Universität Karlsruhe
Rechenzentrum
Postfach 6980
76128 Karlsruhe
Germany

Interner Bericht Nr. 61/95

LINSOL
(LINear SOLver)
Description and User's Guide for the parallelized version

D R A F T
Version 0.96

Abteilung "Numerikforschung für Supercomputer"
Rechenzentrum der Universität Karlsruhe
December 1995

Authors: Rüdiger Weiss, Hartmut Häfner and Willi Schönauer

Contents

Chapter 1

Introduction

Many technical and scientific problems can be described by systems of partial differential equations. These equations are usually linearized and discretized by finite differences [91, 92, 93, 102], finite elements [46, 47, 111] or boundary elements. For all these methods a linear system has to be solved as an inner kernel. The solution of linear systems is also required for applications that do not arise from partial differential equations, for example for the simulation of electric networks or for the design of computer chips.

Direct iterative solvers applied to sparse matrices, resulting from the finite difference and finite element method, produce fill-in, so that even the memory capabilities of supercomputers are insufficient. Therefore, iterative solvers have to be applied. For full matrices, resulting for example from the boundary element method, iterative techniques may be much faster than direct methods if a few digits of accuracy are sufficient – as typical for engineering applications [129]. The dimension of the systems may be extremely large for fine discretizations. The necessary memory and computing power can only be provided by massively parallel computers with distributed memory.

In order to provide general applicable iterative linear solvers for the community of scientific computing the LINSOL program package has been designed. The focus of this package is on efficient implementation on massively parallel systems, portability and robustness.

LINSOL uses state-of-the-art solvers as basic methods. Only a few preconditioners are up to now implemented, but they will be complemented in near future. LINSOL is tuned to massively parallel systems with distributed memory. Therefore, the message passing programming style is used. LINSOL supports many matrix formats for the convenience of the users. Moreover, adaptive method selection schemes called polyalgorithms are implemented.

Chapter 2

Iterative Methods

Our problem is to solve the linear system

$$Ax = b. \tag{2.1}$$

The matrix

$$A = \begin{pmatrix} a_{1,1} & \dots & a_{1,n} \\ \vdots & & \vdots \\ a_{n,1} & \dots & a_{n,n} \end{pmatrix} \in \mathbb{R}^{n \times n},$$

is a real, square matrix of dimension n and $x,\ b \in \mathbb{R}^n$. Though many of the presented methods work also for singular matrices we assume that A is not singular in order to get a unique solution and unique error estimates.

2.1 Iterates, Residuals and Errors

Starting from an initial guess x_0 an iterative solution method constructs a sequence of approximations

$$x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4 \rightarrow \dots$$

so that

$$x_k \rightarrow x \tag{2.2}$$

for $k \rightarrow \infty$. The recurrence can be very sophisticated and non-linear. The residuals are defined by

$$r_k = Ax_k - b\ , \tag{2.3}$$

and the errors by

$$e_k = x_k - x \ . \tag{2.4}$$

The residuals and errors are connected by

$$r_k = Ae_k, \tag{2.5}$$

because

$$r_k = Ax_k - b = A(x_k - A^{-1}b) = A(x_k - x) = Ae_k.$$

As a result of $||r_k|| = ||Ae_k|| \leq ||A|| \cdot ||e_k||$ and $||e_k|| = ||A^{-1}r_k|| \leq ||A^{-1}|| \cdot ||r_k||$ the following inequalities are valid:

$$\frac{1}{\kappa}\frac{||r_k||}{||r_0||} \leq \frac{||e_k||}{||e_0||} \leq \kappa\frac{||r_k||}{||r_0||} \leq \kappa^2\frac{||e_k||}{||e_0||} \ , \tag{2.6}$$

where $\kappa = ||A||\ ||A^{-1}||$ is the condition number of A. Iterative methods are usually controlled by the observation of the norm of the residual. In reality the error should be controlled, but the error is not available. If κ is near to unity the norm of the relative residuals is strongly connected to the norm of the relative errors, i. e. if the residuals decrease sufficiently the errors will be reduced as well. For a large condition number the residuals can decrease while the norm of the errors does not decrease or even increases and vice-versa. Therefore, we always consider the residuals and the errors for all discussed methods.

An iterative method is parameter-dependent if parameters have to be determined a priori. Such methods are for example the SOR method [137] and the large family of semi-iterative methods [26, 27, 28, 73, 74] - to name only a few. Parameter-free methods calculate the iterates by using information of the iteration steps. Such methods are for example the Jacobi, the Gauss-Seidel and the large families of generalized CG [4, 35, 51, 61, 112], error-minimizing [39, 125], conjugate Krylov subspace and quasi-minimal residual methods [33, 34, 37, 38]. In LINSOL only parameter-free methods are implemented because in general inner properties of the matrices needed to determine the parameters are unknown.

2.2 Normalization

Most of the here discussed iterative methods are residual-reduction methods. If we multiply one of the n equations e.g. by 10^6, the solution does not change, but the component of the residual of that equation has 10^6-fold weight and dominates the whole iteration process. Therefore it is mandatory for residual reduction methods to normalize the equations that they have equal weight. Normalizations balance the equations, i. e.

for problems arising from partial differential equations the different physical components are equilibrized. Instead of the system (2.1) we solve the system

$$PAx = Pb \tag{2.7}$$

with the diagonal matrix

$$P = \begin{pmatrix} p_1 & 0 & \dots & \dots & 0 \\ 0 & p_2 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ 0 & \dots & \dots & 0 & p_n \end{pmatrix} . \tag{2.8}$$

The technique is also called diagonal scaling. The following choices of p_i are used:

$$p_i = \frac{1}{a_{i,i}} , \tag{2.9}$$

$$p_i = \frac{1}{\sum_{j=1}^{n} |a_{i,j}|} sign(a_{i,i}) , \tag{2.10}$$

$$p_i = \frac{1}{\sqrt{\sum_{j=1}^{n} a_{i,j}^2}} , \tag{2.11}$$

$$p_i = \frac{a_{i,i}}{\sum_{j=1}^{n} a_{i,j}^2} . \tag{2.12}$$

The choice of P according to (2.9) or (2.12) is not applicable if $a_{i,i} = 0$ for one i.

We present a sample problem in order to show how efficient such a simple technique can be for distinct problems. We use the following nomenclature.

O denotes no normalization.

I denotes a normalization according to (2.9). This technique is called Jacobi normalization because it is equivalent to perform a step of the Jacobi method.

II denotes a normalization according to (2.10). This technique is called line normalization.

III denotes a normalization according to (2.11).

IV denotes a normalization according to (2.12). In this case the Frobenius norm of $PA - I$ is minimized.

Sample Problem I Let us consider a rough model of the 3-dimensional Navier-Stokes equations. Let be

$$\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right)^T ,$$

$$\Delta = \nabla^T \nabla = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}.$$

We will solve the following partial differential equation (PDE) for the velocity

$$v = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} ,$$

$$\Delta v + v + \rho(v^T \nabla)v = h \ , \tag{2.13}$$

with Dirichlet boundary conditions on the unit cube. The parameter ρ simulates a Reynolds number. The right-hand side

$$h = \begin{pmatrix} h_1 \\ h_2 \\ h_3 \end{pmatrix}$$

is determined so that equation (2.13) has the exact solution

$$\bar{v} = \begin{pmatrix} \bar{v}_1 \\ \bar{v}_2 \\ \bar{v}_3 \end{pmatrix} = \begin{pmatrix} sin 2\pi x \ cos 2\pi y \ cos 2\pi z \\ cos 2\pi x \ sin 2\pi y \ cos 2\pi z \\ cos 2\pi x \ cos 2\pi y \ sin 2\pi z \end{pmatrix} .$$

The linear system arises from a finite difference discretization with consistency order 4 and from the linearization in the first Newton step. It is generated by the FIDISOL program package [102]. The calculations have been performed on a $20 \times 20 \times 20$ grid. The dimension of the linear system is $n = 24000$.

We use the PRES20 method for the sample problem I with $\rho = 1$. From figure 2.1 it is obvious that normalization drastically improves the convergence. Normalizations according to I, II and IV are comparable. While II gives the best result for the residuals, I is the best normalization for the errors. For $\rho = 500$ BICO is used. All normalizations and even no normalization give similar results, see figure 2.2. III and II are the best normalizations, in particular for the errors III is slightly better than all others. For $\rho = 10^5$ CGNR-ATPRES is used. In figure 2.3 the curves according to III and IV cannot be distinguished. For this example only normalization according to II or no

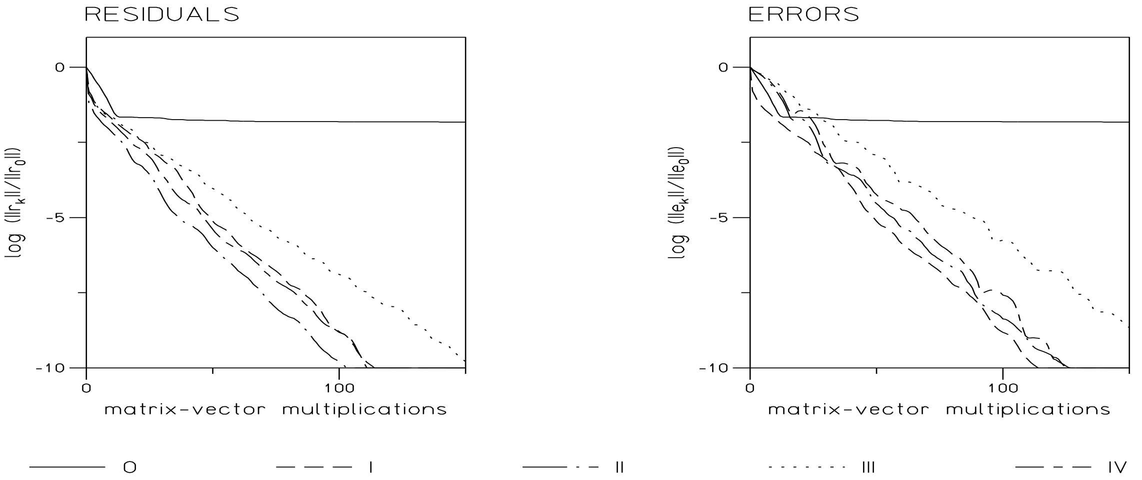


Figure 2.1: Comparison of normalizations for the sample problem I with $\rho = 1$ and PRES20 as basic method

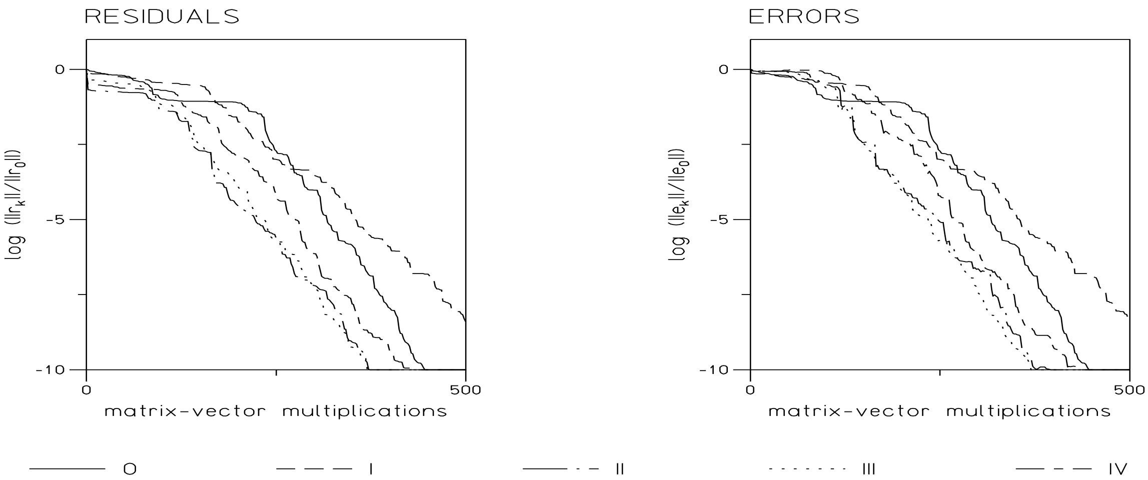


Figure 2.2: Comparison of normalizations for the sample problem I with $\rho = 500$ and BICO as basic method

normalization is applicable. For the errors no normalization is the best technique for a low accuracy but loses if a accuracy of more than 10^{-6} is required.

If the matrix A is symmetric, then we solve the system

$$PAPy = Pb \tag{2.14}$$

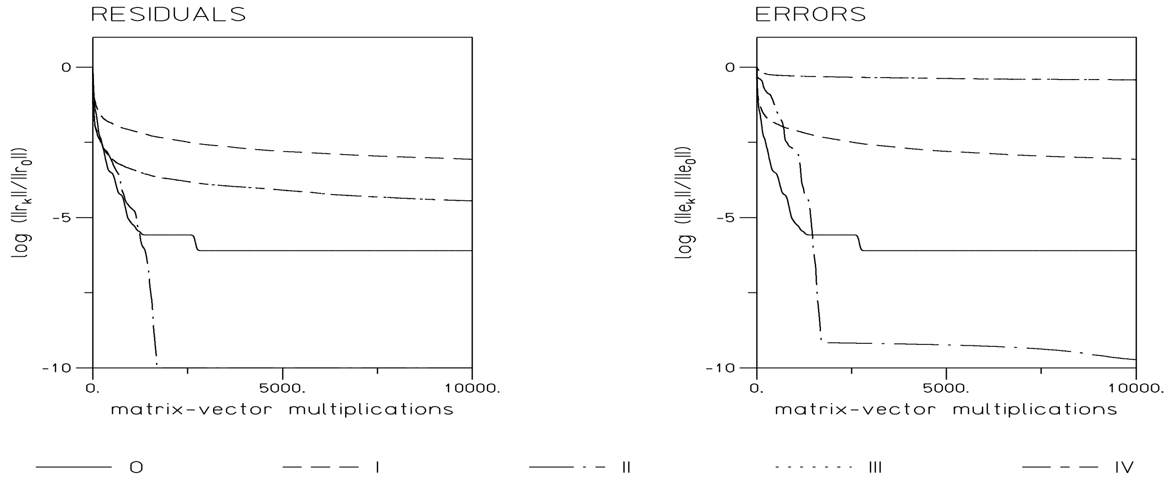


Figure 2.3: Comparison of normalizations for the sample problem I with $\rho = 10^5$ and ATPRES as basic method

because by this choice PAP is symmetric as well. In this case as normalization

$$\tilde{p}_i = \sqrt{|p_i|} sign(p_i) \tag{2.15}$$

is used with p_i according to (2.9), (2.10), (2.11) or (2.12). The solution x of (2.1) is obtained from

$$x = Py. \tag{2.16}$$

In LINSOL

`ilin(15) = msprec =`	0	*no normalization*
	1	*normalization I according to (2.7)*
	2	*normalization II according to (2.7)*
	3	*normalization III according to (2.7)*
	4	*normalization IV according to (2.7)*
	11	*normalization I according to (2.14)*
	12	*normalization II according to (2.14)*
	13	*normalization III according to (2.14)*
	14	*normalization IV according to (2.14)*

2.3 Smoothing

The norm of the residuals of a generalized conjugate gradient method may heavily oscillate. This fact is known from practice. Therefore, the smoothing algorithm was introduced by Schönauer [98, 93] in order to get a norm non-increasing function of the index for the residuals. In each iteration step a one-dimensional minimization is performed. However, the concept of smoothing is quite more general. It can be applied to any iterative method and we do not confine our investigations to the original residual-minimizing algorithm.

Let x_k be the iterate and r_k the residual resulting from an iterative method. The sequence

$$s_0 = r_0 \ , z_0 = x_0 \ , \tag{2.17}$$

$$s_k = s_{k-1} + \gamma_k(r_k - s_{k-1}) \ , \tag{2.18}$$

$$z_k = z_{k-1} + \gamma_k(x_k - z_{k-1}) \tag{2.19}$$

is a corresponding smoothed method.

The smoothed sequence is a mean of the actual element of the original sequence and the previous element of the smoothed sequence. The smoothed sequence does not react on the original sequence. The different smoothing techniques vary by the determination of the coefficient γ_k. We will describe in the following residual-minimizing and quasi-residual smoothing.

