used for benchmarking new machines, and predict their performance on the whole application without having to port and run the entire application.

6.11.1 Parallelization and vectorization

Except for the termination control mechanism (which has constant complexity and can be ignored in efficiency considerations), the operations discussed in Sections 6.4 to 6.8 are the only computations that have to be done in each iteration. With the techniques presented above, all these operations are perfectly vectorizable and parallelizable. The focus is now on how an efficient vectorization and parallelization can be portable.

Linear operations on vectors and vector dot products are part of the BLAS1[31] library [LHKK79]. All supercomputer manufacturers provide highly tuned assembler implementations for BLAS. Still, better performance is obtained by coding these operations as loops, and leaving the tuning to an optimizing compiler. The compiler can minimize memory access and optimize vector pipeline utilization by grouping and chaining several vector operations. An implementation based on BLAS has to split larger vector expressions into simpler operations that match BLAS routines. For expressions

---

[31] Basic Linear Algebra Subroutines, level 1: vector-vector operations.

matching BLAS exactly, the compiler substitutes the corresponding BLAS implementation, anyway.

As BLAS is intended for dense matrix linear algebra, it does not include indirect addressing, as used in the inner loops of Algorithms 6.1 and 6.2. These loops, which dominate the performance in sparse matrix-vector multiplication and incomplete factorization preconditioning in a jagged diagonal implementation, are included in one proposal to extend BLAS to sparse matrices [OK90], but omitted in another proposal [DGL91]. Neither one of these extensions is currently popular enough for an implementor to rely on its availability.

No compiler would vectorize or parallelize spontaneously the inner loop (on code line 5) of Algorithm 6.2. The compiler does not know that using the index vector $\texttt{JD.j}[k, 0 : \texttt{JD.c}[k] - 1]$ on the left-hand side of the assignment does not incur a data dependency. We have to insert a **compiler directive** (or compiler pragma) telling the compiler that each entry in the index vector occurs only once.

6.11.2 Choice of a programming language

Despite its age, FORTRAN is still by far the most widely used programming language for numerical applications, and it is usually available on all platforms intended for scientific computing[32]. In the last years, C has reached the same level of availability. ADA compilers generally exist, too, but only as an (usually expensive or export-restricted) option. C++ can be made available everywhere C exists, because of a good C++-to-C preprocessor. All other languages have to be excluded because of their current lack of availability.

An important point is inter-language compatibility. A package should be callable from any language. Although there are syntactical differences from one manufacturer to another (e.g. FORTRAN compilers add here and there an underscore character to identifiers), it is not too difficult to have mutual calls between FORTRAN, C, and C++.

It often turns out that functionally equivalent code fragments perform better if the code is written in FORTRAN rather than in C. One reason for this is that, knowing how much numerical software is written in FORTRAN, many

---

[32] As FORTRAN 90 is not sufficiently available yet, we consider here only FORTRAN 77.

manufacturers put more development effort into the optimization capabilities of the FORTRAN compiler. Another reason is conceptual differences. FORTRAN assumes (that is, leaves the responsibility to the programmer) that arrays passed as parameters do not overlap each other. C treats such arrays by definition as general pointers, which are allowed to point to the same or overlapping regions. This fact inhibits optimizations (as automatic vectorization and parallelization) in many cases in C. The programmer has to insert another pragma, telling the compiler that the vectors on the left-hand side of an assignment do not overlap with anything on the right-hand side. Advanced C compilers may solve this problem through interprocedural pointer tracking [LMSS91].

One weak point of C used to be its lack of redundancy, but this problem has largely been reduced through the strong typing in ANSI C and C++.

The essential disadvantage of FORTRAN is its poor expressibility. The absence of true data structuring and abstraction capabilities makes handling of more complicated data structures (such as those in the previous sections of this chapter) clumsy and unreadable. Some of the flexibility issues mentioned in Section 6.10 are hard to deal with. Because of the lack of dynamic memory allocation, many FORTRAN library functions expect the caller to provide an additional "work array" parameter that should be sufficiently large to fit the internal data structures of the callee. This is not really necessary, however, as FORTRAN can use C's memory allocation routines through simple (yet clumsy looking) tricks [Hei91].

So there may be a slight loss in runtime efficiency when using C or C++ instead of FORTRAN. To minimize this runtime penalty, the most critical parts of the code should be implemented in the fastest available language. For an iterative solver, these are the inner loops identified in Section 6.11.1, plus possibly some loops in the functions setting up incomplete factorizations and data structures. For all other purposes, the existence of modern software engineering oriented concepts in C and especially in C++ improves readability of the code, simplifies debugging of complex data structures, improves robustness, and thus reduces development time. Among all the available languages, C++ optimizes one particular resource: programmer efficiency.

6.11.3 Self-restraint

With the current spread of UNIX-like operating systems, using OS-specific primitives for file access and resource monitoring does not hinder portability.

Still, minor differences in these accesses may cause trouble when porting. A very conservative and restrictive attitude is advisable, limiting OS contact to the strict minimum.

Similar comments apply to the use of certain language constructs. Compiler-specific features, as attractive as they may seem, should absolutely be avoided. One should also refrain from using peculiarities whose specification in the standard might be ambiguous.

6.12 The PILS package

A package of iterative linear solvers, called PILS, was developed along with this thesis[33]. PILS features practically all the variants of iterative solvers presented in this document.

6.12.1 Features

In particular, PILS implements the CG, CGNR, GCR(ℓ), Orthomin(ℓ), GMRES(ℓ), BiCG, CGS, Bi-CGSTAB, and BiCGStab2 methods. The available preconditioners are Jacobi, SSOR, D-ILU, ILU, ILU(k) (positional dropping based on the elimination tree), ND(τ) (numerical dropping), nested iterative solvers, external preconditioning, or no preconditioner at all. Preconditioners can be in right or split position. The iterative methods are all formulated in a way that allows the use of the Eisenstat trick to speed up preconditioned matrix-vector products on SSOR and D-ILU preconditioners. Convergence criteria based on the residual and based on a solution error estimate are available. Breakdowns are fixed by restarting the iterative method. Several monitoring options can be switched on and off. Inside an application, selected linear systems (including matrix, right-hand side, initial guess, and final approximation) can be saved to disk, for later experiments with the `pils` stand-alone driver.

The unknowns are reordered by greedy coloring to parallelize and vectorize the no-fill incomplete factorization preconditioners. To reduce the full fill before the numerical dropping factorization, the unknowns can also be

---

[33] *Pils* happens to also be the name of a low fermenting, strongly hopped beer originating from Plzeň in western Bohemia, Czechoslovakia. Rumors that it took many a *Pils* to write PILS remain unconfirmed.

reordered by the fill-reducing ordering techniques known as reverse Cuthill-McKee [GL81], nested dissection [GL81], and natural ordering by physical coordinates [HPWF91].

Any combination of the options above (including all parameters for methods, preconditioners, estimators, termination control mechanisms) forms a PILS **solver type**. Nested preconditioning can use any solver type as inner preconditioner, including recursively nested solvers. Whenever a solver type fails to fulfill its convergence criterion after its maximum number of iterations, PILS activates its ersatz solver type (which can specify abandon). Several different solver types can be tried (and chained, by using the final approximation of a failing type as initial guess for the next ersatz type).

All specifications to PILS, including its internal defaults, instructions by the client application, and instructions to the `pils` driver, are given in an interpreted specification language. Users of the client application can bypass the applications choices, even at runtime (that is, while the application is running and has called PILS already).

6.12.2 Implementation

The main data structure in PILS stores the lower and the upper triangle of a sparse matrix by color blocks, each block using jagged diagonals that form a partitioning by matchings. For matrices in other data structures (compressed rows, structurally symmetrized compressed rows [BCD$^+$89], blocked and masked connection lists), PILS constructs once a mapping based on the sparsity structure, and uses it to convert quickly all matrices with the same sparsity structure appearing in later solver calls.

PILS runs currently on Sun-3, Sun-4, DECsystem 5500, IBM RS/6000, Trace Multiflow, HP 9000/700, Sequent Symmetry S81, Silicon Graphics, Alliant FX/80, Alliant FX/2800, Convex C-200, Cray-2, Cray X/MP, Cray Y/MP, NEC SX-3. It uses vectorization and parallelization features where available.

The implementation of PILS currently[34] consists of 21,500 lines of code. Table 6.4 shows which languages were used. Although most of the PILS code is written in C++, PILS spends most of its execution time in optimized FORTRAN code.

---

[34] PILS Revision 1.27

programming language	*percentage of* PILS *code*
C++	88.3
FORTRAN	6.4
YACC	2.0
LEX	1.5
MAKE	1.3
SED	0.2
PILS specification language	0.2

Table 6.4: *Programming languages used in* PILS, *and their contribution to the total code size.*

6.12.3 Use

PILS is integrated into the 2-D device simulator GENSIM [Bür90], the 3-D device simulator SECOND [Hei91], and the multi-dimensional device simulator SIMUL [KMFW91], and has been released as a stand-alone package to several institutions. Inside those simulators, it has been in daily production runs at our Laboratory and at over a dozen other sites, over a period of one and a half years.

All the experiments for this thesis were done with PILS, on linear systems PILS extracted from these device simulators. Algorithmic variants were selected by setting the appropriate options in the `pils` driver.

A few more specific details on PILS, along with extracts of this thesis, can be found in [PF91b, PF92].

6.12.4 Drawbacks

Since the rest of Section 6.12 highlights the strong points of PILS, its weaknesses should be mentioned as well. This section lists a few such imperfections, and discusses their reasons, which are mostly due to the evolutionary development of the package.

Although PILS has been tested, used, and experimented with extensively on real-world device simulation problems, only a few experiments have been done with problems from some other domains, like constructed ill-conditioned model problems, circuit simulation, and surface parametrization, and other applications have been ignored completely. While this is better than working only with toy or model problems, or reusing over and over again the same test suite, some conclusions may potentially be restricted to device simulation.

At the time the development of PILS started, in early 1990, Bi-CGSTAB had not yet been invented, CGS was suspicious due to its erratic convergence behavior [Pin90, HPWF91], and GMRES-like methods were already known to be weak on ill-conditioned problems when restarted or truncated [BCD$^+$89]. BiCG was considered the best iterative method for device simulation. Consequently, transposed matrix-vector products had to be as efficient as possible, and this led to the sparse matrix data structure based on partitioning by matchings. To exploit potential structural symmetry (which exists for one-variable discretizations, and on some blocks of the matrices in the coupled Newton solution), and since regular and transposed matrix-vector products are equally frequent in BiCG, the upper triangular part of the matrix was stored in row-oriented jagged diagonals, and the lower triangular in column-oriented jagged diagonals. This means now that all the jagged diagonals have to be partitionings by matchings, even if a method without transposed operations is used. This results in a storage and timing overhead of between one and ten percent (due to explicit zero fill in the matchings, see Section 6.7.3), and to an annoyingly slow data structure initialization (which takes as long as solving a few linear systems with this same sparsity structure, but is still negligible in general device simulation runs, where many more such linear systems have to be solved). In spite of the (more or less, see the next paragraph) object-oriented design, changing this fundamental data structure in PILS would require a coding effort of at least two months.

When investigating the availability of compilers in early 1990, we found that C++ was not yet wide spread enough to be used; in particular, it was not available on Cray machines. Even though the ANSI C standard [Ame90] was published in Spring 1990, a reasonable subset of this standard was supported by enough compilers in early 1990, so that ANSI C was chosen as the main implementation language. The switch to C++ took only a few days, in Spring 1991. Later additions and partial reorganizations used the features of C++, but, although I tried an object-oriented style in ANSI C as well, parts of the package are still inhomogeneous with respect to programming style.

The price of flexibility is code size. An application hardly uses more than two or three solver types, but has to link the full-blown PILS library.

PILS currently has no support for distributed memory. My experience with iterative solvers on a DMPP, described in other sections of this chapter, comes from a separate project [PAF92, AzBF+90]. With some effort of replacing the lower operations, it would be possible to port PILS to a DMPP. To be useful and efficient within an application, however, the client application would have to be designed with distributed-memory parallelism in mind, and a different interface between client and library would be needed.

6.13 Benchmarks

The performance of scientific code, especially on supercomputers, is usually measured in MFlops (millions of floating-point operations per second). Such numbers may be reasonable for problems where a given algorithm is known to be the "best" and "fastest", but even in that case, there is lots of room for cheating[35,36]. We have seen above that for irregular sparse matrix applications, the memory traffic is of the same order as the number of computation operations and should therefore not be ignored. More important, there is no single iterative solver that is the fastest for all problems (even not for all problems among our restricted class) and on all machines.

I use MFlops, anyway, in this section, and only in this section, and I will justify this choice soon.

6.13.1 Varying the iterative method

Consider again the linear system we used in Chapter 3 to compare the convergence behavior of the iterative methods. Table 6.5 shows the performance PILS obtains for this problem on a particular machine.

The higher MFlops rates obtained for the methods of the GMRES-family compared to the biorthogonalization methods is typical: The relative percentage of linear operations among all computations is lower for the

---

[35] It is not a secret that many compilers recognize constructs from the LINPACK benchmark and substitute tuned libraries.

[36] The standard algorithm for multiplying two $n \times n$ dense matrices requires $2n^3$ floating-point operations. Strassen's fast matrix multiplication algorithm is numerically less stable, but requires only $O(n^{2.8})$ operations [Str69, GvL83]. Dividing $2n^3$ by the time Strassen's algorithm took, MFlops rates exceeding the peak performance of the machine used have been presented [Mer91].

Method	Performance [MFlops]	Convergence [iterations]	Time [seconds]
CGNR	98	>400	-
Orthomin(10)	148	>400	-
GCR(10)	123	361	12.16
GMRES(10)	116	388	12.24
GMRES(∞)	188	89	5.15
BiCG	100	140	7.18
CGS	100	78	4.09
Bi-CGSTAB	103	65	3.43
BiCGStab2	110	65	3.53

Table 6.5: *Performance of different split D-ILU preconditioned methods on one processor of a Cray Y/MP.*

latter. Since sparse matrix-vector multiplications involve indirect memory addressing, these operations are slower. The important column in Table 6.5, however, is the last one, showing the time in seconds to solve the system. Note that on a similar machine without hardware gather/scatter support, GMRES(∞) would certainly converge in less time than Bi-CGSTAB (provided that enough memory is available to store the 89 direction vectors). See also Section 6.2.