The original approach of Schönauer is to minimize the residual in the Euclidean norm

$$||s_k|| = min_{\gamma_k}||s_{k-1} + \gamma_k(r_k - s_{k-1})||. \tag{2.20}$$

We get

$$\gamma_k = -\frac{s_{k-1}^T(r_k - s_{k-1})}{||r_k - s_{k-1}||^2}. \tag{2.21}$$

This technique is called minimal residual (MR) smoothing (see Algorithm 1). A direct result of (2.20) is that the smoothed residual is a function of the iteration index, with a non-increasing norm, i. e.

$$\|s_k\| \leq \|s_{k-1}\|, \tag{2.22}$$

$$\|s_k\| \leq \|r_k\|. \tag{2.23}$$

The concept of quasi minimal residual (QMR) smoothing was introduced by Freund and Nachtigal [37] for the QMR method. Zhou and Walker [139] generalized the technique and created the name of this kind of smoothing. The idea is to choose s_k so

Let x_k be the iterate and r_k the residual resulting from an iterative method. Let

$$z_0 = x_0$$
$$s_0 = r_0$$

Calculate for $k \geq 1$:

$$\gamma_k = -\frac{s_{k-1}^T(r_k - s_{k-1})}{||r_k - s_{k-1}||^2}$$
$$z_k = z_{k-1} + \gamma_k(x_k - z_{k-1})$$
$$s_k = s_{k-1} + \gamma_k(r_k - s_{k-1})$$

Algorithm 1: MINIMAL RESIDUAL (MR) SMOOTHING

that

$$s_k = \frac{1}{\sum_{i=0}^{k} \frac{1}{||r_i||^2}} \sum_{i=0}^{k} \frac{1}{||r_i||^2} r_i \tag{2.24}$$

is satisfied (see Algorithm 2). This choice establishes a weighted mean of the original residuals. If $\|r_i\|$ is large, then the contribution to $\|s_k\|$ is neglectable because $\frac{1}{||r_i||^2}$ is small. Quasi-residual smoothing and residual-minimizing smoothing coincide if the r_k are orthogonal.

Since Schönauer's development [99] a lot of theoretical investigations have been performed [52, 124, 125, 126, 127, 123, 139]. It turns out that smoothing has some remarkable properties. In general smoothing transforms iterative methods. In [126] it is proved that MR smoothing tranforms classical CG into ORTHOMIN and ORTHORES into GMRES. Zhou and Walker [139] show that QMR smoothing transforms BCG into QMR and an auxiliary sequence resulting from CGS into TFQMR [34]. In Table 2.1 the transformations of distinct methods are given. For an explanation of the methods in Table 2.1 see the description of the particular methods.

Smoothing can be considered from a theoretical and a practical viewpoint. The results of the theory show that methods can be transformed by a very simple process. From a practical viewpoint in our opinion smoothing should be applied to basic methods with oscillating residual norms as for example BCG, in order to obtain in each iteration step a reliable approximation, all the more because the additional work can be neglected. In Figure 2.4 the basic method BCG is compared with the MR smoothed BICO and the QMR smoothed QMR. Peaks of BCG are eliminated by both techniques and BICO and QMR behave very similar. The number of matrix-vector multiplications is a measure for the number of iterations. Note that the scale is logarithmic for the relative residuals and errors so that the non-smoothed BCG method oscillates in orders of magnitude.

Let x_k be the iterate and r_k the residual resulting from an iterative method. Let

$$z_0 = x_0$$
$$s_0 = r_0$$
$$\tau_0^2 = ||r_0||^2$$

Calculate for $k \geq 1$:

$$\tau_k^2 = \frac{1}{\frac{1}{\tau_{k-1}^2} + \frac{1}{||r_k||^2}}$$
$$\gamma_k = \frac{\tau_k^2}{||r_k||^2}$$
$$z_k = z_{k-1} + \gamma_k(x_k - z_{k-1})$$
$$s_k = s_{k-1} + \gamma_k(r_k - s_{k-1})$$

Algorithm 2: QUASI MINIMAL RESIDUAL (QMR) SMOOTHING

MR smoothing:	CG	→	ORTHOMIN
	ORTHORES	→	GMRES
	CGNE	→	CGNR
QMR smoothing:	BCG	→	QMR
	CGS	→	TFQMR

Table 2.1: Transformations by smoothing

Finally, we observe that different methods can be implemented simultaneously without essential computational work by smoothing, for example the pairs ORTHORES and GMRES, BCG and QMR, BCG and BICO, CGNE and CGNR.

In LINSOL

`ilin(18) = smooth` `= 0` *returns the non-smoothed solution*
`> 0` *returns the MR smoothed solution*
`< 0` *returns the QMR smoothed solution*

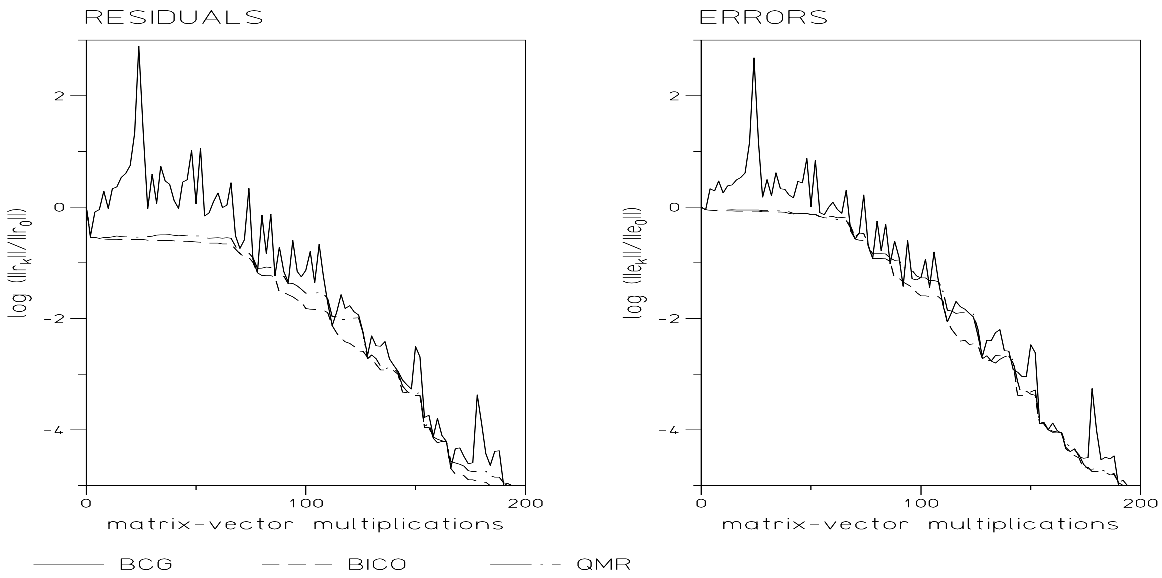


Figure 2.4: Comparison of different smoothers for the sample problem I with $\rho = 500$ and BCG as basic method

2.4 Stopping Criteria

In LINSOL the following stopping criteria are used. The user specifies a relative tolerance ϵ. If the Euclidean residual norm of the normalized system (2.7) fulfils

$$\frac{\|PAx_k - Pb\|}{\|Pb\|} < \epsilon, \tag{2.25}$$

and the maximum residual norm of the original system (2.1) or the normalized system (2.7) fulfils

$$\frac{\|Ax_k - b\|_\infty}{\|b\|_\infty} < \epsilon, \tag{2.26}$$

$$\frac{\|PAx_k - Pb\|_\infty}{\|Pb\|_\infty} < \epsilon, \tag{2.27}$$

respectively, then the iteration is stopped. LINSOL stops as well if a maximum number of iterations specified by the user is performed.

In LINSOL	
`ilin(9) = noprec`	`= 0` *normalized system is considered (2.27)*
	$\neq$ `0` *original system is considered (2.26)*
`ilin(2) = itmax`	*maximum number of iterations*

Chapter 3

The Symmetric and Positive Definite Case

In this chapter methods for symmetric and positive definite systems are discussed. The matrix A is called symmetric if

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

and it is called positive definite if

$$y^T A y > 0 \tag{3.2}$$

for all vectors $0 \neq y \in \mathbb{R}^n$.

The method of steepest descent is fundamental for many modern iterative techniques. The idea of the method of steepest descent, proposed by Temple [113], is to minimize the quadratic form

$$F(z) = \frac{1}{2}(Az - b)^T A^{-1}(Az - b)\ . \tag{3.3}$$

If A is symmetric and positive definite, then A^{-1} is symmetric and positive definite. Therefore, $F(z) \geq 0$ for all z and $F(z) = 0$ if and only if $Az - b = 0$. Therefore, the minimum of (3.3) is the solution of (2.1).

The gradient of (3.3) is precisely $Az - b$. The gradient is orthogonal to the isolines of (3.3) and, therefore, in the direction of the steepest descent. The gradient at the iteration step $k-1$ is $Ax_{k-1} - b = r_{k-1}$. It is quite natural to choose

$$x_k = x_{k-1} + \delta_k r_{k-1} \tag{3.4}$$

and to determine δ_k by a one-dimensional minimization so that

$$F(x_k) = \min_{\delta_k} F(x_{k-1} + \delta_k r_{k-1}). \tag{3.5}$$

Equation (3.5) is equivalent to

$$\delta_k = -\frac{r_{k-1}^T r_{k-1}}{r_{k-1}^T A r_{k-1}}. \tag{3.6}$$

The update vector $\delta_k r_{k-1} = \delta_k(Ax_{k-1} - b)$ is a multiple of the gradient of the functional (3.3) in the point x_{k-1} and, therefore, orthogonal to the isoline $F(z) = F(x_{k-1})$. Thus the method proceeds in the direction of the steepest descent. Moreover, the update direction is parallel to the isoline $F(z) = F(x_k)$ because of the minimization property (3.5); see in figure 3.1 how the method proceeds from the old to the new iterates. The method converges, if A is positive definite.

Note that the square root of

$$2F(x_k) = r_k^T A^{-1} r_k = e_k^T A e_k = \|e_k\|_A^2 \tag{3.7}$$

is called energy norm of the error because for systems arising from structural analysis this quantity is equivalent to the potential energy of the discretized problem. It would be meaningful to minimize the Euclidean norm of the error $\|e_k\|$. But for this choice the coefficient δ_k in (3.4) cannot be determined because the unknown error of the previous step is needed. Another feasible possibility is to minimize the residual norm $\|r_k\|$.

3.1 Classical CG

The conjugate gradient (CG) method was developed by Hestenes and Stiefel [61] in the early 1950s for the solution of linear systems with a symmetric, positive definite matrix. The method can be considered as a direct method because the exact solution is obtained at least in the step n in the absence of roundoff errors. Until the early 1970s the method was not widely used. With the increasing use of computers and the consideration as an iterative method by Reid [82] the method recovered importance.

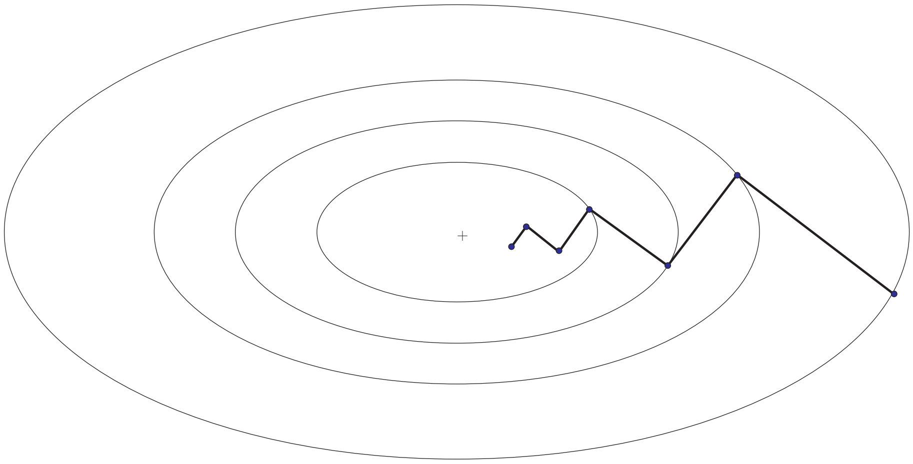
The idea of the classical conjugate gradient method is to minimize the quadratic form (3.3) by using a multi-dimensional minimization improving the steepest descent method. For the CG method choose

$$x_k = x_{k-1} + \sum_{i=1}^{k} \delta_{i,k} r_{i-1}. \tag{3.8}$$

The $\delta_{i,k}$ are determined from

$$\begin{aligned} F(x_k) &= \min_{\delta_{1,k},\ldots,\delta_{k,k}} F(x_{k-1} + \sum_{i=1}^{k} \delta_{i,k} r_{i-1}) \\ &\overset{(3.7)}{=} \frac{1}{2} \min_{\delta_{1,k},\ldots,\delta_{k,k}} ||e_{k-1} + \sum_{i=1}^{k} \delta_{i,k} r_{i-1}||_A^2. \end{aligned} \tag{3.9}$$

steepest descent



classical CG

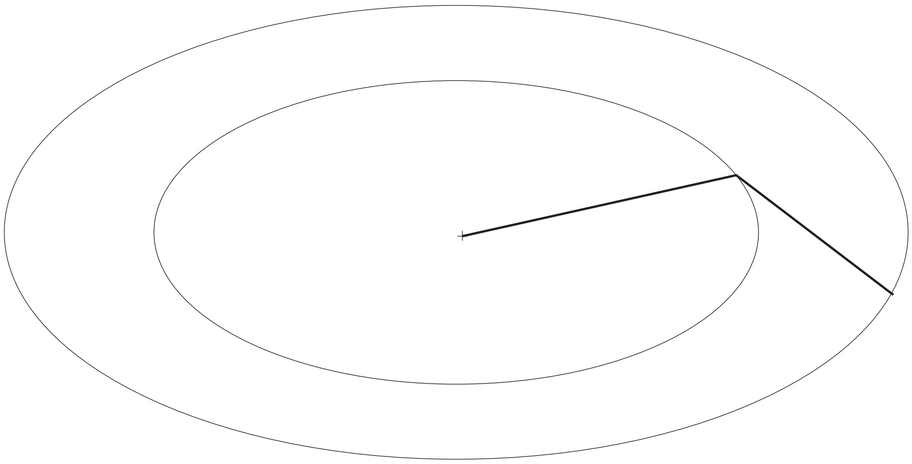


Figure 3.1: Proceeding of the steepest descent and the classical CG method

If the ellipses of the isolines of $F(z) = constant$ have very different semi-axes the convergence of the steepest descent method may be poor, while the more sophisticated update direction in (3.8) improves the convergence. In figure 3.1 the proceeding of classical CG is depicted in comparison with the steepest descent method. For this two-dimensional example the exact solution is achieved in the second iteration step for classical CG because of the two-dimensional minimization (3.9).

We get in the absence of rounding errors the exact solution in the iteration step n. However, a considerable reduction of the residuals and errors is generally obtained in far less iteration steps.

We obtain from (3.9) the orthogonality of the residuals

$$r_k^T r_{j-1} = 0 \tag{3.10}$$

for $j = 1, \ldots, k$. Thus, the minimization condition (3.9) is equivalent to the orthogonality conditions (3.10). Instead of using the minimization property for the definition of CG the orthogonalization conditions can be used. The orthogonalities can be considered as weak formulation for the condition that the residual vanishes for the true solution.

Equation (3.8) suggests that the length of the recurrence increases with the iteration but the new iterate of classical CG can be calculated by a short recurrence, i. e. two 2-term recurrences. The algorithm will be given in the description of ORTHOMIN because both methods can be implented simultaneously (see Algorithm 3).

The method converges, if A is symmetric and positive definite and then

$$||e_k||_A \leq 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k ||e_0||_A \tag{3.11}$$

where $\kappa(A) = \|A\| \, \|A^{-1}\|$ is the condition number of A. An easy-to-read introduction to classical CG is given by Shewchuk [106].

3.2 ORTHOMIN

The idea of ORTHOMIN by Vinsome [120] is to choose x_k as in (3.8), but to determine the $\delta_{i,k}$ from

$$\|r_k\| \quad = \quad \min_{\delta_{1,k},\ldots,\delta_{k,k}} \|r_{k-1} + \sum_{i=1}^{k} \delta_{i,k} A r_{i-1}\|. \tag{3.12}$$

The method can be obtained from classical CG by MR smoothing. The presented algorithm is based on this approach and different from the original implementation by Vinsome (see Algorithm 3). The algorithm requires one matrix-vector multiplication by A per iteration step.