6.13.2 Varying the preconditioner

Table 6.6 shows variations of the performance with different preconditioners, for the one combination of an iterative method, a problem, and a machine. This comparison is probably the most unfair in this section. The data structures in PILS are optimized for the ILU and D-ILU types of preconditioners, so these achieve the highest rates (note that, because of the Eisenstat trick [Eis81] used in D-ILU and SSOR, the relative contribution of sparse matrix operations is lower, so the MFlops rates are higher than for ILU preconditioning). Unpreconditioned and diagonally preconditioned solvers would be faster with a data structure targeted solely to matrix-vector multiplication. The preconditioners with fill-in run mostly in scalar mode, and this explains their poor performance on this rather well conditioned system. Because of the differences in the relative timings, the decision as to whether a fast vectorized (e.g., D-ILU) or a more robust (e.g., ND) preconditioner should be used to solve an ill-conditioned system may depend on the machine architecture.

Preconditioner	*Performance [MFlops]* C-220	Y/MP	*Number of iterations*	*Total time [seconds]* C-220	Y/MP
none	6.66	100	>400	-	-
Jacobi	6.70	102	153	114	7.45
SSOR	6.74	103	69	55	3.62
D-ILU	6.76	103	65	53	3.43
ILU	6.24	94	61	110	11.47
ILU(1)	2.84	15	63	277	53.96
ND(0.01)	2.78	15	29	164	45.35

Table 6.6: *Performance of Bi-CGSTAB with different preconditioners (in split position), on a Convex C-220 and on a Cray Y/MP (one processor used). Note that for technical reasons, the MFlops rates are measured only in the iteration phase, and not in the preconditioner set-up.*

6.13.3 Varying the machine

Table 6.7 reports the MFlops rates for one algorithm (split D-ILU preconditioned Bi-CGSTAB) when applied to the same set of problems on different machines (note that the algorithm was not able to *solve* all these systems to the desired accuracy in a reasonable number of iterations). The differences in the performance clearly reflect the classes these machines belong to: previous generation microprocessors, today's microprocessors, minisupers, and supercomputers.

The problem size (i.e., the number of unknowns) increases from left to right in Table 6.7. The performance does not always improve with increasing problem size; rather, the opposite more often the case. There are two reasons for this effect, and both are related to memory access speed.

First of all, the average number of nonzeros per row differs in the five examples. The two problems on the left of Table 6.7 come from 2-D simulations, the other three from 3-D simulations. Also, the first three problems involve one unknown per grid point, the two others three unknowns

Problem	lddh	uvdih	bp25e	dr15c	mct70c
Unknowns	2.7k	22k	26k	47k	210k
Nonzeros/row	6.9	6.5	9.1	21	20
Grid dimension	2-D	2-D	3-D	3-D	3-D
Sun-3/280	0.10	0.10			
Sequent S81 (i386+fpa)	0.20	0.18	0.18	0.16	
Sun SparcStation 1+	1.05	0.93	0.92		
DEC system 5500	1.98	1.46	1.33		
Sun SparcStation 2	1.87	1.48	1.47	1.28	
Sun-4/490	1.85	1.59	1.54	1.37	1.22
Alliant FX/80 (6 procs.)	5.09	3.65	3.55	3.46	2.53
Convex C-220	7.79	8.00	7.36	6.76	6.85
Cray-2	37.28	38.29	31.73	26.61	28.73
Cray Y/MP	105.42	126.83	114.50	103.55	106.18
NEC SX-3/22	156.53	197.43	142.03	136.49	

Table 6.7: *MFlops rates for split D-ILU preconditioned Bi-CGSTAB on different machines. Except for the Alliant, only one processor was used on each machine.*

per point. More nonzeros per row lead to a higher percentage of indirect memory accesses (inside the sparse matrix operations), which are generally slower than the direct (and even sequential) accesses in the rest of the operations.

For problems with approximately the same number of nonzeros per row ("lddh" and "uvdih", "dr15c" and "mct70c"), a positive effect of problem size on performance can be observed for the four machines at the bottom of the table. On other machines, the performance still decreases, and this is because the cache hit ratio decreases.

Table 6.7 lists sequential computers, shared-memory multiprocessors, and vector supercomputers. The reported MFlops rates are those obtained in real device simulations. Numbers from distributed-memory multicomputers are not included, because neither PILS nor any device simulator with the potentials of PILS' clients is running on any such machine. The DMPP parallelization techniques presented in this chapter could be added to PILS, but none of the state-of-the-art device simulators can be ported easily to such a machine.

Problem	n	d_{aver}	*Grid*	*Speedup*
BBIG	2069	6.8	2-D	45.0
LDD	2674	6.9	2-D	47.3
BIPOL4K	4913	12.0	3-D	42.6
DR15	15564	8.6	3-D	49.4
BIPOL20K	20412	12.3	3-D	51.9
DRAM0	22680	24.4	3-D	54.3
DRAM1	22680	6.6	3-D	54.5
BP25C	76926	20.1	3-D	43.8

Table 6.8: *Speedup obtained on a 64-processor DMPP.*

The DMPP parallelization techniques presented in this chapter are not fictitious, though. They were evaluated in a separate package and on a simulated DMPP. The matrices used are real irregularly structured matrices from 2-D and 3-D device simulation. The target machine was K2, a planned and partly realized 64-processor DMPP with low-latency and high-bandwidth communication in an 8 × 8 torus raw-data interconnection network [AFNV90, AzBF^{+}90]. K2 was simulated using the K9 simulator [BPA89]. Table 6.8 gives some of the results obtained with two-dimensional geometric mapping and balanced greedy coloring, which came out as the best heuristics for mapping and coloring. Since K2 was never built and only simulated results are available, I restricted Table 6.8 to speedups (over a single processor, using an AMD 29050 RISC processor model) rather than presenting timings and MFlops rates. A detailed discussion of the results can be found in [PAF92].

6.13.4 Varying the storage consumption

We saw in Section 4.7.4 how iterative methods preconditioned by the ND(τ) preconditioner are capable of solving ill-conditioned linear systems. Decreasing the drop tolerance parameter τ reduces the number of iterations, but increases the number of nonzero entries in the factors of the approximate factorization. The curve in the left bottom quadrant of Figure 6.20 shows again the trade-off between total solution time and storage requirements for different drop tolerances. This curve replicates Figure 4.4 on page 94, except that the horizontal axis of the plot does not represent relative fill, but total real memory use for the algorithm and the preconditioner, on a Convex-220.