In LINSOL
`ilin(1) = ms = 123` *and* `ilin(18) = smooth = 0` *CG*
`ilin(1) = ms = 123` *and* `ilin(18) = smooth > 0` *ORTHOMIN*

Choose an initial guess $x = z$, $r = p = s = Ax - b$, $\rho = \|r\|^2$ *and calculate in each iteration step*

$$\left.\begin{array}{l} \left.\begin{array}{l} f = Ap \\ \alpha = -\frac{r^T r}{p^T f} \\ x = x + \alpha p \\ r = r + \alpha f \\ \rho_{old} = \rho \\ \rho = \|r\|^2 \\ \beta = \frac{\rho}{\rho_{old}} \\ p = r + \beta p \end{array}\right\} CG \\ \\ \left.\begin{array}{l} \gamma = -\frac{s^T(r-s)}{(r-s)^T(r-s)} \\ s = s + \gamma(r - s) \\ z = z + \gamma(x - z) \end{array}\right\} MR\ smoothing \end{array}\right\} ORTHOMIN$$

Use as approximation

$$\begin{array}{ll} x & \textit{for CG,} \\ z & \textit{for ORTHOMIN.} \end{array}$$

Algorithm 3: CG and ORTHOMIN

3.3 Fridman's Method

The idea of Fridman's algorithm [39] (see Algorithm 4) is to choose x_k a little bit different than in (3.8)

$$x_k = x_{k-1} + \sum_{i=1}^{k} \delta_{i,k} A r_{i-1}, \tag{3.13}$$

and to minimize the error in the Euclidean norm

$$||e_k|| \quad = \quad \min_{\delta_{1,k},\ldots,\delta_{k,k}} \|r_{k-1} + \sum_{i=1}^{k} \delta_{i,k} A r_{i-1}\|. \tag{3.14}$$

Fridman's implementation turned out to be unstable, see [112]. A stable version was proposed by Stoer and Freund, also in [112], and by Freund [32]. Fletcher [31] showed that auxiliary iterates calculated by SYMMLQ [78] coincide mathematically with the

Let $x = z$ *be any initial guess,* $r = s = Ax - b$, $old = 0$, $new = 1$, $\alpha_{new} = 0$, $q_{old} = A^T r$, $\rho = \|q_{old}\|$, $q_{old} = \frac{q_{old}}{\rho}$, $y_{old} = \frac{r}{\rho}$, $k = 1$. *Calculate in each iteration step*

$$\begin{array}{l}
\gamma = b^T y_{old} - x^T q_{old} \\
x = x + \gamma q_{old} \\
if\ mod(k,20) = 0\ then \\
\qquad \left.\begin{array}{l} r = Ax - b \\ \gamma = -\frac{s^T(r-s)}{(r-s)^T(r-s)} \\ s = s + \gamma(r-s) \\ z = z + \gamma(x-z) \end{array}\right\} \begin{array}{l} residual+ \\ MR\ smoothing \end{array} \\
f = Aq_{old}, \\
\alpha_{old} = -q_{old}^T f \\
if\ k \geq 2\ then \\
\qquad \alpha_{new} = -q_{new}^T f \\
q_{new} = f + \alpha_{old} q_{old} + \alpha_{new} q_{new} \\
\rho = ||q_{new}|| \\
if\ \rho\ \leq\ 10^{-20}\ then \\
\qquad restart \\
\phi = \dfrac{1}{\rho} \\
q_{new} = \phi q_{new} \\
y_{new} = \phi(q_{old} + \alpha_{old} y_{old} + \alpha_{new} y_{new}) \\
h = old,\ old = new,\ new = h,\ k = k+1 \\
restart: \left\{\begin{array}{l} r = Ax - b \\ \gamma = -\frac{s^T(r-s)}{(r-s)^T(r-s)} \\ s = s + \gamma(r-s) \\ z = z + \gamma(x-z) \\ old = 0,\ new = 1,\ k = 1 \\ q_{old} = Ar \\ \rho = \|q_{old}\| \\ q_{old} = \frac{q_{old}}{\rho} \\ y_{old} = \frac{r}{\rho} \\ \alpha_{new} = 0 \end{array}\right.
\end{array}$$

Algorithm 4: Fridman's algorithm

property	method
energy norm-minimizing	CG
residual-minimizing	ORTHOMIN
error-minimizing	Fridman's method

Table 3.1: Comparison of minimization properties in the symmetric case

approximations of Fridman's algorithm. Therefore, SYMMLQ generates a stable implementation of that method as well. The algorithm requires one matrix-vector multiplication by A per iteration step.

In LINSOL
`ilin(1) = ms = 124` *and* `ilin(18) = smooth = 0` *Fridman's method*

3.4 Comparison

The three methods for symmetric matrices are closely related. It can be shown that the iterates of CG and ORTHOMIN can be represented by

$$x_k = x_0 + \sum_{i=0}^{k-1} \beta_{i,k} A^i r_0, \tag{3.15}$$

while those of Fridman's method fulfil

$$x_k = x_0 + \sum_{i=0}^{k-1} \beta_{i,k} A^{i+1} r_0. \tag{3.16}$$

The space that is spanned by the vectors $r_0, \ldots, A^i r_0$ is called Krylov space generated by r_0 and A. Therefore these methods are also referred as Krylov subspace methods. The minimization properties are listed in Table 3.1. In table 3.2 the methods are classified with respect to work and storage requirements. The requirements for all three methods are comparable. Note that in LINSOL methods that can be transformed by smoothing are implemented as one algorithm so that the storage requirements always include the two additional vectors needed for smoothing. Moreover 5 additional buffer vectors are required for a multiplication by a symmetric matrix.

In LINSOL
Dimension `ldw` *of the array* `dw`
`ldw = 11` *for CG, ORTHOMIN*
`ldw = 14` *for Fridman's method*

method	mvm	st
CG	1	4
ORTHOMIN	1	6
Fridman's method	1	7

mvm = matrix-vector multiplications per iteration, st = number of auxiliary vectors of length n

Table 3.2: Comparison of work and storage in the symmetric case

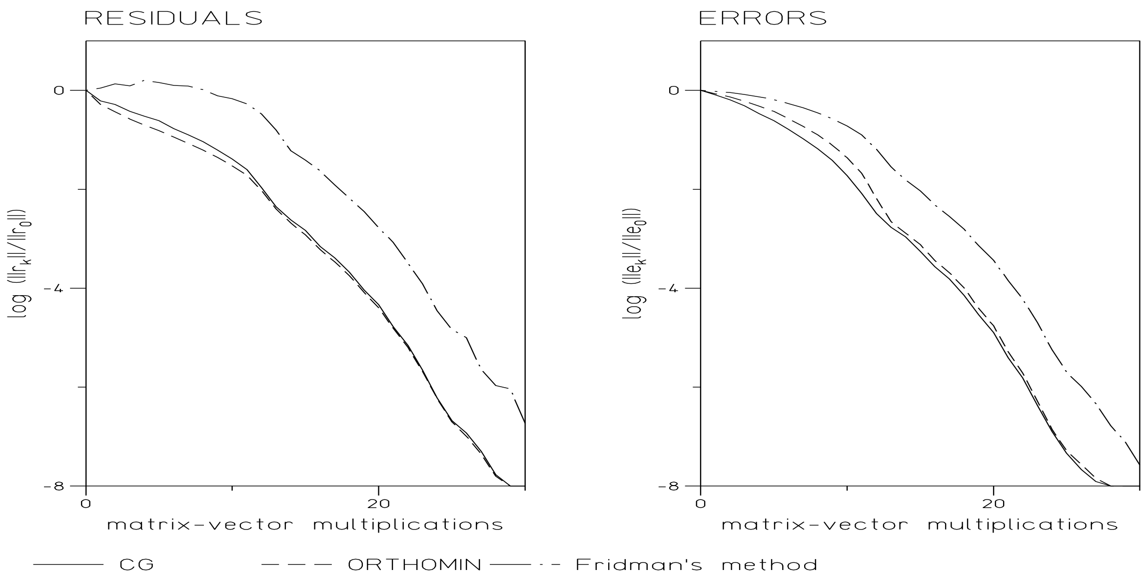


Figure 3.2: GMRES, CG and GMERR for sample problem I

In Figure 3.2 we consider the reduction of the residuals and errors for the sample problem

$$\Delta v = h \tag{3.17}$$

with Dirichlet boundary conditions on the unit cube, where $v = (v_1, v_2, v_3)^T$, $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$. The right-hand side $h = (h_1, h_2, h_3)^T$ is determined so that equation (3.17) has the exact solution

$$\bar{v} = \begin{pmatrix} \bar{v}_1 \\ \bar{v}_2 \\ \bar{v}_3 \end{pmatrix} = \begin{pmatrix} sin2\pi x \; cos2\pi y \; cos2\pi z \\ cos2\pi x \; sin2\pi y \; cos2\pi z \\ cos2\pi x \; cos2\pi y \; sin2\pi z \end{pmatrix} .$$

The linear system arises from a finite difference discretization with consistency order 2 generated by the FIDISOL program package [102] on a $20 \times 20 \times 20$ grid. The matrix is normalized according to (2.15) with p_i as in (2.9). The dimension of the system is $n = 24000$. The convergence of CG, ORTHOMIN and Fridman's method is comparable, see Figure 3.2. Note that the number of matrix-vector multiplications corresponds to the number of iterations. It should be stressed that a relative reduction of the residuals and errors by 8 orders of magnitude is achieved with 30 i. e. in $1.25 \cdot 10^{-3} \cdot n$ matrix-vector multiplications. Thus, the iterative methods are by a factor of 10^3 faster than direct methods.

Chapter 4

The Nonsymmetric Case

The method of steepest descent and the classical CG method are the basis for generalized CG methods. These two methods are suited for matrices that are diagonally dominant or symmetric and positive definite. Of course the methods can also be applied to other systems, but the convergence is not guaranteed or may be very slow. Starting from the 1970s the classical CG method has been generalized in order to obtain convergent techniques for non-symmetric and non-positive definite systems. A whole family of methods results from this generalization; see e. g. [4, 29, 35, 51, 61, 112].

4.1 Long Recurrences and Restarted or Truncated Methods

4.1.1 ORTHORES

In this section we describe a direct generalization of classical CG for nonsymmetric matrices. This family of methods is characterized by the minimization of the energy norm of the error in the following sense:

$$\|e_k\|_A \le \sqrt{1+\frac{\rho^2}{\mu^2}} \min_{\beta_1,\dots,\beta_k} \|\sum_{i=1}^{k} \beta_i A^i e_0 + e_0\|_A, \tag{4.1}$$

where ρ is the spectral radius of the skewsymmetric part $R = \frac{1}{2}(A - A^T)$ of A and μ is the smallest eigenvalue of the symmetric part $M = \frac{1}{2}(A + A^T)$ of A. If A is symmetric and positive definite, then we obtain the classical CG method [61] by Hestenes and Stiefel. Paige and Saunders proposed SYMMLQ [78], a numerical stable implementation for symmetric matrices. For positive real matrices we obtain ORTHORES (ORTHOgonal RESiduals), proposed by Young [137]. The algorithm will be given in the description of GMRES because both methods can be implented simultaneously (see Algorithm 5).

4.1.2 GMRES

For these methods the residuals are minimized in the Euclidean norm.

$$||r_k|| \quad = \quad \min_{\beta_1 \ ,..,\beta_k} ||\sum_{i=1}^{k} \beta_i A^i r_0 + r_0||, \tag{4.2}$$

One of the first realizations is ORTHOMIN by Vinsome [120] for symmetric positive definite matrices. Paige and Saunders proposed MINRES [78], a numerically stable implementation for symmetric matrices. GMRES (Generalized Minimal RESidual method) was proposed by Saad and Schultz [90] and works for arbitrary matrices. There are various implementations of GMRES [122]. We give the implementation based on MR smoothing of ORTHORES (see Algorithm 5). The algorithms require one matrix-vector multiplication by A per iteration step. The methods cannot be calculated with a short recurrence unless certain favorable conditions are valid.

It should be stated that the ORTHORES implementation describes as well the steepest descent as the classical CG methods. The steepest descent method according is equivalent to Algorithm 5 with $\nu = 0$ instead of $(*)$ $\nu = \nu + 1$. Classical CG can be implemented as special ORTHORES method according to Algorithm 5 with $\nu = 1$ instead of $(*)$ $\nu = \nu + 1$.

4.1.3 GMERR

GMERR [125, 83] (see Algorithm 6) minimizes the error

$$||e_k|| \quad = \quad \min_{\beta_1,\dots,\beta_k} ||\sum_{i=1}^{k} \beta_i \left(A^T\right)^i r_0 + e_0||. \tag{4.3}$$

We get the classical algorithm of Fridman [39] if A is symmetric and positive definite.

This implementation is a Gram-Schmidt algorithm for q_k. The algorithm requires one matrix-vector multiplication by A^T per iteration step. The exact variant cannot be calculated with a short recurrence unless certain favorable conditions are valid. For example, for a symmtric A the calculation of the q_k is possible with a short recurrence.

A problem for GMERR is the controlling of the convergence because the errors decrease but they cannot be calculated. If we observe the norm of the residuals, it may oscillate heavily. Moreover, an additional matrix-vector multiplication is needed for the calculation of the residuals. Therefore, we calculate the residuals only every 20th step

Choose an initial guess $x_0 = z$, $r_0 = s = Ax_0 - b$, $k = 0$, $\nu = 0$, $new = 0$, $\beta_0 = 1$ and calculate in each iteration step

$$\left.\begin{array}{l}
\left.\begin{array}{ll}
 & k = k + 1,\ \ old = new,\ \ new = new + 1 \\
 & f = Ar_{old} \\
 & \alpha_i = -\frac{r_i^T f}{r_i^T r_i}\ \ for\ i = 0, \ldots, \nu \\
 & \psi = \sum_{i=0}^{\nu} \alpha_i \beta_i \\
 & if\ \psi \neq 0\ then \\
 & \qquad \phi = \frac{1}{\psi} \\
 & \qquad \beta_{new} = 1 \\
 & else \\
 & \qquad \left.\begin{array}{l} \phi = 1 \\ \beta_{new} = 0 \end{array}\right\} curable\ breakdown \\
 & r_{new} = \phi(f + \sum_{i=0}^{\nu} \alpha_i r_i) \\
 & x_{new} = \phi(r_{old} + \sum_{i=0}^{\nu} \alpha_i x_i) \\
(*) & \nu = \nu + 1
\end{array}\right\} ORTHORES \\
\quad if\ \psi \neq 0\ then \\
\left.\begin{array}{l}
\qquad \gamma = -\frac{s^T(r_{new} - s)}{(r_{new} - s)^T(r_{new} - s)} \\
\qquad s = s + \gamma(r_{new} - s) \\
\qquad z = z + \gamma(x_{new} - z)
\end{array}\right\} MR\ smoothing
\end{array}\right\} GMRES$$

Use as approximation

$$\begin{array}{ll}
x_{new} & for\ ORTHORES\ if\ \beta_{new} \neq 0\ , \\
z & for\ GMRES.
\end{array}$$

Algorithm 5: ORTHORES and GMRES

and to apply residual-minimizing smoothing. The norm of the smoothed residuals is used for control.

The investigations in [32] show that GMERR is competitive with GMRES if the matrix is symmetric. The residual reduction is slightly better for GMRES while the error reduction is slightly better for GMERR. For non-normal matrices there are no theoretical results up to now. Practical tests give a rather confusing impression. There are examples where GMRES works fine while GMERR converges very slowly. There are other examples where GMERR is very fast but GMRES is very bad.

Let $x = z$ *be any initial guess,* $r = s = Ax - b$, $q_0 = A^T r$, $\rho = \|q_0\|$, $q_0 = \frac{q_0}{\rho}$, $y_0 = \frac{r}{\rho}$, $k = 1$, $\nu = 1$. *Calculate in each iteration step*

$$
\begin{aligned}
&\gamma = b^T y_{k-1} - x^T q_{k-1}, \\
&x = x + \gamma q_{k-1} \ , \\
&if\ mod(k,20) = 0\ then \\
&\qquad \left.\begin{aligned} &r = Ax - b \\ &\gamma = -\tfrac{s^T(r-s)}{(r-s)^T(r-s)} \\ &s = s + \gamma(r-s) \\ &z = z + \gamma(x - z) \end{aligned}\right\} \begin{aligned} &residual+ \\ &MR\ smoothing \end{aligned} \\
&f = A^T q_{k-1} \\
&for\ i = 1, \ldots, \nu\ calculate \\
&\qquad \alpha_i = -q_{k-i}^T f \\
&q_k = f + \sum_{i=1}^{\nu} \alpha_i q_{k-i} \ , \\
&\rho = ||q_k|| \\
&if\ \rho\ \leq\ 10^{-20}\ then \\
&\qquad restart \\
&\phi = \frac{1}{\rho} \ , \\
&q_k = \phi q_k \ , \\
&y_k = \phi(q_{k-1} + \sum_{i=1}^{\nu} \alpha_i y_{k-i}) \ . \\
&k = k + 1 \\
(*) \qquad &\nu = \nu + 1 \\
&restart : \left\{\begin{aligned} &r = Ax - b \\ &\gamma = -\tfrac{s^T(r-s)}{(r-s)^T(r-s)} \\ &s = s + \gamma(r-s) \\ &z = z + \gamma(x - z) \\ &q_0 = A^T r \\ &\rho = \|q_0\| \\ &q_0 = \tfrac{q_0}{\rho} \\ &y_0 = \tfrac{r}{\rho} \\ &k = 1 \end{aligned}\right.
\end{aligned}
$$

Algorithm 6: GMERR

4.1.4 Truncated and Restarted Versions

For exact methods the storage and computing requirements increase with the iteration step. For full matrices the storage increases for exact methods only up to 2 times of the amount needed for the matrix itself. Thus, for full matrices exact methods can be applied instead of preconditioning. For large and sparse systems the situation is different because the storage would increase up to that of the full matrix so that even the capacity of today's supercomputers would be exceeded in later steps. Exact methods can only be reformulated as short recurrences if special conditions are satisfied. For example: for GMRES, ORTHORES and GMERR A has to be symmetric. In general truncated and restarted versions of the above mentioned methods have to be applied to solve non-symmetric problems for large and sparse systems.

ORTHORES-R(σ) is the restarted ORTHORES method according to Algorithm 5 with $\nu = k \ mod \ \sigma$ instead of $(*)\ \nu = \nu + 1$. Minimization properties corresponding to (4.1) are valid in the restart interval.

ORTHORES-T(σ) is the truncated ORTHORES method according to Algorithm 5 with $\nu = min(k, \sigma - 1)$ instead of $(*)\ \nu = \nu + 1$. Minimization properties corresponding to (4.1) are unknown.

PRES20 (Pseudo-RESidual method 20) [93] (see Algorithm 7) is a special version of smoothed truncated ORTHORES-T(5). The sequence is restarted every 20th step from the smoothed sequence. As a result of many tests with very large and sparse matrices these parameters of truncation and restart have been demonstrated to be very efficient. The method requires one matrix-vector multiplication by A in each iteration step. A short recurrence is forced by truncation and restart.

GMRES-R(σ) is the restarted GMRES method according to Algorithm 5 with $\nu = k \ mod \ \sigma$ instead of $(*)\ \nu = \nu + 1$. Minimization properties corresponding to (4.2) are valid in the restart interval.

GMRES-T(σ) is the truncated GMRES method according to Algorithm 5 with $\nu = min(k, \sigma - 1)$ instead of $(*)\ \nu = \nu + 1$. Minimization properties corresponding to (4.2) are valid in the truncation interval.

GMERR(σ) is the restarted GMERR method according to Algorithm 6 with $\nu = (k-1) \ mod \ \sigma \ + 1$ instead of $(*)\ \nu = \nu + 1$. Minimization properties corresponding to

Choose an initial guess $x_0 = z$, $r_0 = s = Ax_0 - b$, $k1 = 0$, $new = 0$, $\beta_0 = 1$ *and calculate in each iteration step*

$$
\left.\begin{array}{l}
k1 = k1 + 1,\ \ old = new,\ \ new = mod(new+1, 5) \\
f = Ar_{old} \\
\sigma = min(k1, 5) - 1 \\
\alpha_i = -\frac{r_i^T f}{r_i^T r_i}\ \ for\ i = 0, \ldots, \sigma \\
if\ \ \sum_{i=0}^{\sigma} \beta_i = 0\ then \\
\qquad stop\ incurable\ breakdown \\
\psi = \sum_{i=0}^{\sigma} \alpha_i \beta_i \\
if\ \psi \neq 0\ then \\
\qquad \phi = \frac{1}{\psi} \\
\qquad \beta_{new} = 1 \\
else \\
\qquad \left.\begin{array}{l} \phi = 1 \\ \beta_{new} = 0 \end{array}\right\} curable\ breakdown \\
r_{new} = \phi(f + \sum_{i=0}^{\sigma} \alpha_i r_i) \\
x_{new} = \phi(r_{old} + \sum_{i=0}^{\sigma} \alpha_i x_i)
\end{array}\right\} ORTHORES-T(5)
$$

$$
\begin{array}{l}
if\ \psi \neq 0\ then \\
\qquad \left.\begin{array}{l}
\gamma = -\frac{s^T(r_{new}-s)}{(r_{new}-s)^T(r_{new}-s)} \\
s = s + \gamma(r_{new} - s) \\
z = z + \gamma(x_{new} - z)
\end{array}\right\} MR\ smoothing
\end{array}
$$

$$
\begin{array}{l}
if\ mod(k1, 20) = 0\ then \\
\qquad \left.\begin{array}{l}
x_{old} = z \\
r_{old} = s \\
new = 0,\ \ k1 = 0,\ \ \beta_0 = 1
\end{array}\right\} restart
\end{array}
$$

Use as approximation z for PRES20.

Algorithm 7: PRES20

(4.3) are valid in the restart interval.

In LINSOL

`ilin(1) = ms = 7` *and* `ilin(18) = smooth = 0`	*ORTHORES-R(5)*
`ilin(1) = ms = 8` *and* `ilin(18) = smooth = 0`	*ORTHORES-T(5)*
`ilin(1) = ms = 1` *and* `ilin(18) = smooth > 0`	*PRES20*
`ilin(1) = ms = 7` *and* `ilin(18) = smooth > 0`	*GMRES-R(5)*
`ilin(1) = ms = 8` *and* `ilin(18) = smooth > 0`	*GMRES-T(5)*
`ilin(1) = ms = 11` *and* `ilin(18) = smooth = 0`	*GMERR(5)*

4.2 Biconjugate Gradient-based Methods

In this section methods are discussed that have the same origin: the biconjugate gradients (BCG). All of them have the common property that two matrix-vector multiplications have to be performed in each iteration step and that the exact methods can be formulated with a short recurrence. The latter property is the reason for the popularity of BCG-based methods. We will discuss BCG itself, BICO, QMR, CGS and BiCGSTAB.

4.2.1 BCG

The biconjugate gradients (BCG) were introduced by Lanczos [69] and later reformulated by Fletcher [31]. The method can be considered as method generated by the Lanczos algorithm [68]. BCG can also be considered as method applied to the double system

$$\begin{pmatrix} A & 0 \\ 0 & A^T \end{pmatrix} \begin{pmatrix} x \\ x^* \end{pmatrix} = \begin{pmatrix} b \\ b^* \end{pmatrix}, \tag{4.4}$$

where b^* is arbitrary. The iterates are calculated by a double CG ansatz (3.8)

$$x_k = x_{k-1} + \sum_{i=1}^{k} \delta_{i,k} r_{i-1}, \tag{4.5}$$

$$x_k^* = x_{k-1}^* + \sum_{i=1}^{k} \delta_{i,k} r_{i-1}^*, \tag{4.6}$$

where the residuals have the form

$$r_k = Ax_k - b \tag{4.7}$$

$$r_k^* = A^T x_k^* - b^*. \tag{4.8}$$

r_k^* is called shadow residual. The coefficients are obtained from the biorthogonalities

$$r_k^T r_{k-i}^* = 0, \tag{4.9}$$

$$[r_k^*]^T r_{k-i} = 0, \tag{4.10}$$

for $i = 1, \ldots, k$. These orthogonalities can be considered as weak formulation for the condition that the residual and the shadow residual vanish for the true solution. This weak condition resembles the CG condition (3.10). It turns out that the iterates can be calculated by a short recurrence.

The residuals can be represented with a matrix polynomial

$$\Pi_k(A) = \sum_{i=1}^{k} \gamma_i A^i + I \tag{4.11}$$

so that

$$r_k = \Pi_k(A) r_0, \tag{4.12}$$

$$r_k^* = \Pi_k(A^T) r_0^*. \tag{4.13}$$

Thus, the same polynomial is used for both seqences.

The algorithm produces highly oscillating residuals. The method may break down if a division by zero occurs. In this case look-ahead strategies, see [12, 13, 14, 36, 136] can avoid breakdowns. A simple method to overcome breakdowns is to restart the algorithm. This remedy is sufficient according to our experience in many practical cases. The algorithm will be given in the description of BICO because both methods can be implented simultaneously (see Algorithm 8).

4.2.2 BICO

It is known from practice that the convergence behavior of BCG in the Euclidean norm can be very erratic. In order to obtain a better behavior Schönauer applied minimal residual smoothing to BCG. This algorithm is called BICO (see Algorithm 8). Thus, we obtain

$$\|r_k\| = \min_{\gamma_k} \|r_{k-1} + \gamma_k (r_k^{BCG} - r_{k-1})\|, \tag{4.14}$$

A direct result of the residual-minimizing smoothing is that the BICO residual is a function of the iteration index, with a non-increasing norm, i. e.

$$\|r_k\| \leq \|r_{k-1}\|, \tag{4.15}$$

and

$$\|r_k\| \leq \|r_k^{BCG}\|. \tag{4.16}$$

The algorithm requires two matrix-vector multiplications per iteration step, one by A and one by A^T. Note that the structure of the algorithm is similar to classical CG, Algorithm 3.

In LINSOL
`ilin(1) = ms = 2` *and* `ilin(18) = smooth = 0` *BCG*
`ilin(1) = ms = 2` *and* `ilin(18) = smooth > 0` *BICO*

Choose an initial guess $x = z$, $r = s = r^* = p = p^* = Ax - b$, $\rho = r^T r^*$ *and calculate in each iteration step*

$$
\left.
\begin{array}{l}
\left.
\begin{array}{l}
f = Ap \\
f^* = A^T p^* \\
if\ p^T f^* \neq 0\ and\ \rho \neq 0\ then \\
\qquad \delta = -\frac{r^T r^*}{p^T f^*} \\
else \\
\qquad restart \\
r = r + \delta f \\
r^* = r^* + \delta f^* \\
x = x + \delta p
\end{array}
\right\} \ BCG \\
\\
\left.
\begin{array}{l}
\gamma = -\frac{s^T (r-s)}{(r-s)^T (r-s)} \\
s = s + \gamma (r - s) \\
z = z + \gamma (x - z)
\end{array}
\right\} \ MR\ smoothing \\
\\
\left.
\begin{array}{l}
\rho_{old} = \rho \\
\rho = r^T r^* \\
\beta = \frac{\rho}{\rho_{old}} \\
p = r + \beta p \\
p^* = r^* + \beta p^*
\end{array}
\right\} \ BCG
\end{array}
\right\} \ BICO
$$

$$
restart : \left\{
\begin{array}{l}
x = z \\
r = r^* = p = p^* = s \\
\rho = r^T r^*
\end{array}
\right.
$$

Use as approximation

$$
\begin{array}{ll}
x & for\ BCG, \\
z & for\ BICO.
\end{array}
$$

Algorithm 8: BCG and BICO

4.2.3 QMR

The quasi-minimal residual method (QMR) was developed by Freund and Nachtigal [37]. QMR describes a whole family of methods depending on parameters $\omega_0, \ldots, \omega_{n-1}$. Like BCG the method can be considered as method generated by the Lanczos algorithm [68]. Zhou and Walker [139] show that quasi-residual smoothing transforms BCG into QMR.

Choose an initial guess $x = z$, $r = s = r^* = p = p^* = Ax - b$, $\tau^2 = \|r\|^2$, $\rho = r^T r^*$ *and calculate in each iteration step*

$$
\left.\begin{array}{l}
\left.\begin{array}{l}
f = Ap \\
f^* = A^T p^* \\
if\ p^T f^* \neq 0\ and\ \rho \neq 0\ then \\
\qquad \delta = -\frac{r^T r^*}{p^T f^*} \\
else \\
\qquad restart \\
r = r + \delta f \\
r^* = r^* + \delta f^* \\
x = x + \delta p
\end{array}\right\} BCG \\
\\
\left.\begin{array}{l}
\tau^2 = \frac{1}{\frac{1}{\tau^2} + \frac{1}{||r||^2}} \\
\gamma = \frac{\tau^2}{||r||^2}, \\
s = s + \gamma(r - s) \\
z = z + \gamma(x - z)
\end{array}\right\} QMR\ smoothing \\
\\
\left.\begin{array}{l}
\rho_{old} = \rho \\
\rho = r^T r^* \\
\beta = \frac{\rho}{\rho_{old}} \\
p = r + \beta p \\
p^* = r^* + \beta p^*
\end{array}\right\} BCG
\end{array}\right\} QMR
$$

$$
restart : \left\{\begin{array}{l}
x = z \\
r = r^* = p = p^* = s \\
\rho = r^T r^*
\end{array}\right.
$$

Use as approximation

$$
\begin{array}{ll}
x & for\ BCG, \\
z & for\ QMR.
\end{array}
$$

Algorithm 9: BCG and QMR

If we omit the look-ahead process that is proposed in the original paper by Freund and Nachtigal and use $\omega_0, \ldots, \omega_{n-1} = 1$, then we can formulate the algorithm quite simply (see Algorithm 9). The algorithm requires two matrix-vector multiplications per iteration step, one by A and one by A^T.

In LINSOL	
`ilin(1) = ms = 6` *and* `ilin(18) = smooth = 0`	*BCG*
`ilin(1) = ms = 6` *and* `ilin(18) = smooth < 0`	*QMR*

4.2.4 CGS

CGS (Conjugate Gradient Squared) (see Algorithm 10) was introduced by Sonneveld [110]. The idea is to square the BCG matrix polynomial Π_k (4.11) and to obtain

$$e_k = \Pi_k^2(A)e_0, \tag{4.17}$$

$$r_k = \Pi_k^2(A)r_0. \tag{4.18}$$

If Π_k tends to zero, then a faster convergence is obtained by Π_k^2. Indeed, in many cases CGS converges twice as fast as BCG, but the convergence may be more erratic than for BCG. CGS has the additional advantage that no matrix-vector multiplication by A^T is necessary. Instead two matrix-vector multiplications by A are necessary. Thus it is suited for problems, where the matrix A and, therefore, A^T are not explicitely available or where on massively parallel computers A^T cannot be easily accessed even if A is available. The algorithm requires two matrix-vector multiplications by A per iteration step.

In LINSOL	
`ilin(1) = ms = 5` *and* `ilin(18) = smooth = 0`	*CGS*
`ilin(1) = ms = 5` *and* `ilin(18) = smooth > 0`	*CGS smoothed*

4.2.5 BiCGSTAB

BiCGSTAB (BiConjugate Gradients STABilized) was introduced by van der Vorst [118] (see Algorithm 11). The idea is to substitute the squared matrix polynomial Π_k^2 of CGS, see (4.17) and (4.18), by the product of a matrix polynomial Ψ_k by Π_k in order to get a smoother convergence and to stabilize the iteration. Thus, we get

$$e_k = \Psi_k(A)\Pi_k(A)e_0, \tag{4.19}$$

$$r_k = \Psi_k(A)\Pi_k(A)r_0. \tag{4.20}$$

Ψ_k is recursively defined by

$$\Psi_k(A) = (\gamma_k A + I)\Psi_{k-1}(A), \tag{4.21}$$

Choose an initial guess $x = z$, $r = Ax - b$, choose $r^* = s = r$, set $\rho = r^T r^*$, $p = u = r$ and calculate in each iteration step

$$
\left.\begin{array}{l}
\left.\begin{array}{l}
v = Ap \\
if\ v^T r^* \neq 0\ and\ \rho \neq 0\ then \\
\qquad \delta = -\frac{\rho}{v^T r^*} \\
else \\
\qquad restart \\
q = 2u + \delta v \\
t = Aq \\
r = r + \delta t \\
x = x + \delta q
\end{array}\right\} \ CGS \\
\\
\left.\begin{array}{l}
\gamma = -\frac{s^T (r-s)}{(r-s)^T (r-s)} \\
s = s + \gamma (r - s) \\
z = z + \gamma (x - z)
\end{array}\right\} \ MR\ smoothing \\
\\
\left.\begin{array}{l}
\rho_{old} = \rho \\
\rho = r^T r^* \\
\beta = \frac{\rho}{\rho_{old}} \\
u = r + \beta (q - u) \\
p = 2u - r + \beta^2 p
\end{array}\right\} \ CGS
\end{array}\right\} \ \begin{array}{l} MR\ smoothed \\ CGS \end{array}
$$

$$
restart : \left\{ \begin{array}{l}
x = z \\
r = r^* = p = u = s \\
\rho = r^T r^*
\end{array}\right.
$$

Use as approximation

x *for CGS,*
z *for CGS smoothed.*

Algorithm 10: CGS and CGS smoothed

where γ_k is determined from a 1-dimensional minimization

$$\|r_k\| = \min_{\gamma_k} \|(\gamma_k A + I)\Psi_{k-1}(A)\Pi_k(A) r_0\|. \tag{4.22}$$

The algorithm requires two matrix-vector multiplications by A per iteration step.

Gutknecht [53], Sleijpen and Fokkema [108] and Sleijpen, van der Vorst and Fokkema [109] generalized BiCGSTAB so that Ψ_k is determined by an ℓ-dimensional minimization. These methods are called BiCGSTAB(ℓ). Other generalizations called product

Choose an initial guess $x = z$, $r = Ax - b$, choose $r^* = s = r$, set $\rho = r^T r^*$, $p = r$ and calculate in each iteration step

$$
\left.\left.\begin{array}{l}
v = Ap \\
if\ v^T r^* \neq 0\ and\ \rho \neq 0\ then \\
\qquad \delta = -\frac{\rho}{v^T r^*} \\
else \\
\qquad restart \\
q = r + \delta v \\
t = Aq \\
\alpha = -\frac{q^T t}{\|t\|^2} \\
r = q + \alpha t \\
x = x + \delta p + \alpha q
\end{array}\right\} BiCGSTAB \\
\left.\begin{array}{l}
\gamma = -\frac{s^T(r-s)}{(r-s)^T(r-s)} \\
s = s + \gamma(r - s) \\
z = z + \gamma(x - z)
\end{array}\right\} MR\ smoothing \\
\left.\begin{array}{l}
\rho_{old} = \rho \\
\rho = r^T r^* \\
\beta = \frac{\rho}{\rho_{old}} \frac{\delta}{\alpha} \\
p = r + \beta(p + \alpha v)
\end{array}\right\} BiCGSTAB
\right\} \begin{array}{l} MR\ smoothed \\ BiCGSTAB \end{array}
$$

$$
restart : \begin{cases} x = z \\ r = r^* = p = s \\ \rho = r^T r^* \end{cases}
$$

Use as approximation

x *for BiCGSTAB,*
z *for BiCGSTAB smoothed.*

Algorithm 11: BiCGSTAB and BiCGSTAB smoothed

methods were given by Brezinski and Redivo-Zaglia [11] and generalized product-type methods by Zhang [138]. Chan, Gallopoulos, Simoncini, Szeto and Tong applied quasi-minimal residual smoothing to BiCGSTAB and obtained a smoother converging method

QMRCGSTAB [16].