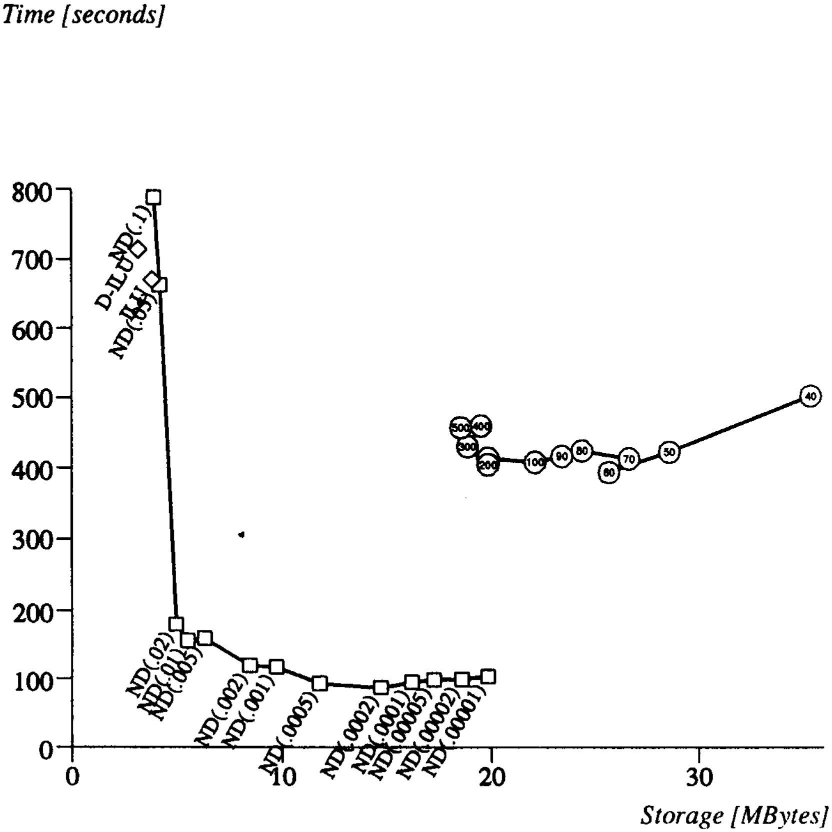


Figure 6.20: *Storage size and solution time for an ill-conditioned linear system on a Convex C-220.*

The nested solver analyzed in Section 4.7.5 showed a similar time-storage trade-off. Controlled by the maximum number of inner iterations, the number of outer iterations determines how many back vectors need to be stored for the outer GCR(∞) method. The curve in the upper right quadrant of Figure 6.20 replicates the lower (best) variant from Figure 4.5 on page 96, plotting total solution time as a function of total memory use on a Convex-220.

The relative locations of these two curves in Figure 6.20 indicate a defeat of the nested solvers agains the ND(τ) preconditioned solvers: nested solvers require both more storage and more time to solve this linear system. This assertion is valid for the Convex-220, but what about other machines ?

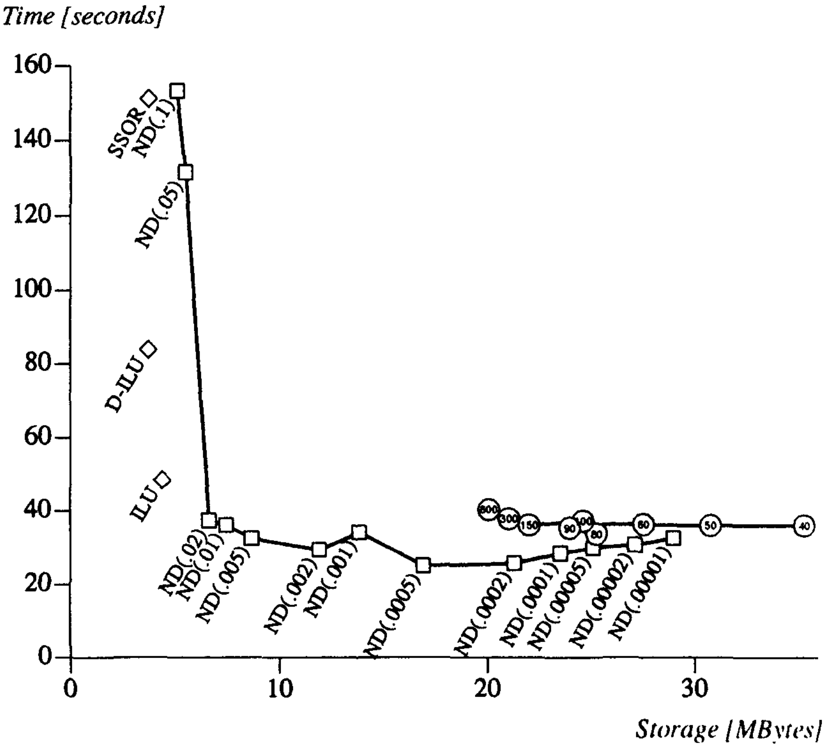


Figure 6.21: *Storage size and solution time for an ill-conditioned linear system on a Cray Y/MP.*

Figure 6.21 displays the results of exactly the same series of experiments, but on a Cray Y/MP. The distinction is much less pronounced, and this for two reasons:

1. The ND(τ) preconditioner runs mainly in scalar mode. The performance ratio between vector and scalar code is much higher on the Cray than on the Convex, so the ND(τ) preconditioned solvers are not as much faster than the others on the Cray.
2. ND(τ) uses a considerable amount of integer storage. One integer

> takes 8 bytes on a Cray, but only 4 bytes on a Convex. Therefore, the preconditioner requires almost twice as much memory on the Cray, even if the number of fill entries is the same.

Also, the shape of the curves is not exactly the same. This is due to the differences in the floating-point formats on the two machines, which lead to different rounding effects and sometimes ultimately to different iteration numbers (see Section 5.6).

7

Conclusions

7.1 Current status

As with many other applications in scientific computing, semiconductor device simulation requires repeated solution of large, sparse systems of linear equations. Local grid refinement leads to very irregular sparsity structures for the matrices. The nature of the partial differential equations leads to notoriously ill-conditioned linear systems.

Superlinearly growing storage requirements limit the range of application of sparse direct solvers for large, sparse systems of linear equations. Iterative linear solvers are a viable and efficient alternative. Their behavior on unsymmetric and ill-conditioned linear systems requires special attention.

An **iterative solver** consists of an iterative method, a preconditioner, and a termination control mechanism. All successful iterative methods proceed by finding in a sequence of Krylov subspaces the stationary point of a function whose single global stationary point coincides with the solution of the linear system.

GMRES needs the minimum number of matrix-vector products to minimize the residual in the Krylov subspace $\mathcal{K}_k(A, r_0)$, but can often not be used as is because of its storage requirements, which increase with the iteration number k. Truncated and restarted variants of GMRES have shown to be much less effective in device simulation applications.

BiCG and variants thereof are generally the most efficient and successful iterative methods for device simulation. Their theoretical possibility for breakdown occurs very rarely in practice, and can easily be fixed through restarting. The Bi-CGSTAB method has emerged as the leading method in this class.

The most effective **preconditioners** consist of incomplete LU-factorizations. D-ILU is a very fast incomplete factorization preconditioner that suits the needs of the great majority of linear systems in device simulation. Some very ill-conditioned linear systems can currently only be solved efficiently by an approximate factorization preconditioner based on numerical dropping. A flexible solver package adapts automatically to be most efficient on the majority of solver calls and still effective on the difficult cases.

The fact that the matrix is only used as an operator for matrix-vector multiplication appears to offer a high degree of parallelism in iterative solvers. This is only partly true. Linear operations, vector dot products, and matrix-vector products with regularly structured matrices are indeed easy to vectorize and parallelize. On sparse matrices with highly irregular structure, the available parallelism is more difficult to exploit; on vector computers through complicated data structures, and on distributed-memory multicomputers through heuristics that map problem locality onto interconnection network locality. Memory access speed and network latency largely dominate the performance on today's architectures.

The inherent parallelism in incomplete factorization preconditioners is insufficient. Reorderings based on graph coloring construct no-fill incomplete factorization preconditioners that vectorize and parallelize well. Factorizations with limited fill, however, are mostly sequential.

The relative performance of different iterative solvers varies with the architecture of the platform. Solvers for certain very ill-conditioned problems cannot exploit most of the high-performance architectural features. Distributed-memory multicomputers are efficient only if the message latency is very low, so that the time to transmit one or a few numbers is in the time range of only one or a few floating-point operations.

7.2 Future research

Bi-CGSTAB, QMR, and BiCGStab2 have all been published in the last two years; CGS has been known since 1984, but the first major journal publication appeared in 1989. With the quick success of these variants of the BiCG algorithm, some further developments in this direction may be expected. QMR is promising, but its parameter set needs to be tuned to cope with ill-conditioned systems, and a stable squared variant will have to come. The extra cost for the two-dimensional minimization hinders BiCGStab2 from having a true edge over Bi-CGSTAB. Rather than doing the two-dimensional step at every even iteration, a quantitative heuristic should be found to use two-dimensional minimization only at those iterations where it brings a (timing) advantage over two one-dimensional steps. Very little is known as yet about the influence of the dual initial residual vector $\hat{r}_0$ in BiCG and its variants.

Nested iterative solvers may be a more efficient way of reducing the memory requirements of GMRES-like methods, without loosing so many of the nice theoretical properties like with restarting or truncation. The huge parameter space for combining inner and outer solvers has to be explored in more detail.

Dropping criteria and dropping strategies for the numerical dropping factorization preconditioner need more theoretical analysis, in the hope of finding better criteria. While the current implementation delivers the fill after dropping as a function of the tolerance τ, the factorization process should adapt the tolerance to meet prescribed storage limits. A parallelizable or vectorizable variant of numerically controlled approximate factorization preconditioners is needed to keep them competitive even on architectures with a high parallelism gain.

Automatic switching between fast and robust iterative solvers should not only be possible a posteriori, after the faster solver has run for a given number of iterations, but be controlled by a fast a priori criterion.

The construction of data structures for highly parallel operations is relatively expensive. It is presented in this thesis under the assumptions that many linear systems with the same sparsity structure are solved, so that the overhead for constructing parallelism is recovered after several numerical solver calls. It will need some tuning to cope with applications where the discretization grid changes frequently, e.g., with adaptive refinement.

7.3 Future visions

The black box solver, a function that solves any linear system exactly and in minimum time, is what users really want. Such a situation exists today for relatively small dense direct solvers. For those classes of linear systems where iterative solvers are the most efficient algorithms, such an ideal and universal black box will probably stay an illusion.

Automatically adapting methods and preconditioners may provide an acceptable approximation to such a black box solver. Switching to a more expensive but stabler preconditioner if another iterative solver does not converge in a limited number of iterations is only a very rough first step of adaptation. The two ideas—preconditioners for which set-up time and quality increase monotonically with a parameter (like a drop tolerance), and iterative methods which can deal with preconditioners that vary from one iteration to another (like nested iterative solvers)—could be combined in a more smoothly adaptively improving preconditioned iterative solver.

Massive parallelism is trivial to obtain in diagonally preconditioned iterative methods on regular tensor-product grids. Short-circuiting to the conclusion that iterative solvers in general map perfectly to machines with many processors is a dangerous fallacy. Higher quality preconditioners and irregular structures require a significant amount of implementation work in the entire application code. Communication is finely dispersed, and not network bandwidth on large data exchanges, but message latency when sending a single or a few data items is the crucial parameter that decides whether a distributed-memory multicomputer can be successfully used for iterative solvers.

Since the relative performance of different iterative solvers varies strongly with the machine architecture, the idea of automatic adaptation should be carried to machine dependence. On-the-fly timings can determine the relative efficiency of different variants.

As an example for a combination of all the ideas in this section, we could think of an asynchronously improving preconditioner. At the black box solver call, two logical tasks start: the first applies an iterative method with a cheap, fast preconditioner, the second computes a more complicated preconditioner. When the better preconditioner is ready, the iterative method, which may already have performed a number of iterations with the cheaper preconditioner, starts using the better preconditioner. The second task then starts constructing an even better, but even more expensive preconditioner. A one-processor implementation of this idea could fork into two processes under quasi-concurrent control of the multitasking operating system. A multiprocessor would allocate a certain number of processors to either of the two tasks. The relative priority or the percentage of allocated processors could favor the first task at the beginning and progressively increase the resources allocated to the second task. While a uniprocessor would switch to the better preconditioner after fifty iterations with the fast preconditioner, a highly parallel machine might do this switch only after several hundred iterations. Assuming that future parallel machines support this efficiently, a better multiprocessor implementation could be to fork into two tasks running on all processors, and the first task would run in the spare time that the second (less parallel and requiring more data transfer) task waits for incoming data.

List of Figures

Leer - Vide - Empty

List of Algorithms

List of Tables

Leer - Vide - Empty

Bibliography

[AFNV90] M. Annaratone, M. Fillo, K. Nakabayashi, and M. Viredaz. The K2 parallel processor: Architecture and hardware implementation. In *Proc. 17th Symposium on Computer Architecture*, Seattle, June 1990. IEEE-ACM.

[AH76] K. Appel and W. Haken. Every planar map is four colourable. *Bulletin of the American Mathematical Society*, 82:711–712, 1976.

[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1983.

[AJ84] M. A. Ajiz and A. Jennings. A robust incomplete Cholesky-conjugate gradient algorithm. *Journal for Numerical Methods in Engineering*, 20:949–966, 1984.

[AJ89] G. Alaghband and H. F. Jordan. Sparse Gaussian elimination with controlled fill-in on a shared memory multiprocessor. *IEEE Transactions on Computers*, 38(11):1539–1557, November 1989.

[Ame90] American National Standards Institute, 1430 Broadway, New York, NY 10018. *American National Standard for Information Systems - Programming Language - C*, 1990. ANSI Standard X3.159-1989.

[AMS90] S. F. Ashby, T. A. Manteuffel, and P. E. Saylor. A taxonomy for conjugate gradient methods. *SIAM J. Numer. Anal.*, 27(6):1542–1568, December 1990.

[AÖ87] C. Aykanat and F. Özgüner. Large grain parallel conjugate gradient algorithms on a hypercube multiprocessor. In S. K. Sahni, editor, *Proceedings of the 1987 International Conference on Parallel Processing*, pages 641–644. Pennsylvania State University Press, August 1987.

[AÖES88] C. Aykanat, F. Özgüner, F. Ercal, and P. Sadayappan. Iterative algorithms for solution of large sparse systems of linear equations on hypercubes. *IEEE Transactions on Computers*, C-37(12):1554–1568, December 1988.

[APR89] M. Annaratone, C. Pommerell, and R. Rühl. Interprocessor communication speed and performance in distributed-memory parallel processors. In *Proc. 16th Symposium on Computer Architecture*, pages 315–324, Jerusalem, June 1989. IEEE-ACM.