In LINSOL	
`ilin(1) = ms = 3` *and* `ilin(18) = smooth = 0`	*BiCGSTAB*
`ilin(1) = ms = 3` *and* `ilin(18) = smooth > 0`	*BiCGSTAB smoothed*

4.3 Normal Equations

Let us consider the two linear systems

$$AA^T y = b, \tag{4.23}$$
$$A^T A x = A^T b. \tag{4.24}$$

They are called normal equations. The solution of (4.24) is the solution of (2.1) and we get the solution of the original system x from (4.23) by

$$x = A^T y. \tag{4.25}$$

The matrices AA^T and $A^T A$ are symmetric and positive definite and, therefore, classical CG can be applied to the normal equations. We obtain the two methods CGNE and CGNR.

The speed of convergence may be very slow if the eigenvalues of the iteration matrix AA^T are more scattered than those of the original system matrix A. The condition number is squared, i. e. the convergence is slow for matrices close to symmetric matrices, but in the general case this is not true. Theoretically the methods converge always. Thus, they are very robust but not very efficient.

4.3.1 CGNE

If we apply classical CG to the normal equations (4.23)

$$AA^T y = b,$$

then we get the solution of the original system x by

$$x = A^T y.$$

For the errors of the original system

$$||e_k|| = \min_{\beta_1 \ ,..,\beta_k} || \sum_{i=1}^{k} \beta_i \left(A^T A\right)^i e_0 + e_0|| \tag{4.26}$$

is valid. Thus, the method minimizes the error in the Euclidean norm and is called CGNE (CG Normal equations Error-minimizing). Reformulated to the approximations and residuals of the original system it was proposed by Craig [19]. The algorithm will be given in the description of CGNR-ATPRES because both methods can be implented simultaneously (see Algorithm 12).

4.3.2 CGNR-ATPRES

If we apply classical CG to the normal equations (4.24)

$$A^T Ax = A^T b,$$

then we get for the residuals of the original system

$$||r_k|| = \min_{\beta_1 ,..,\beta_k} ||\sum_{i=1}^{k} \beta_i \left(AA^T\right)^i r_0 + r_0||. \tag{4.27}$$

Indeed, the method can be reformulated such that only the approximations and residuals of the original system are used. This method is called CGNR (CG Normal equations Residual minimizing). There are various implementations. We give the ATPRES (*A* Transposed Pseudo RESidual method) [100] implementation that results from applying residual-minimizing smoothing to CGNE (see Algorithm 12). The algorithms require two matrix-vector multiplications per iteration step, one by A and one by A^T.

In LINSOL
`ilin(1) = ms = 4` *and* `ilin(18) = smooth = 0` *CGNE*
`ilin(1) = ms = 4` *and* `ilin(18) = smooth > 0` *ATPRES*

4.4 Overview

The essential operations for the here discussed methods are matrix-vector multiplications, dot products and triadic operations. For realistic problems the work for the matrix-vector multiplication dominates the work for other operations. The required storage is depending on the recurrence length and is a crucial point for large systems, i. e. the value of restart and truncation parameters, denoted simply by σ, may not be arbitrary large. The matrix-vector multiplication is optimally vectorizable and parallelizable, thus the methods are optimally suited for today's supercomputers.

In table 4.1 the different methods are classified with respect to work and storage requirements and in table 4.2 with respect to the particular minimization properties. Note that in LINSOL methods that can be transformed by smoothing are implemented as one algorithm so that the storage requirements always include the two additional

Choose an initial guess $x_0 = z$, $r_0 = s = Ax_0 - b$, $k = 0$, $old = 1$, $new = 0$, $x_{new} = 0$, $r_{new} = 0$ *and calculate in each iteration step*

$$\left.\begin{array}{l}
\left.\begin{array}{l}
k = k + 1 \\
h = old,\ \ old = new,\ \ new = h \\
d = A^T r_{old} \\
f = Ad \\
\alpha_{old} = -\frac{r_{old}^T f}{\|r_{old}\|^2} \\
if\ k \geq 2\ then \\
\qquad \alpha_{new} = -\frac{r_{new}^T f}{\|r_{new}\|^2} \\
else \\
\qquad \alpha_{new} = 0 \\
\phi = \frac{1}{\alpha_{old} + \alpha_{new}} \\
x_{new} = \phi(d + \alpha_{old} x_{old} + \alpha_{new} x_{new}) \\
r_{new} = \phi(f + \alpha_{old} r_{old} + \alpha_{new} r_{new})
\end{array}\right\} CGNE \\
\\
\left.\begin{array}{l}
\gamma = -\frac{s^T (r_{new} - s)}{(r_{new} - s)^T (r_{new} - s)} \\
s = s + \gamma (r_{new} - s) \\
z = z + \gamma (x_{new} - z)
\end{array}\right\} MR\ smoothing
\end{array}\right\} ATPRES$$

Use as approximation

$$\begin{array}{ll} x_{new} & \text{for CGNE,} \\ z & \text{for ATPRES.} \end{array}$$

Algorithm 12: CGNE and ATPRES

vectors needed for smoothing. Moreover, 2 additional buffer vectors are required for a multiplication by A and 3 buffer vectors for a multiplication by A^T. For BCG, BICO and QMR the multiplications by A and by A^T are performed simultaneously so that 5 buffer vectors are needed.

In LINSOL

Dimension `ldw` *of the array* `dw`

`ldw =`	`15`	*for*	*ORTHORES-R(5), ORTHORES-T(5), PRES20, GMRES-R(5), GMRES-T(5)*
	`18`	*for*	*GMERR(5)*
	`14`	*for*	*BCG, BICO, QMR,*
	`11`	*for*	*CGS, CGS-smoothed, BiCGSTAB, BiCGSTAB smoothed*
	`12`	*for*	*CGNE, ATPRES*

method	mvm	st
ORTHORES-R(σ)	1	$2\sigma+1$
ORTHORES-T(σ)	1	$2\sigma+1$
PRES20	1	13
GMRES-R(σ)	1	$2\sigma+3$
GMRES-T(σ)	1	$2\sigma+3$
GMERR(σ)	1	$2\sigma+3$
BCG	2	7
BICO	2	9
QMR	2	9
CGS	2	7
CGS smoothed	2	9
BiCGSTAB	2	7
BiCGSTAB smoothed	2	9
CGNE	2	7
ATPRES	2	9

mvm = matrix-vector multiplications per iteration, st = number of auxiliary vectors of length n

Table 4.1: Comparison of work and storage

	original system	normal equations
error-minimizing	GMERR	CGNE
energy norm-minimizing	ORTHORES	
residual-minimizing	GMRES	CGNR-ATPRES

Table 4.2: Comparison of minimization properties for CG methods and GMERR

Numerical tests comparing all these methods are presented in [103].

The BCG-based methods are closely related because they have the same origin: BCG. Numerical tests confirm a similar behavior of these methods; see [103]. The BCG and CGS residuals and errors oscillate heavily while all other BCG-based methods (BICO, QMR, and BiCGSTAB) produce a smoother convergence. Therefore BICO, QMR or BiCGSTAB should be applied.

Note that ORTHORES and GMRES methods can be computed in parallel without additional costs by smoothing, see [124], and that GMRES minimizes the residuals in the Euclidean norm while ORTHORES minimizes the error in the energy norm. GMRES guarantees monotonicly decreasing residuals. ORTHORES may be better as regards the decrease of the error.

PRES20, the BCG derivates, CGNE and CGNR-ATPRES use short recurrences, ORTHORES and GMRES use long recurrences for large restart and truncation parameters. Note that BCG and ORTHORES as well as GMRES, QMR and BICO coincide for symmetric matrices and simplify so that they all can be formulated with a short recurrence.

Chapter 5

Polyalgorithms

In order to get an automatic selection of an appropriate solver there have been many attempts to construct hybrid methods or polyalgorithms. It would have been impossible to construct such algorithms considering all here discussed methods – or the quite larger set of methods described in literature. A key for the development of these techniques is the perception that iterative methods can be represented by a quite smaller set.

We can derive from practical experience the following rules of thumb. ORTHORES and GMRES are similar methods and converge for small restart and truncation parameters if the eigenvalues of A are close to the real axis, see figure 5.1. A special optimized version for this case is PRES20. BICO and QMR converge if the eigenvalues are in the positive half plane, see figure 5.2. Then they are superior to ORTHORES and GMRES with small restart and truncation parameters. The BiCGSTAB versions behave similar, but still perform very well when ORTHORES and GMRES are better than BICO and QMR. CGNR-ATPRES and CGNE are the best methods for scattered eigenvalues, however they are rather slow for systems where the eigenvalues are close to the real axis, see figure 5.3.

From the above discussion it seems to be reasonable to select one representative method of the truncated or restarted CG methods, one representative BCG-based method and one of the methods based on the normal equations. Some of the adaptive selection schemes estimate the spectrum of A and select the solver according to this information. The here implemented polyalgorithms proposed in [93, 99, 105] do not need information about inner properties of the matrix. They monitor the norm of the residuals and switch from efficient but less robust methods to more robust but less efficient methods.

The polyalgorithms I (see Algorithm 13) and II (see Algorithm 14) start with PRES20. If the reduction of the residual is not sufficient after certain steps, then BICO is applied. If BICO converges too slowly the algorithm switches to the emergency exit ATPRES (Algorithm I) or CGNE (Algorithm II). The polyalgorithms III (see Algorithm 15) and IV (see Algorithm 16) start with GMERR(5) . If the reduction of the residual is not

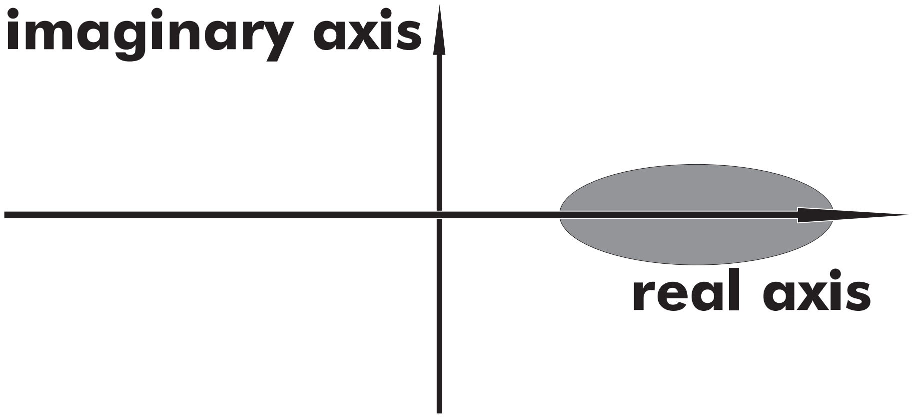


Figure 5.1: Domain of eigenvalues, where restarted or truncated ORTHORES and GMRES methods are efficient

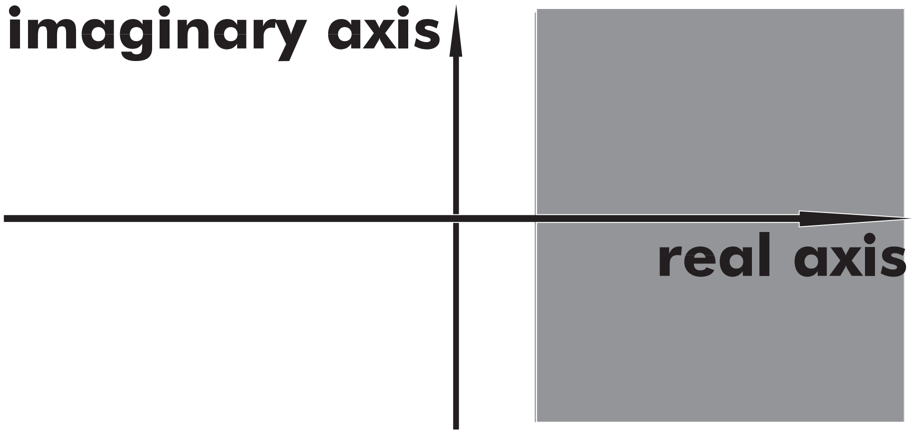


Figure 5.2: Domain of eigenvalues, where methods resulting from BCG are efficient

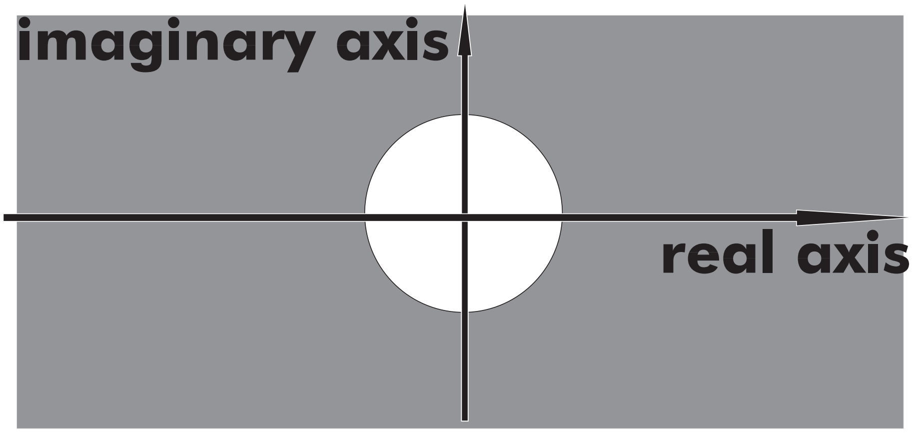


Figure 5.3: Domain of eigenvalues, where methods based on the normal equations have to be applied

Start with	*PRES20*	*if* $k \geq 20$ *and* $\frac{\|r_k\|}{\|r_0\|} \geq 0.5$ *switch to*
	BICO	*if* $k \geq 1000$ *and* $\frac{\|r_k\|}{\|r_0\|} \geq 0.1$ *switch to*
	ATPRES	

Algorithm 13: Polyalgorithm I

Start with	*PRES20*	*if* $k \geq 20$ *and* $\frac{\|r_k\|}{\|r_0\|} \geq 0.5$ *switch to*
	BICO	*if* $k \geq 1000$ *and* $\frac{\|r_k\|}{\|r_0\|} \geq 0.1$ *switch to*
	CGNE	

Algorithm 14: Polyalgorithm II

sufficient after certain steps, then BiCGSTAB is applied. If BiCGSTAB converges too slowly the algorithm switches to the emergency exit ATPRES (Algorithm III) or CGNE (Algorithm IV).

In LINSOL

`ilin(1) = ms = 10` *and* `ilin(18) = smooth > 0` *polyalgorithm I*
`ilin(1) = ms = 20` *and* `ilin(18) = smooth > 0` *polyalgorithm II*
`ilin(1) = ms = 30` *and* `ilin(18) = smooth > 0` *polyalgorithm III*
`ilin(1) = ms = 40` *and* `ilin(18) = smooth > 0` *polyalgorithm IV*

Dimension `ldw` *of the array* `dw`

`ldw = 15` *for Polyalgorithms I and II*
`ldw = 18` *for Polyalgorithms III and IV*

In order to demonstrate how polyalgorithm I works we execute it for the solution of

Start with	*GMERR(5)*	*if* $k \geq 20$ *and* $\frac{\|r_k\|}{\|r_0\|} \geq 0.5$ *switch to*
	BiCGSTAB smoothed	*if* $k \geq 1000$ *and* $\frac{\|r_k\|}{\|r_0\|} \geq 0.1$ *switch to*
	ATPRES	

Algorithm 15: Polyalgorithm III

Start with	*GMERR(5)*	*if* $k \geq 20$ *and* $\frac{\|\|r_k\|\|}{\|\|r_0\|\|} \geq 0.5$ *switch to*
	BiCGSTAB	*if* $k \geq 1000$ *and* $\frac{\|\|r_k\|\|}{\|\|r_0\|\|} \geq 0.1$ *switch to*
	CGNE	

Algorithm 16: Polyalgorithm IV

the model PDE

$$\Delta v + v + R(v^T \nabla) v = h \ , \tag{5.1}$$

with solution

$$v = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} sin2\pi x \; cos2\pi y \; cos2\pi z \\ cos2\pi x \; sin2\pi y \; cos2\pi z \\ cos2\pi x \; cos2\pi y \; sin2\pi z \end{pmatrix} .$$

We prescribe Dirichlet boundary conditions, i.e. v_1, v_2, v_3 on the unit cube. We generate A and b by the FIDISOL program package [102] for the consistency orders 4 in x, y, z-direction on a grid 40 x 40 x 40 (n = 192 000). The initial values for v_1, v_2, v_3 are mean values of interpolations between the bondary values in x, y, z-direction. For these values the matrix A of the first Newton step is used as test-matrix. We force LINSOL to reduce the L_2-norm of the relative residual $||r_k||/||r_0||$ of the normalized equations (2.10) to be reduced to 10^{-20}. In the Figure 5.4 we depict the log of the relative residual as a function of the MVMs (matrix-vector multiplications). Note that PRES20 needs one MVM, but BICO and ATPRES need two MVMs per iteration step. The MVMs are a good measure for the computation. For $R = 1$ and $R = 10$ the polyalgorithm follows PRES20, for $R = 100$ the polyalgorithm is between PRES20 and BICO, for $R = 1000$ the polyalgorithm follows BICO and for $R = 10000$ it follows nearly ATPRES. Thus the polyalgorithm with the above mentioned strategy follows nearly the best method. This is what we can call making "intelligent" software by numerical engineering.

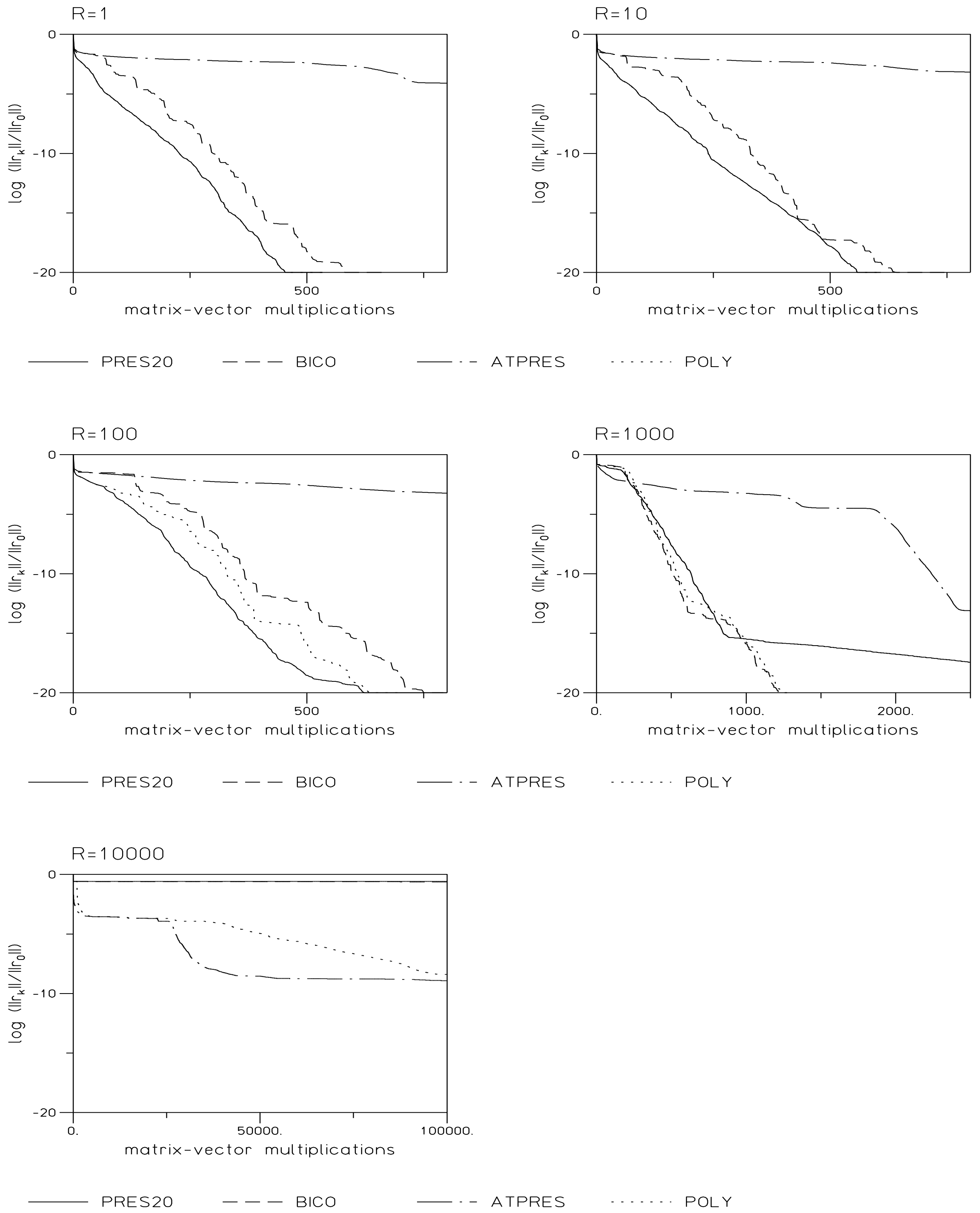


Figure 5.4: Polyalgorithm I for the sample problem with $R = 1$, $R = 10$, $R = 100$, $R = 1000$ and $R = 10000$

Chapter 6

Preconditioning

The idea of preconditioning is to consider a related linear system with the same solution as the original problem, which can be easier solved. In fact today there is no efficient linear solver without any preconditioning. The problem is that each preconditioning technique is suited for a different type of problem. Up to now we know of no robust preconditioning technique, i. e. a technique that is successfully applicable for all or at least most type of problems.

We will define constant and step-depending preconditioning and apply the results for the convergence of orthogonalization methods to preconditioned systems yielding to preconditioned methods. The most popular preconditioned techniques are surveyed subsequently.

Instead of solving the linear system (2.1) it may be convenient to solve the preconditioned system

$$P_L A P_R y = P_L b \ . \tag{6.1}$$

$P_L \in \mathbb{R}^{n \times n}$ is a non-singular left-hand preconditioning matrix and $P_R \in \mathbb{R}^{n \times n}$ is a non-singular right-hand preconditioning matrix. The solution of (2.1) is obtained from the solution of (6.1) by

$$x = P_R y \ . \tag{6.2}$$

The left-hand preconditioning matrix P_L acts on the equations, while the right-hand preconditioning matrix P_R acts on the solution.

The optimal choice of P_L, P_R would be

$$P_R P_L = A^{-1} \ , \tag{6.3}$$

because then the solution is obtained directly by

$$y = P_L b \ , \tag{6.4}$$

$$x = P_R P_L b = A^{-1} b \ . \tag{6.5}$$

Of course the solution of (6.3) is a more difficult task than the solution of (2.1). But all preconditioning techniques try to choose P_L and P_R so that $P_L P_R$ is close to A^{-1} or that $P_L A P_R$ is closer to the unit matrix than the matrix A itself.

It is also possible to choose step-dependent preconditioners. The idea is to improve the preconditioning matrices in each iteration step together with the approximations.

Preconditioning from the left-hand side is as usual as preconditioning from the right-hand side. For both techniques the same construction principles for the preconditioning matrices are used. In practice all three cases apply:

- preconditioning by P_L, while $P_R = I$,
- preconditioning by P_R, while $P_L = I$ and
- preconditioning by both: P_L and P_R.

For many methods the preconditioning matrix must not be explicitly calculated. It is sufficient to know how it acts on a vector. Techniques that are based on this proceeding are the polynomial preconditioners, the incomplete LU decomposition and the incomplete Cholesky method.

Techniques that calculate preconditioning matrices explicitly construct incomplete inverses, i. e. matrices that have a certain sparsity pattern and approximate the inverse of A. This proceeding has the advantage that during the iteration there is only one additional matrix-vector multiplication per iteration step that can be optimally vectorized and parallelized, see e. g. [93, 128]. There are investigations on the structural properties of band matrices [22, 84] and their inverses. Techniques based on this proceeding are the normalization, the incomplete Gauss-Jordan method, the incomplete^2 Gauss-Jordan method, the incomplete Frobenius inverses and projection methods.

In the next section we describe the main features and ideas of special preconditioners without giving details in order to get an overview, see also [3]. We write in the sequel simply P for the preconditioning matrices, if the distinct technique is equally applicable to right-hand and left-hand preconditioning matrices.

In LINSOL
Presently, only normalization and preconditioning by the normal equations is implemented.

6.1 Normalization

Normalization discussed in the first chapter "Iterative Methods" can be considered as preconditioning.

6.2 Polynomial Preconditioners

The idea is to choose the preconditioning matrices so that

$$P = \sum_{i=1}^{j} \theta_i A^i + \theta_0 I \; . \tag{6.6}$$

There are various possibilities for the choice of j and θ_i; see [32, 45, 64, 85]. If for example A is symmetric and the minimal and maximal eigenvalue have been estimated or are known, a parameter-dependent preconditioner can be constructed by choosing P as Chebyshev polynomial on an interval enclosing the eigenvalues. Eiermann, Li and Varga give an encompassing theoretical analysis [25] for more general cases. Fischer and Freund propose an adaptive weighted polynomial preconditioning [30].

Polynomial preconditioners can be optimally vectorized and parallelized, but j matrix-vector multiplications by A have to be performed in each step.

In LINSOL
not implemented

6.3 Normal Equations

The normal equations

$$A^T A x = A^T b \tag{6.7}$$

can be considered as preconditioning by A^T from the left-hand side and

$$A A^T y = b \tag{6.8}$$

can be considered as preconditioning by A^T from the right-hand side. Corresponding CG methods are CGNR and Craig's method CGNE .

6.4 Incomplete LU-decomposition (ILU)

For sparse systems with a certain structure resulting e. g. from the finite difference and finite element method a common technique is to use an incomplete LU-decomposition (ILU). A theoretical investigation is given by Il'in [63] and by Wittum and Liebau [134]. Let L be a lower triangular matrix, U be an upper triangular matrix. Decompose A according to

$$A = LU + R \tag{6.9}$$

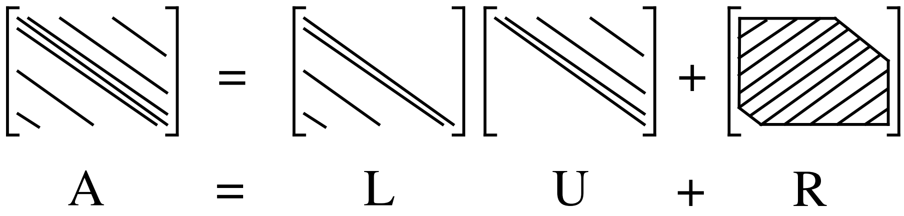


Figure 6.1: Incomplete LU-decomposition with the pattern of A

with

$$r_{k,l} = \begin{cases} 0 & if \quad (k,l) \in J \\ a_{k,l} - \sum_{j=1}^{min(k,l)} l_{k,j} u_{j,l} & if \quad (k,l) \notin J \end{cases} . \tag{6.10}$$

The idea is to neglect R and to appoximate A by an incomplete factorization. Usually the set J is the pattern of the matrix A, i. e. $J = \{(k,l) | a_{k,l} \neq 0\}$, see figure 6.1, or determined by an adaptive process monitoring the size of $r_{k,l}$. Thus, we have the two techniques to decompose by position or by value. The algorithm for the determination of L and U is highly recursive. This decomposition can be used for left-hand preconditioning

$$U^{-1} L^{-1} A x = U^{-1} L^{-1} b,$$

for right-hand preconditioning

$$A U^{-1} L^{-1} y = b,$$

or for preconditioning from both sides

$$L^{-1} A U^{-1} y = L^{-1} b.$$

Clearly, L and U will be never inverted, but triangular systems will be solved whenever the evaluation of $L^{-1}z$ or $U^{-1}z$ is required for any vector z. During each iteration step two triangular systems have to be solved. This solution is vectorizable but the vectorlength varies preventing an efficient vectorization. Very elaborate strategies for an efficient implementation on vector machines for special matrix structures were proposed by Fujino [40], by Fujino, Mori and Takeuchi [42], by Fujino and Takeuchi [43] and by Yamamoto and Takahashi [135].

The algorithm has been modified in order to achieve more efficiency to

- MILU (modified ILU) by Gustafsson [50]. The rowsum of R is added to the main diagonal of LU, i. e. the dropped values are saved on the main diagonal.

- RILU (relaxed ILU) by Axelsson and Lindskog [4]. A fraction α of the rowsum of R with $0 \leq \alpha \leq 1$ is added to the main diagonal of LU.

- ILUT (ILU threshold) by Saad [88] is a compromise between decomposition by position and by value in order to achieve robustness and efficiency at a time.
- ILUM (ILU multi-elimination) by Saad [87] an optimized version for parallel computers.
- S/P ordering techniques by Beauwens [6, 8, 7] and by Notay [76, 75], where columns or rows of the matrix are interchanged in order to achieve a better convergence.

An excellent investigation of different basic methods in combination with different ILU strategies is given by Meier Yang [71].

In LINSOL
not yet implemented

6.5 Incomplete Cholesky (IC)

If A is symmetric and positive definite, an incomplete Cholesky decomposition (IC) can be applied instead of the LU-decomposition. The advantage is that this algorithm is more stable and needs only half of the operation of incomplete LU. The IC method is the basis for ILU and was introduced by Buleev (in "close" research reports) and published in the book of Marchuk [70]. It seems that Varga [119] and Oliphant [77] did independently research on the same method. The algorithm was investigated by Meijerlink and van der Vorst [72, 114, 115]. Let C be an upper triangular matrix. Decompose A according to

$$A = C^T C + R \tag{6.11}$$

with

$$r_{k,l} = \begin{cases} 0 & if \quad (k,l) \in J \\ a_{k,l} - \sum_{j=1}^{min(k,l)} c_{j,k} c_{j,l} & if \quad (k,l) \notin J \end{cases} . \tag{6.12}$$

As for ILU R is neglected and A is approximated by an incomplete factorization. Usually the set J is the pattern of the matrix A, i. e. $J = \{(k,l) | a_{k,l} \neq 0\}$, or determined by an adaptive process monitoring the size of $r_{k,l}$. During each iteration step two triangular systems have to be solved.

The algorithm for the determination of C is highly recursive. Fujino and Doi [41], Jones and Plassmann [65] achieve efficient implementations on vector computers for special matrix structures by multicoloring, van der Vorst [116, 117] by hyperplane orderings.

If IC is used for preconditioning from both sides

$$C^{-T}AC^{-1}y = C^{-T}b,$$

then the symmetry of the system matrix will be transferred to the preconditioned system because

$$(C^{-T}AC^{-1})^T = C^{-T}AC^{-1}$$

if A is symmetric. This has the advantage that many iterative methods simplify or can be formulated with a short recurrence.

In LINSOL
not yet implemented

6.6 Incomplete QR (IQR)

Let Q be an orthonormal and R be an upper triangular matrix. Decompose A according to

$$A = QR + E \tag{6.13}$$

with

$$e_{k,l} = \begin{cases} 0 & if \quad (k,l) \in J \\ a_{k,l} - \sum_{j=1}^{n} q_{k,j} r_{j,l} & if \quad (k,l) \notin J \end{cases} . \tag{6.14}$$

The remainder E is neglected as it is done for ILU and IC. Thus the matrix is approximated by an incomplete factorization. Usually the set J is the pattern of the matrix A, i. e. $J = \{(k,l) | a_{k,l} \neq 0\}$, or determined by an adaptive process monitoring the size of $e_{k,l}$. If we substitute Q^{-1} by Q^T then the decomposition can be used for left-hand preconditioning

$$R^{-1}Q^T Ax = R^{-1}Q^T b,$$

for right-hand preconditioning

$$AR^{-1}Q^T y = b,$$

or for preconditioning from both sides

$$Q^T AR^{-1} y = Q^T b.$$

In this form the method is not yet applicable to large and sparse systems because Q is in general a full matrix. The size of Q would exceed the storage capabilities even of supercomputers. Therefore, we substitute Q^T by $R^{-T}A^T$ that would be an equivalence if $A = QR$. Thus, we get for example as right-hand preconditioner

$$P = R^{-1}R^{-T}A^T. \tag{6.15}$$

In each iteration step two triangular systems (with R and with R^T) have to be solved. A special IQR decomposition was used by Portugal, Bastos, Júdice, Paixão and Terlaky for the solution of the linear transportation problem by interior point algorithms [81]. There exist investigations for the sparsity patterns of QR decompositions for sparse matrices [60], thus justifying the incomplete approach. A QR decomposition of A corresponds to a Cholesky decomposition of A^TA because

$$A^TA = R^TQ^TQR = R^TR. \tag{6.16}$$

Thus, an incomplete QR algorithm is equivalent to an incomplete Cholesky decomposition of the normal equations, but the calculation of A^TA which can be prohibitive because of storage reasons is not necessary. Moreover, there are different dropping strategies possible. For example instead of monitoring the size of $e_{k,l}$ for the adaptive choice of J, a pattern for Q and R can be prescribed, so that the smallest entries are dropped. Such a decomposition was proposed by Saad [86]. There are modifications based on the calculation of

$$A\tilde{R} = Q + E. \tag{6.17}$$

This technique has the advantage that by using as for example right-hand preconditioner

$$P = \tilde{R}\tilde{R}^TA^T \tag{6.18}$$

no triangular system has to be solved. Additional matrix-vector multiplications by $\tilde{R}$, $\tilde{R}^T$ and A^T are necessary that can be optimally vectorized and parallelized.