[AS90] S. F. Ashby and M. K. Seager. A proposed standard for iterative linear solvers. Technical report, Lawrence Livermore National Laboratory, January 1990.

[Avr76] M. Avriel. *Nonlinear Programming: Analysis and Methods.* Prentice-Hall, 1976.

[AzBF$^+$90] M. Annaratone, G. zur Bonsen, M. Fillo, K. Nakabayashi, C. Pommerell, R. Rühl, P. Steiner, and M. Viredaz. Ein Parallel-Computer mit verteiltem Speicher: Das K2-Projekt. *Bulletin SEV/VSE*, 81(17):11–18, August 1990.

[BCC$^+$88] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An integrated solution to high-speed parallel computing. In *Supercomputing '88*, Kissimmee, FL, November 1988.

[BCCS89] R. E. Bank, T. F. Chan, W. M. Coughran, Jr., and R. K. Smith. The alternate-block-factorization procedure for systems of partial differential equations. *BIT*, 29:938–954, 1989.

[BCD$^+$89] R. E. Bank, W. M. Coughran, Jr., M. A. Driscoll, R. K. Smith, and W. Fichtner. Iterative methods in semiconductor device simulation. *Computer Physics Communications*, 53:201–212, 1989.

[BCF$^+$85] R. E. Bank, W. M. Coughran, Jr., W. Fichtner, E. H. Grosse, D. J. Rose, and R. K. Smith. Transient simulation of silicon devices and circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, CAD-4:436–451, June 1985.

[BDMN79] G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Chartwell-Bratt, Kent, 1979.

[BPA89] P. Beadle, C. Pommerell, and M. Annaratone. K9: A simulator of distributed-memory parallel processors. In *Supercomputing '89*, pages 765–774, Reno, NV, November 1989. ACM-IEEE.

[BR81] R. E. Bank and D. J. Rose. Global approximate Newton methods. *Numer. Math.*, 37:279–295, 1981.

[BR90] R. E. Bank and D. J. Rose. On the complexity of sparse Gaussian elimination via bordering. *SIAM J. Sci. Stat. Comput.*, 11(1):145–160, January 1990.

[BRF83] R. E. Bank, D. J. Rose, and W. Fichtner. Numerical methods for semiconductor device simulation. *SIAM J. Sci. and Stat. Comput*, 4:416–35, 1983.

[BS87] R. E. Bank and R. K. Smith. General sparse elimination requires no permanent integer storage. *SIAM J. Sci. Stat. Comput.*, 8(4):574–584, July 1987.

[BS91] C. Brezinski and H. Sadok. Avoiding breakdown in the CGS algorithm. Publication ANO-242, Université des Sciences et Techniques de Lille Flandres-Artois, Laboratoire d'Analyse Numérique et d'Optimisation, 59655 Villeneuve D'Ascq, Cedex, France, Mars 1991.

[Bür90] J. F. Bürgler. *Discretization and Grid Adaptation in Semiconductor Device Modeling*. PhD thesis, ETH-Zürich, 1990. publ. by Hartung-Gorre Verlag, Konstanz, Germany.

[BZS91] C. Brezinski, M. R. Zaglia, and H. Sadok. A breakdown-free Lanczos' type algorithm for solving linear systems. Publication ANO-239, Université des Sciences et Techniques de Lille Flandres-Artois, Laboratoire d'Analyse Numérique et d'Optimisation, 59655 Villeneuve D'Ascq, Cedex, France, Janvier 1991.

[CdPvdV91] T. F. Chan, L. de Pillis, and H. A. van der Vorst. A transpose-free squared Lanczos algorithm and application to solving nonsymmetric linear systems. Technical report, University of California at Los Angeles, 1991.

[CG89] A. T. Chronopoulos and C. W. Gear. s-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25:153–168, 1989.

[CGO76] P. Concus, G. H. Golub, and D. P. O'Leary. A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations. In J. R. Bunch and D. J. Rose, editors, *Sparse Matrix Computations*, pages 309–332. Academic Press, New York, 1976.

[CHF91] P. Conti, N. Hitschfeld, and W. Fichtner. Ω - an octree-based mixed element grid allocator for adaptive 3d device simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 10(10):1231–1241, October 1991.

[Con91] P. Conti. *Grid Generation for Three-dimensional Semiconductor Device Simulation.* PhD thesis, ETH-Zürich, 1991. publ. by Hartung-Gorre Verlag, Konstanz, Germany.

[Dal90] W. J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, C-39(6):775–785, June 1990.

[DDSvdV91] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers.* SIAM, 1991.

[DER86] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, London, England, 1986.

[Deu90] P. Deuflhard. Global inexact Newton methods for very large scale nonlinear problems. Preprint SC 90-2, Konrad-Zuse-Zentrum für Informationstechnik, Berlin, 1990.

[DFW90] P. Deuflhard, R. Freund, and A. Walter. Fast secant methods for the iterative solution of large nonsymmetric linear systems. *Impact of Computing in Science and Engineering*, 2:244–276, 1990.

[DGL91] D. S. Dodson, R. G. Grimes, and J. G. Lewis. Sparse extensions to the FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 17(2):253–263, June 1991.

[dH86] C. den Heijer. Preconditioned iterative methods for nonsymmetric linear systems. In K. Board and D. R. J. Owen, editors, *Simulation of Semiconductor Devices and Processes, Volume 2, SISDEP II Proceedings*, pages 267–285, Swansea, United Kingdom, July 1986. Pineridge Press.

[DvdV91] M. Driessen and H. A. van der Vorst. BI-CGSTAB in semiconductor modelling. In W. Fichtner and D. Aemmer, editors, *Simulation of Semiconductor Devices and Processes Vol. 4*, pages 45–54. Hartung-Gorre Verlag, September 1991.

[Ede91] A. Edelman. The first annual large dense linear system survey. *ACM SIGNUM Newsletter*, 26(4):6–12, October 1991.

[EGRS59] M. E. Engeli, T. Ginsburg, H. Rutishauser, and E. Stiefel. *Refined iterative methods for computations of the solution and the eigenvalues of self-adjoint boundary value problems*. Mitt. Nr. 8, Inst. angew. Math. ETH Zürich. Birkhaeuser, 1959.

[Eis81] S. C. Eisenstat. Efficient implementation of a class of preconditioned conjugate gradient methods. *SIAM J. Sci. Stat. Comput.*, 2(1):1–4, March 1981.

[Elm82] H. C. Elman. *Iterative Methods for Large, Sparse Nonsymmetric Systems of Linear Equations*. PhD thesis, Yale University Department of Computer Science, April 1982.

[FGN90] R. W. Freund, M. H. Gutknecht, and N. Nachtigal. An implementation of the look-ahead Lanczos algorithm for non-Hermitian matrices Part I. Technical Report 90.45, RIACS, NASA Ames Research Center, November 1990.

[FGN92] R. W. Freund, G. H. Golub, and N. Nachtigal. Iterative solution of linear systems. *Acta Numerica*, 1, 1992.