In LINSOL
not yet implemented

6.7 Incomplete Gauss-Jordan

The Gauss-Jordan algorithm calculates the inverse of A by constructing a sequence of matrices:

$$A^{(0)} = A\ ,$$

$$
\begin{aligned}
P^{(0)} &= I \ , \\
A^{(i)} &= C^{(i)} A^{(i-1)} \ , \\
P^{(i)} &= C^{(i)} P^{(i-1)}
\end{aligned}
$$

for $i = 1, \ldots, n$. The matrices $C^{(i)}$ are chosen so that zeros are generated in the off-diagonal elements of the column i of $A^{(i)}$:

$$
A^{(i)} = \begin{pmatrix}
a_{1,1}^{(i)} & 0 & \ldots & 0 & a_{1,i+1}^{(i)} & \ldots & a_{1,n}^{(i)} \\
0 & \ddots & \ddots & \vdots & \vdots & & \vdots \\
\vdots & \ddots & \ddots & 0 & \vdots & & \vdots \\
\vdots & & \ddots & a_{i,i}^{(i)} & \vdots & & \vdots \\
\vdots & & & 0 & \vdots & & \vdots \\
\vdots & & & \vdots & \vdots & & \vdots \\
0 & \ldots & \ldots & 0 & a_{n,i+1}^{(i)} & \ldots & a_{n,n}^{(i)}
\end{pmatrix} .
$$

$A^{(n)}$ is a diagonal matrix and

$$
\begin{aligned}
A^{(n)} &= C^{(n)} \cdot \ldots \cdot C^{(1)} A \ , \\
P^{(n)} &= C^{(n)} \cdot \ldots \cdot C^{(1)} \ , \\
A^{-1} &= A^{(n)-1} P^{(n)} \ .
\end{aligned}
$$

The incomplete Gauss-Jordan algorithm constructs a sequence of matrices preserving a certain pattern in $P^{(i)}$ and $A^{(i)}$ in the following way: Let u_i be the i-th unit vector and

$$
\begin{aligned}
A^{(0)} &= A \ , \\
P^{(0)} &= I \ , \\
C^{(i)} &= I + (I - \frac{1}{a_{i,i}^{(i-1)}} A^{(i-1)}) u_i u_i^T \ ,
\end{aligned}
$$

$$
a_{k,l}^{(i)} = \begin{cases} \sum_{j=1}^{n} c_{k,j}^{(i)} a_{j,l}^{(i-1)} & if \quad (k,l) \in J \\ 0 & if \quad (k,l) \notin J \end{cases} \ , \tag{6.19}
$$

$$
p_{k,l}^{(i)} = \begin{cases} \sum_{j=1}^{n} c_{k,j}^{(i)} p_{j,l}^{(i-1)} & if \quad (k,l) \in J \\ 0 & if \quad (k,l) \notin J \end{cases} \tag{6.20}
$$

for $i = 1, \ldots, n$. Usually the set J is the pattern of the matrix A, i. e. $J = \{(k,l) | a_{k,l} \neq 0\}$, or consisting of more elements. The first i columns of $A^{(i)}$ only have diagonal entries. Therefore, $A^{(n)}$ is a diagonal matrix. Choose as preconditioning matrix

$$
P = A^{(n)-1} \cdot P^{(n)} \tag{6.21}
$$

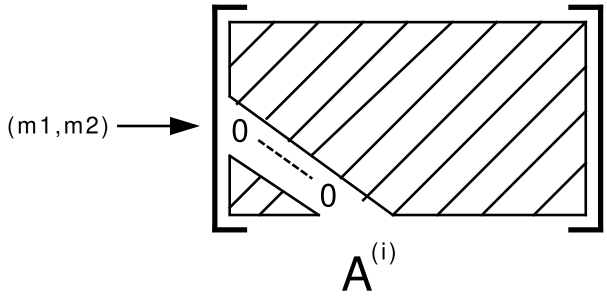


Figure 6.2: Generation of zero-diagonals in $A^{(i)}$ by the incomplete² Gauss-Jordan method

The calculation of the incomplete inverse according to this algorithm is highly recursive. The number of operations for the determination of the matrix P is $\frac{\sharp J^2}{n}$ corresponding to $\frac{\sharp J}{n}$ matrix-vector multiplications where $\sharp J$ is the number of elements in J. If all indices are in J we obtain the Gauss-Jordan algorithm resulting in $P = A^{-1}$.

The incomplete² Gauss-Jordan algorithm is a modification of the incomplete Gauss-Jordan algorithm, i. e. the recursion for the determination of the incomplete inverse is dropped and vectorization is forced [133]. One zero diagonal in the matrix $A^{(i)}$ is produced in order to obtain an excellent vectorization for matrices having a diagonal structure, see figure 6.2. In the next step this zero diagonal would be destroyed, but it is kept. The incomplete² Gauss-Jordan algorithm constructs a sequence of matrices in the following way: Let be (m_1, m_2) the start index of an occupied diagonal of the matrix A, u_i the i-th unit vector and

$$
\begin{aligned}
A^{(0)} &= A \ , \\
P^{(0)} &= I \ , \\
\tilde{C}^{(i)} &= I - \sum_{j=0}^{n-|m_1-m_2|-1} \frac{a^{(i-1)}_{m_1+j,m_2+j}}{a^{(i-1)}_{m_2+j,m_2+j}} u_{m_1+j} u^T_{m_2+j} \ , \\
a^{(i)}_{k,l} &= \begin{cases} \sum_{j=1}^{n} \tilde{c}^{(i)}_{k,j} a^{(i-1)}_{j,l} & if \quad (k,l) \in J^{(i)} \\ 0 & if \quad (k,l) \notin J^{(i)} \end{cases} \ , \qquad (6.22) \\
p^{(i)}_{k,l} &= \begin{cases} \sum_{j=1}^{n} \tilde{c}^{(i)}_{k,j} p^{(i-1)}_{j,l} & if \quad (k,l) \in J^{(i)} \\ 0 & if \quad (k,l) \notin J^{(i)} \end{cases} \qquad (6.23)
\end{aligned}
$$

$J^{(i)} = \{(k,l) | a^{(i)}_{k,l} \neq 0\}$ is the pattern of the matrix $A^{(i)}$. Choose as preconditioning matrix

$$P = A^{(n)-1} \cdot P^{(n)} \qquad (6.24)$$

The matrix P does not converge to A^{-1} for an increasing set $J^{(0)}$. The number of

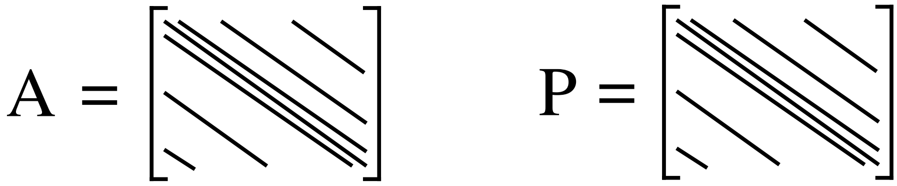


Figure 6.3: Pattern of A and P for incomplete Frobenius inverses and projection methods

operations for the determination of the matrix P is $\frac{\sharp J^2}{n}$ corresponding to $\frac{\sharp J}{n}$ matrix-vector multiplications.

In LINSOL
not yet implemented

6.8 Incomplete Frobenius Inverse

The idea of this incomplete inverse, see [48, 107, 62, 17], is to minimize the Frobenius norm

$$\begin{aligned} \|AP - I\|_{Frobenius} &= \|\sum_{i=1}^{n} APu_i - u_i\| \\ &= \sum_{i=1}^{n} \|APu_i - u_i\| \\ &= \sum_{i=1}^{n} \|Ap_i - u_i\| \end{aligned}$$

where u_i is the i-th unit vector and p_i is the i-th column vector of P. $P = (p_{k,l})$ has a prescribed pattern: $p_{k,l} = \ 0 \quad if \quad (k,l) \notin J$, see figure 6.3 if J is the pattern of A. See [18, 62] for very elaborate adaptive choices of J. The items $\|Ap_i - u_i\|$ can be minimized independently by solving n small linear systems:

$$\sum_{k} u_l^T A^T A u_k p_{k,i} = u_l^T A^T u_i \tag{6.25}$$

for $(k,i) \in J$ and $(l,i) \in J$ and $i = 1, \ldots, n$. If in each column of P there are m entries, then the columns of P can be calculated independently by solving n symmetric and positive definite linear systems of dimension m. This solution procedure can be optimally vectorized and parallelized. The number of operations for the determination of the matrix P is $\frac{1}{6}(\frac{\sharp J^3}{n^2}+3\frac{\sharp J^2}{n}+3\sharp J)$ corresponding to $\frac{1}{6}(\frac{\sharp J^2}{n^2}+3\frac{\sharp J}{n}+3)$ matrix-vector multiplications.

In LINSOL
not yet implemented

6.9 Projection Methods

Projection methods are a generalization of incomplete Frobenius preconditioning. Equation (6.25) is equivalent to

$$u_l^T A^T [AP - I] u_i = 0 \tag{6.26}$$

for all l with $(l,i) \in J$ and $i = 1, \ldots, n$. If the matrix in the brackets disappears we have the optimal preconditioning. Therefore, equation (6.26) can be considered as weak formulation for $AP - I = 0$. In (6.26) this matrix is projected from left and from right. A natural generalization is to consider the equation

$$u_l^T Z_L [AP - I] Z_R u_i = 0 \ , \tag{6.27}$$

where Z_L and Z_R are non-singular matrices. Equation (6.27) is again a weak formulation for $AP - I = 0$, i. e. $P = A^{-1}$. Equation (6.27) includes the Frobenius preconditioning as special case with $Z_L = A^T$ and $Z_R = I$. If $Z_R = I$ the columns of P can be calculated independently by solving n small linear systems for the columns of P:

$$\sum_k u_l^T Z_L A u_k p_{k,i} = u_l^T Z_L u_i \tag{6.28}$$

for $(k,i) \in J$ and $(l,i) \in J$ and $i = 1, \ldots, n$. An obvious choice of Z_L and Z_R is $Z_L = Z_R = I$ resulting in the construction prescription for P:

$$u_l^T [AP - I] u_i = 0 \ . \tag{6.29}$$

In figure 6.3 the pattern of P is depicted if it is chosen to be the pattern of A.

Note that the systems resulting from (6.29) are in general non-symmetric and non-positive definite, but have a certain sparsity structure if A is sparse, in contrast to the symmetric, positive definite and full systems resulting from the calculation of the incomplete Frobenius inverse. Thus, an iterative solution of the small systems becomes attractive [140]. Theoretical investigations have been performed by Kolotilina and Yeremin [67].

In LINSOL
not yet implemented

6.10 Iterative Methods as Preconditioners

Any iterative method can be used as preconditioner. It is also possible to precondition the preconditioner and so on, so that we arrive at a cascade of iterative methods. For practical experience with this type of preconditioning using CG methods see [89, 100, 121]. There are investigations for preconditioners based on SOR by Kaporin, Kolotilina and Yeremin [66].

In LINSOL
not yet implemented

6.11 Multi-level Methods

Multilevel methods have become a powerful tool for the solution of partial differential equations. We do not give details, but point out the main ideas and describe a distinct method. For details we refer to [1, 5, 10, 15, 23, 54, 55, 56, 80].

The idea of multi-level methods is to consider a sequence of different discretizations of a problem by the finite difference or by the finite element method. A sequence of coarse to fine grids is constructed. The solution is required for the finest grid. An approximation of a fine grid solution is distributed onto a coarse grid, called restriction, or the problem is solved approximately on a coarse grid and the approximation is interpolated onto a finer grid, called prolongation. The restrictions and prolongations of the approximations can be used as starting guess, as defect correction or as preconditioning for the next system. The idea is that a solution on a coarse grid is cheap and may transfer much information onto the finer grid.

Algebraic multigrid is based on an analysis of the graph of the matrix and does not consider any grid. The reduction process is based on the relations between the entries of the matrix [80, 79, 44].

Classical multigrid methods are combined with matrix splitting methods as for example SOR. It is known that SOR dampes very well the short waves of the error compared with the grid size, while the long waves need a lot of iterations. A long wave becomes a short wave on a coarser grid and is damped on this grid much faster. Importantly, the number of operations increases only linearly with the number of unknowns.

Multigrid methods are normally tuned to the special problem. The data reduction (DARE) method for linear systems arising from the finite difference method is a special multi-level method. The technique is suited for arbitrary operators of nonlinear systems of elliptic and parabolic partial differential equations with arbitrary nonlinear boundary conditions. The consistency order of the finite difference method can be arbitrary.

Details are given in [9, 101, 104, 131, 132].

In LINSOL
not yet implemented

6.12 Discussion

Normalization is cheap, easy to implement, optimally vectorizable and parallelizable. It may result in a quite better convergence. Therefore, it should always be applied. Due to its simple nature it cannot strongly accelerate slowly converging problems.

Polynomial preconditioners are applicable for general matrices and reduce the number of iteration steps. They require j matrix-vector multiplications per iteration. For minimization methods based on Krylov spaces the convergence is slowed down if we measure the operations. But the number of dot products can be reduced by polynomial preconditioners. Therefore, this technique may reduce the CPU-time on massively parallel systems.

ILU and IC decompositions require the solution of two triangular systems during each iteration step. Therefore, they cannot be executed efficiently on vector and parallel computers unless distinct matrix structures allow very elaborate coding.

The amount of operations is a problem for QR decompositions. To determine strategies to reduce the number of operations further research seems to be necessary.

The incomplete Gauss-Jordan variants are not robust. They seem to work satisfactorily only for matrices with scattered diagonals.

The incomplete Frobenius inverse and projection methods are not robust if a fixed pattern is prescribed. Therefore, adaptive techniques for the determination of the pattern of the preconditioner are required. For this purpose further research is necessary. For Frobenius preconditioning an elaborate strategy to determine the pattern adaptively has been proposed recently by Huckle and Grote [62].

Preconditioning by iterative methods requires an elaborate control strategy. We know of no robust implementation of this technique.

Multilevel methods are primarily suited for linear systems arising from partial differential equations. There are attempts to generalize these methods in order to obtain algorithms for linear systems arising from arbitrary problems. But these efforts are up to now more related to basic research than to production programs. However, multi-level methods require fine-tuned parameters. DARE does not depend on parameters, but is restricted to structured grids.

We would like to remark that all discussed preconditioning techniques except normalization depend on parameters. For polynomial preconditioners the degree of the polynomial has to be chosen. For ILU, IC, IQR, incomplete Gauss-Jordan, Frobenius

preconditioning by	vectorizable/ parallelizable during start-up	vectorizable/ parallelizable during iteration	approximation for
Normalization	yes	yes	A^{-1}
Polynomials	yes	yes	A^{-1}
Normal equations	yes	yes	A^{-1}
ILU	not efficient	not efficient	A
IC	not efficient	not efficient	A
IQR			
$A = QR + E$	yes	not efficient	A
$A\tilde{R} = Q + E$	yes	yes	A^{-1}
Incomplete Gauss-Jordan	not efficient	yes	A^{-1}
Incomplete^2 Gauss-Jordan	yes	yes	A^{-1}
Incomplete Frobenius inverse	yes	yes	A^{-1}
Projection methods	yes	yes	A^{-1}
Iterative methods	yes	yes	A^{-1}
Multilevel methods	yes	yes	A^{-1}

Table 6.1: Different preconditioners

inverses and projection methods the fill-in set J has to be determined. Iterative preconditioners depend on the number of iterations. The same holds for DARE and other multi-level methods for the iterative methods on different levels. Moreover for multi-level methods the strategies of restrictions and prolongations, the weights and other parameters have to be tuned to the special problem.

In table 6.1 the different preconditioners are compared with respect to vectorization and parallelization and for determining if they approximate the inverse of A or perform a decomposition.

Chapter 7

Parallelization

In order to solve large scientific and industrial problems fast algorithms and advanced computer architectures are necessary. But the fastest mathematical algorithm may waste so much CPU-time of the fastest and most advanced available computer that it performs worse than on an ordinary "slow" machine – if it is not implemented properly for the special computer architecture. We will address in this chapter the efficient implementation of Krylov subspace methods on vector and parallel computers. We do not consider the implementation of preconditioners because most preconditioning techniques cannot be implemented on vector and parallel computers efficiently at all for arbitrary matrix structures. Moreover, each particular preconditioning technique, each particular matrix structure and each different strategy for a distinct preconditioner requires another implementation. Thus such an analysis is beyond the scope of this report.

Most of the here described methods use as basic operations

- matrix-vector multiplications (mvm),
- reduce operations like dot products,
- triadic vector operations of the form $vector = vector + scalar * vector$.

For an efficient implementation of the methods on supercomputers it is sufficient to care about these three operations.