[Fle76] R. Fletcher. Conjugate gradient methods for indefinite systems. In G. A. Watson, editor, *Proc. of the 6-th Biennal Dundee Conference on Numerical Analysis*. Springer-Verlag, 1976.

[FM84] V. Faber and T. Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM J. Numer. Anal.*, 21:352–362, 1984.

[FN91] R. W. Freund and N. M. Nachtigal. QMR: a quasi-minimal residual method for non-Hermitian linear systems. *Numer. Math.*, 60:315–339, 1991.

[FRB83] W. Fichtner, D. J. Rose, and R. E. Bank. Semiconductor device simulation. *SIAM J. Sci. and Stat. Comput*, 4:391–415, 1983.

[Fre91] R. W. Freund. A transpose-free quasi-minimal residual algorithm for non-hermitian linear systems. Technical Report 91.18, RIACS, NASA Ames Research Center, September 1991.

[GL81] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1981.

[GL89] A. George and J. W. H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, March 1989.

[GMW81] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization.* Academic Press, 1981.

[GO89] G. H. Golub and D. P. O'Leary. Some history of the conjugate gradient and Lanczos algorithms: 1948-1976. *SIAM Review*, 31(1):50–102, March 1989.

[GSZ90] K. Gallivan, A. Sameh, and Z. Zlatev. Solving general sparse linear systems using conjugate gradient-type methods. In *1990 International Conference on Supercomputing*, pages 132–139, Amsterdam, June 1990. ACM.

[Gut90a] M. H. Gutknecht. A completed theory of the unsymmetric Lanczos process and related algorithms Part II. IPS Research Report 90-16, Interdisciplinary Project Center for Supercomputing, ETH Zürich, September 1990. submitted to SIAM J. Matrix Anal. Appl.

[Gut90b] M. H. Gutknecht. The unsymmetric Lanczos algorithms and their relations to Padé approximation, continued fractions, and the qd algorithm. In *Copper Mountain Conference on Iterative Methods.* SIAM, April 1990.

[Gut91] M. H. Gutknecht. Variants of BiCGStab for matrices with complex spectrum. IPS Research Report 91-14, Interdisciplinary Project Center for Supercomputing, ETH Zürich, August 1991.

[Gut92] M. H. Gutknecht. A completed theory of the unsymmetric Lanczos process and related algorithms Part I. *SIAM J. Matrix Anal. Appl.*, 13(2):594–639, June 1992.

[GvL83] G. H. Golub and C. F. van Loan. *Matrix Computations*. The John Hopkins University Press, Baltimore and London, 1983.

[Hac91] W. Hackbusch. *Iterative Lösung großer schwachbesetzter Gleichungssysteme*. Teubner, Stuttgart, 1991.

[Har72] F. Harary. *Graph Theory*. Addison-Wesley, Reading, Massachusetts, 1972.

[Hei91] G. Heiser. *Design and Implementation of a Three-Dimensional, General Purpose Semiconductor Device Simulator*. PhD thesis, ETH-Zürich, 1991. publ. by Hartung-Gorre Verlag, Konstanz, Germany.

[Her91] M. A. Heroux. A reverse communication interface for "matrix-free" preconditioned iterative solvers. Technical report, Mathematical Software Research Group, Cray Research, Inc., Eagan, MN 55121, 1991.

[HJ85] R. A. Horn and C. R. Johnson. *Matrix Analysis*. Cambridge University Press, 1985.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[HPWF91] G. Heiser, C. Pommerell, J. Weis, and W. Fichtner. Three dimensional numerical semiconductor device simulation: Algorithms, architectures, results. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 10(10):1218–1230, October 1991.

[HS52] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand.*, 49:409–436, 1952.

[HSST92] O. Heinreichsberger, S. Selberherr, M. Stiftinger, and K. P. Traar. Fast iterative solution of carrier continuity equations for three-dimensional device simulation. *SIAM J. Sci. Stat. Comput.*, 13(1), 1992.

[HY81] L. A. Hageman and D. M. Young. *Applied Iterative Methods.* Computer Science and Applied Mathematics. Academic Press, New York, 1981.

[JC91] W. D. Joubert and G. F. Carey. Parallelizable restarted iterative methods for nonsymmetric linear systems. Technical Report CNA-251, Center for Numerical Analysis, University of Texas at Austin, May 1991. to appear in International Journal of Computer Mathematics.

[Jou90] W. Joubert. Lanczos methods for the solution of nonsymmetric systems of linear equations. In *Copper Mountain Conference on Iterative Methods.* SIAM, April 1990.

[KMFW91] K. Kells, S. Müller, W. Fichtner, and G. Wachutka. Simulating temperature effects in multi-dimensional silicon devices with generalized boundary conditions. In W. Fichtner and A. Aemmer, editors, *Simulation of Semiconductor Devices and Processes IV*, pages 141–148. Hartung-Gorre Konstanz, 1991.

[KNY91] L. Y. Kolotilina, A. A. Nikishin, and A. Y. Yeremin. Factorized sparse approximate inverse (FSAI) preconditionings for solving 3d FE systems on massively parallel computers. II. Iterative construction of FSAI preconditioners. In *IMACS International Symposium on Iterative Methods in Linear Algebra*, pages 161–162, Brussels, April 1991. Preliminary Proceedings.

[KOY89] D. R. Kincaid, T. C. Oppe, and D. M. Young. ITPACKV 2D user's guide. Technical Report CNA-232, Center for Numerical Analysis, University of Texas at Austin, May 1989.

[KRYG82] D. R. Kincaid, J. R. Respess, D. M. Young, and R. G. Grimes. ITPACK 2C: A FORTRAN package for solving large sparse linear systems by adaptive accelerated iterative methods. *ACM Transactions on Mathematical Software*, 8(3):302–322, September 1982.

[Kui87] L. K. Kuiper. A comparision of iterative methods as applied to the solution of the nonlinear three-dimensional groundwater flow equation. *SIAM J. Sci. Stat. Comput.*, 8(4):521–528, July 1987.

[Lan50] C. Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *J. Res. Nat. Bur. Stand.*, 45:255–282, 1950.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[Liu90] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.*, 11(1):134–172, January 1990.

[LMSS91] J. Loeliger, R. Metzger, M. Seligman, and S. Stroud. Pointer target tracking - an empirical study. In *Supercomputing '91*, pages 14–23, Albuquerque, NM, November 1991. ACM-IEEE.

[Man79] T. A. Manteuffel. Shifted incomplete Cholesky factorization. In I. S. Duff and G. W. Stewart, editors, *Sparse Matrix Proceedings 1978*, pages 41–61, Philadelphia, PA, 1979. SIAM Publications.

[Mer91] M. Merchant. BLAS routines (single CPU). Paris Seminar on SCILIB, Cray Research Inc., April 12 1991.

[Mey88] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[MKF91] S. Müller, K. Kells, and W. Fichtner. Automatic rectangle-based adaptive mesh generation without obtuse angles. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 1991.

[MR88] R. G. Melhem and K. V. S. Ramarao. Multicolor reordering of sparse matrices resulting from irregular grids. *ACM Transactions on Mathematical Software*, 14(2):117–138, June 1988.

[MvdV77] 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. of Comput.*, 31(137):148–162, January 1977.

[MvdV87] O. A. McBryan and E. F. van de Velde. Hypercube algorithms and implementations. *SIAM J. Sci. Stat. Comput.*, 8(2):s227–s287, March 1987.

[NRT90] N. M. Nachtigal, S. C. Reddy, and L. N. Trefethen. How fast are nonsymmetric matrix iterations? In *Copper Mountain Conference on Iterative Methods*. SIAM, April 1990.

[OK90] T. C. Oppe and D. R. Kincaid. Are there iterative BLAS? In *Copper Mountain Conference on Iterative Methods*. SIAM, April 1990.

[Ort88] J. M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems*. Plenum Press, New York, 1988.

[PAF92] C. Pommerell, M. Annaratone, and W. Fichtner. A set of new mapping and coloring heuristics for distributed-memory parallel processors. *SIAM J. Sci. Stat. Comput.*, 13(1):194–226, January 1992.

[PF91a] C. Pommerell and W. Fichtner. New developments in iterative methods for device simulation. In W. Fichtner and A. Aemmer, editors, *Simulation of Semiconductor Devices and Processes IV*, pages 243–248. Hartung-Gorre Konstanz, 1991.

[PF91b] C. Pommerell and W. Fichtner. PILS: An iterative linear solver package for ill-conditioned systems. In *Supercomputing '91*, pages 588–599, Albuquerque, NM, November 1991. ACM-IEEE.

[PF92] C. Pommerell and W. Fichtner. Memory aspects and performance of iterative solvers. In *Copper Mountain Conference on Iterative Methods*. SIAM, April 1992.

[Pin90] M. R. Pinto. *Comprehensive Semiconductor Device Simulation for Silicon ULSI*. PhD thesis, Stanford, 1990.

[Pis84] S. Pissanetzky. *Sparse Matrix Technology*. Academic Press, Orlando, Florida 32887, 1984.

[PS82a] C. C. Paige and M. A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Transactions on Mathematical Software*, 8(1):43–71, March 1982.

[PS82b] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[Roc70] R. T. Rockafellar. *Convex Analysis*. Princeton University Press, NJ, 1970.

[RSAR91] R. Rios, R. K. Smeltzer, R. Amantea, and A. Rothwarf. A three-dimensional device simulator for radiation-hard MOS-SOS transistors. *Solid-State Electronics*, 34(8):853–859, August 1991.

[Saa82] Y. Saad. The Lanczos biorthogonalization algorithm and other oblique projection methods for solving large unsymmetric systems. *SIAM J. Numer. Anal.*, 19(3):485–506, June 1982.

[Saa88] Y. Saad. Preconditioning techniques for nonsymmetric and indefinite linear systems. *Journal of Computational and Applied Mathematics*, 24:89–105, 1988.

[Saa89] Y. Saad. Krylov subspace methods on supercomputers. *SIAM J. Sci. Stat. Comput.*, 10(6):1200–1232, November 1989.

[Saa90] Y. Saad. Sparskit: a basic tool kit for sparse matrix computations. Technical Report 90.20, RIACS, NASA Ames Research Center, May 1990.

[SBD$^+$76] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines - EISPACK Guide*. Springer-Verlag, 2nd edition, 1976.

[Sed88] R. Sedgewick. *Algorithms*. Addison-Wesley, 2nd edition edition, 1988.

[Sel84] S. Selberherr. *Analysis and Simulation of Semiconductor Devices*. Springer-Verlag, 1984.

[Set89] R. Sethi. *Programming Languages Concepts and Constructs*. Addison-Wesley, Reading, MA, 1989.

[SF73] G. Strang and G. J. Fix. *An Analysis of the Finite Element Method.* Prentice-Hall, 1973.

[Smi78] G. D. Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods.* Oxford University Press, 1978.

[Son89] P. Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 10(1):36–52, January 1989.

[SP88] A. Silberschatz and J. L. Peterson. *Operating System Concepts.* Addison-Wesley, Reading, Mass., alternate edition, 1988.

[SS86] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, July 1986.

[Str69] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[Tre] L. N. Trefethen. Non-normal operators and pseudospectra. In preparation. A summary is available by anonymous ftp from ftp.cs.cornell.edu as file /pub/trefethen/pseudo.tex.

[TRM91] V. E. Taylor, A. Ranade, and D. G. Messerschmitt. Three-dimensional finite-element analyses: Implications for computer architectures. In *Supercomputing '91*, pages 786–795, Albuquerque, NM, November 1991. ACM-IEEE.

[Var62] R. S. Varga. *Matrix Iterative Analysis.* Prentice-Hall, Englewood Cliffs, 1962.

[vdV89] H. A. van der Vorst. The convergence behavior of some iterative solution methods. Report 89-19, Faculty of Technical Mathematics and Informatics, Delft University of Technology, 1989. ISSN 0922-5641.

[vdV92] H. A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13(2):631–644, March 1992.

[vdVV91] H. A. van der Vorst and C. Vuik. GMRESR: a family of nested GMRES methods. Technical Report 91-80, Dept. of Techn. Math. and Inf., TU-Delft, 1991.

[Vin76] P. K. W. Vinsome. ORTHOMIN – an iterative method for solving sparse sets of simultaneous linear equations. In *Proc. Fourth SPE Symposium on Reservoir Simulation*, pages 149–160, Los Angeles, February 1976. Society of Petroleum Engineers. Paper SPE 5739.

[Vit88] P. M. B. Vitányi. Locality, communication, and interconnect length in multicomputers. *SIAM Journal on Computing*, 17(4):659–672, August 1988.

[vR50] W. van Roosbroeck. Theory of flow of electrons and holes in germanium and other semiconductors. *Bell System Tech. J.*, 29:560–607, 1950.

[Vui92] C. Vuik. Further experiences with GMRESR. In *Copper Mountain Conference on Iterative Methods*. SIAM, April 1992.

[VvdV92] C. Vuik and H. A. van der Vorst. A comparison of some GMRES-like methods. *Linear Algebra and Applications*, 160, January 1992.

[Wal88] H. F. Walker. Implementation of the GMRES method using Householder transformations. *SIAM J. Sci. Stat. Comput.*, 9(1):152–163, January 1988.

[YJ80] D. M. Young and K. C. Jea. Generalized conjugate-gradient acceleration of nonsymmetric iterative methods. *Linear Algebra and its Applications*, 34:159–194, 1980.

[YMJ$^+$89] D. P. Young, R. G. Melvin, F. T. Johnson, J. E. Bussoletti, L. B. Wigton, and S. S. Samanth. Application of sparse matrix solvers as effective preconditioners. *SIAM J. Sci. Stat. Comput.*, 10(6):1186–1199, November 1989.

Leer - Vide - Empty

Curriculum Vitae

I was born in Luxembourg on June 30, 1964, and I am a citizen of the Grand-Duchy of Luxembourg. After finishing high school at the *Athénée de Luxembourg* in 1983, I enrolled in the Computer Science faculty of the Swiss Federal Institute of Technology *(ETH)* in Zurich. I received an M.Sc. in Computer Science Engineering *(Dipl. Informatik-Ing. ETH)* in 1988. I joined the Integrated Systems Laboratory of ETH in April 1988 as a research assistant, where I worked first in the K2 distributed-memory parallel processor project, and since November 1989 in the semiconductor device simulation group.