The efficient implementation of these operations is standard for vector computers [93, 128] and the dot product is strongly supported by compilers. Therefore, we will focus on massively parallel systems. For parallel computers there are rules that should be followed to get efficient codes [58] based on the hardware design [57, 94, 95, 96]. For SIMD (single instruction multiple data) architectures an investigation is given by Hanke, Hochbruck and Niethammer [59]. For MIMD (multiple instruction multiple data) architectures there are a lot of investigations. The communication aspects of the CG method are investigated by Crone and van der Vorst [20]. A more efficient implementation of GMRES by grouping the Arnoldi vectors is proposed by de Sturler [21] for massively parallel computers. Hereafter, we will describe the concept proposed in [97, 130]. It is based on the rules in [58] for the message passing programming style. This technique was chosen because up to now it is available for all massively parallel computers in contrast to other techniques and because the distribution of data and operations can be controlled quite efficiently. We will discuss the principles for programming and then show how to implement the two essential operations, the matrix-vector multiplication and the reduce operations. The third essential operation, the triadic operation, is no problem for vector and parallel computers. On parallel computers it can be executed independently for the corresponding parts of the vectors that are distributed on different processors.

7.1 Latency Hiding

The main principle for communication between two independent processes is to hide the communication behind the calculation. This technique is called latency hiding. It only can be achieved if asynchronous communication is supported by the computer manufacturer. The amount of communication that can be hidden depends first on the volume-to-surface ratio of the algorithm and second on the balancing factor of the computer system. The volume-to-surface ratio is the ratio between the computation units and the communication units, which can be overlapped, and depends on the chosen algorithm. The balancing factor is the ratio between the speed of the communication network and the arithmetic performance on a single processor and depends on the manufacturer. Thus a software engineer can enhance the amount of communication that can be hidden by the choice of an algorithm with a better volume-to-surface ratio - if possible - and by the optimal exploitation of the inherent asynchronity in the chosen algorithm (see section 7.4).

7.2 Scalability

For parallel computers it is a desirable property that a program should consume the same CPU-time if the computational amount is increased in the same way as the number of processors. This property is called scalability regarding to the computation time and it guarantees that arbitrarily large problems could be processed in reasonable time if enough processors were available.

Definition 7.1 *An algorithm is called scalable regarding the computation time if the computation time for n operations executed on one processor is roughly the same as for $p \cdot n$ operations executed on p processors.*

It should be mentioned that most of the iterative methods considered as complete algorithm are not scalable regarding to the computation time because usually the number of iterations needed to achieve a certain accuracy increases with the dimension of the system. But the goal is to achieve scalability of the basic operations of iterative methods; and the basic operations of the considered iterative methods are the matrix-vector multiplication, the reduce operations and the triadic operations.

Remark: For the solution of sparse linear systems the computational amount for the basic operations like matrix-vector multiplications is proportional to the number of unknowns and thus the term operations can be replaced here by the term unknowns.

Note that in this definition the number of unknowns (operations) increases with the number of processors following an approach of Gustafson [49]. We think that this definition is meaningful from a scientific and a practical viewpoint. There is no reason to increase the number of processors if the problem can be solved with a less number of processors. The reason to increase the number of processors is that the problem size becomes so large that the execution time exceeds some bound.

There is also a different definition for scalability regarding to the computation time where the problem size is constant, but the number of processors increases. This definition becomes meaningful if a series of problems should be solved in less time, for example to get a reasonable turn-around. However, in this case the ratio between communication with start-up times and computation increases and thus from a particular number of processors no further speed-up is obtained. Moreover, a limit for speed-up is given by Amdahl's law [2].

There is an additional desirable property for parallel computers that a program should consume the same amount of memory per processor if the total memory requirement is increased in the same way as the number of processors. This property is called scalability regarding to the memory and it guarantees that problems which are memory bound could be processed by enlarging the number of processors.

Definition 7.2 *An algorithm is called scalable regarding the memory if the memory requirement for n unknowns on one processor is roughly the same as the memory requirement for $p \cdot n$ unknowns on each of the p processors.*

The program package LINSOL achieves scalability regarding the memory by a blockwise distribution of all vectors and a row-blockwise distribution of the matrix onto the processors.

7.3 Portability

A further important property of LINSOL is the portability that is achieved by a portable message passing interface named P_MPI. P_MPI is a user interface to all common message passing interfaces like MPI, PVM, PARMACS, etc. Each of those can be used by linking a special library which contains the proprietary message passing calls beneath the P_MPI user interface. Thus the different proprietary message passing interfaces are used by seting the matching library name in the link step. For more exact informations we refer to the P_MPI User's Guide [in preparation].

7.4 Matrix-vector Multiplication

For all here discussed methods matrix-vector multiplications

$$c = Ay \tag{7.1}$$

are needed. For BCG, BICO, QMR, CGNE and CGNR-ATPRES the multiplication by the transpose of A

$$d = A^T z \tag{7.2}$$

is additionally required. The overall goal for the implementation of this operation is to achieve scalability, to leave out unnecessary communication cycles and to hide the communication behind the calculation for computers with asynchronous communication (latency hiding).

7.4.1 Matrix Distribution

Figure 7.1 gives an example of a sparse matrix of 64 unknowns distributed to 4 processors. The matrix A is split rowwise into matrix stripes A^i of length n_i (=$lmatbk(i)$), $i = 1, \ldots, p$ (note that generally $n_j \neq n_i$ for $j \neq i$); $ptrmbk(i)$ points to the last row of the matrix stripe A^{i-1}. A^i is a $n_i \times n$ matrix containing the contiguous rows $ptrmbk(i-1)+1, \ldots, ptrmbk(i-1)+lmatbk(i)$ with $ptrmbk(i-1)+lmatbk(i) = ptrmbk(i)$ and $i = 2, \ldots, p$. The matrix stripe A^i is stored in the memory of processor i.

Additionally to this physical subdivision a logical subdivision of each stripe $A^i, i = 1, \ldots, p$ into p column blocks $A^{i,1}, \ldots, A^{i,p}$ following the subdivision scheme for the rows is done. Thus the $n_i \times n_j$ block matrix $A^{i,j}$ is stored in the memory of processor i and contains the columns $ptrmbk(j-1)+1, \ldots, ptrmbk(j-1)+lmatbk(j)$ with $ptrmbk(j-1)+lmatbk(j) = ptrmbk(j)$ and $j = 2, \ldots, p$. Figure 7.2 shows this situation for the example in Figure 7.1.

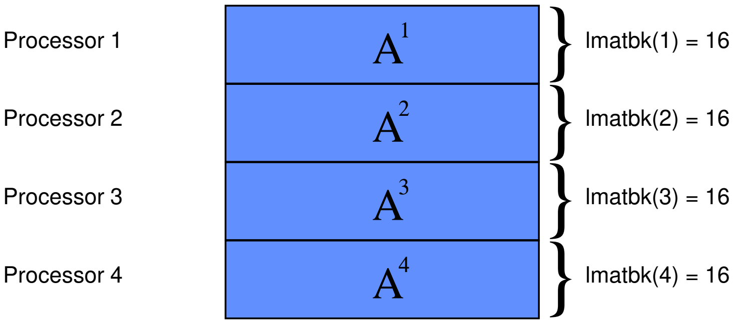


Figure 7.1: The physical distribution of a 64×64 matrix to 4 processors

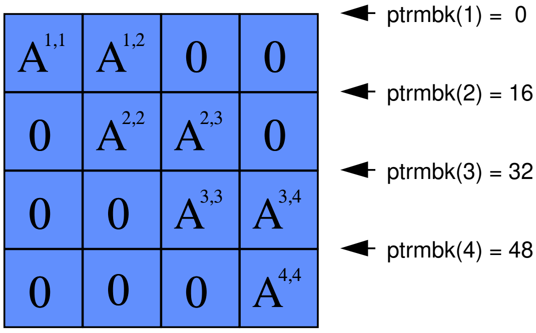


Figure 7.2: The physical and logical distribution of a matrix to 4 processors

If the matrix A is sparse, the block matrices $A^{i,j}$ will be sparse matrices, too. Some will contain only zero entries, called a zero block. Apart from that holds that the (block) matrix-vector multiplication is independent of the special storage format of the matrix. It can be applied to any storage format, e. g. for diagonally stored finite difference matrices, for rowwise stored matrices with column index, for columnwise stored matrices with rowindex, etc. It is sufficient to define for each storage pattern the corresponding block matrix-vector multiplication.

The input vector y and the output vector c are distributed like the rows of the matrix, i.e. processor i has the subvectors $y^i = (y_{ptrmbk(i-1)+1}, \dots, y_{ptrmbk(i)})$ and $c^i = (c_{ptrmbk(i-1)+1}, \dots, c_{ptrmbk(i)})$ of length n_i. The problem is that processor i needs for the processing of the block matrices $A^{i,j}, j \neq i$ the entries of subvector y^j, which are not available on processor i without communication. If the multiplication by the transpose of A is necessary, then the rowwise distribution of A corresponds to a columnwise distribution of A^T and the vectors d and z respectively get the role of the vectors y and c respectively in the multiplication procedure.

7.4.2 The Multiplication by A

The matrix-vector multiplication $c = Ay$ is computed on each processor $i, i = 1, \ldots, p$ with a logical subdivision of the matrix A^i and a physical subdivision of the vector y in p blocks:

$$c^i = c^i + \sum_{j=1}^{p} A^{i,j} y^j \tag{7.3}$$

In the Figure 7.3 the matrix-vector multiplication is depicted on 4 processors. Diagonal matrix blocks (e.g. $A^{1,2}, A^{2,3}, A^{3,4}, A^{4,1}$) are computed concurrently, if at least one of these blocks contains non-zero matrix elements. If all diagonal matrix blocks contain no non-zero matrix elements, the computation of all these diagonal matrix blocks can be skipped, i.e. the number of communication cycles - stored in variable $njump$ - is reduced by one. Thus $njump = p$ holds, if all diagonal matrix blocks contain any non-zero matrix elements; $njump = 0$ would hold, if the whole matrix would contain only zero elements (what a nonsense!). The integer vector $jump = (jump_i)_{i=1,njump}$ indicates the distance to the processor, to which a vector part has to be sent. It holds $\sum_{i=1}^{njump} jump_i = p$, so that at the end of one complete matrix-vector multiplication the data originally stored in the memory of processor j are put again in the memory of processor j, $j = 1, \ldots, p$. In the case of a matrix with no zero diagonal matrix blocks $njump = 4$ holds on 4 processors and the vector $jump$ has the entries $(1, 1, 1, 1)$. In the example of Figure 7.3 one diagonal matrix block can be skipped: we get $njump = 3$ and $jump = (2, 1, 1)$.

Using this optimization strategy for the communication the matrix-vector multiplication runs more efficient, if the number of communication steps $njump$ is as small as possible. In finite element applications this can be realized by a bandwidth optimized numbering of the global nodes.

We use two buffers $bufs$ and $bufr$: $bufs$ for the part of the input vector, which is used to perform the block multiplication $A^{i,j}y^j$ for all steps j, and $bufr$, in which the input vector for the next step is received. The procedure now runs as described in Algorithm 17.

Before starting the matrix-vector multiplication processor i copies his input vector y^i into the send buffer $bufs$. Then it triggers the sending of this buffer to the processor, which needs this vector part in the next computation cycle, and triggers the receiving of the vector part of y, which is needed by itself in the next computation cycle. While the communication runs, the multiplication of the current input vector stored in buffer

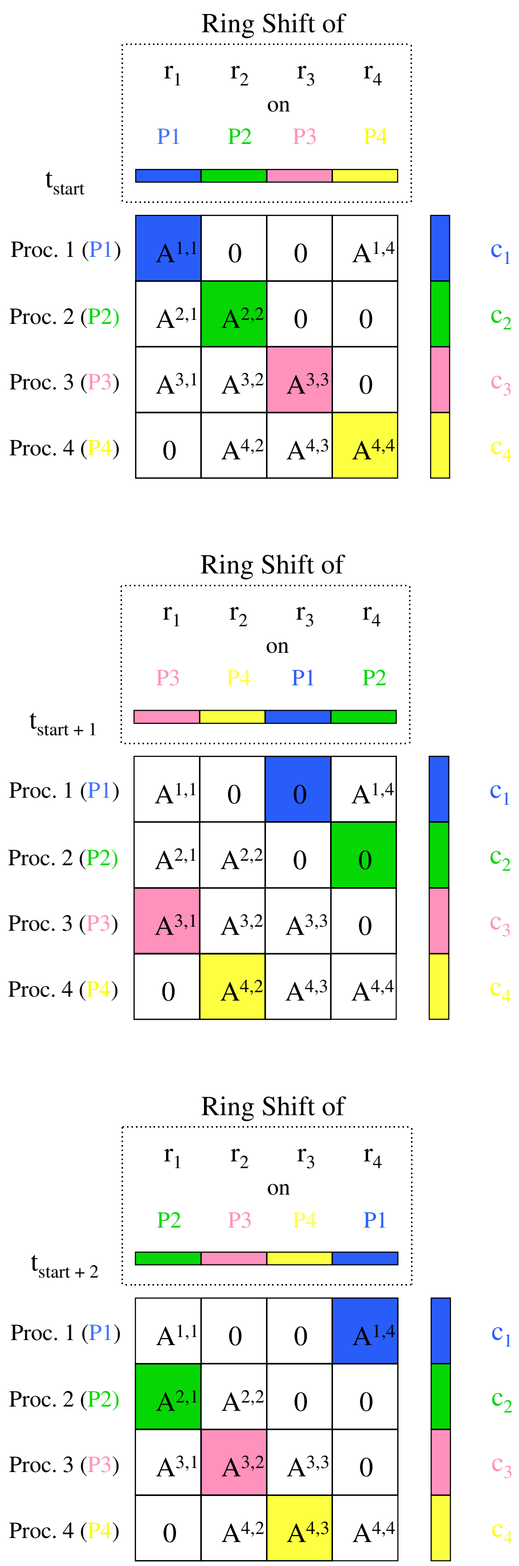


Figure 7.3: The three computation steps of the matrix-vector multiplication on 4 processors

```
i=j
s=pointer(bufs)
r=pointer(bufr)
bufs=y^i

do k=1,njump
        if (p>1) then rcva bufr from processor { i - jump(k), if i - jump(k) > 0
                                                { p + i - jump(k), if i - jump(k) < 1
        if (p>1) then snda bufs to processor { i + jump(k), if i + jump(k) < p + 1
                                              { i + jump(k) - p, if i + jump(k) > p
        c^i = c^i + A^{i,j} * bufs

        j=j-jump(k)
        if (j<1) then j=j+p
        if (p>1) then sndwait
        if (p>1) then rcvwait
        swap s,r
enddo
```

Algorithm 17: Parallel matrix-vector multiplication by A

$bufs$ with the submatrix $A^{i,j}$ is executed. After the synchronization the pointers to the send and the receive buffers are exchanged and the next cycle starts.

7.4.3 The Multiplication by A^T

Now we face the problem of the matrix-vector multiplication with the transposed matrix A^T: $d = A^T z$. Since the matrix A is not transposed physically, now the parts of the result vector d have to be sent around. The parts of the input vector z now don't have to be moved around. Thus on each processor $i, i = 1, \ldots, p$ different vector parts of d are computed by

$$d^j = d^j + \sum_{j=1}^{p} A^{i,j^T} z^i \tag{7.4}$$

Again we need two buffers $bufs$ and $bufr$ to manage the locally non-blocking send and receive of the result vector parts. But since we may not change these buffers, an additional result buffer buf for the matrix-vector multiplication has to be introduced, which is added to the received part after the synchronization. The procedure runs as described in Algorithm 18.

On processor i the matrix-vector multiplication does not start with the main diagonal matrix block but the neighboured matrix block, which is a non-zero matrix block; this is caused by the facts that now the result vector has to be moved around and again

```
i=j
s=pointer(bufs)
r=pointer(bufr)
if (p>1) then bufs=d^i else bufr=d^i

do k=1,njump
        if (p>1) then rcva bufr from processor { i - jump(k), if i - jump(k) > 0
                                               { p + i - jump(k), if i - jump(k) < 1
        if (p>1) then snda bufs to processor { i + jump(k), if i + jump(k) < p + 1
                                             { i + jump(k) - p, if i + jump(k) > p
        buf=0
        j=j-jump(k)
        if (j<1) then j=j+p

        buf = buf + A^{j,i} * z^i

        if (p>1) then rcvwait
        bufr = bufr + buf

        if (p>1) then sndwait
        swap s,r
enddo

d^i =bufs
```

Algorithm 18: Parallel matrix-vector multiplication by A^T

alternating buffer technique is used. The computed block matrix-vector product has always to be added to the received result vector part, because only then simultaneous computation and communication can take place.

Processor i triggers the sending of its result vector part and triggers in the k-th communication cycle the receiving of the result vector part $j = i - \sum_{l=1}^{k} jump(l)$ ($j > 0$ must hold: if $j < 0$ then $j = j + p$) into the buffer $bufr$. Then the multiplication of the input vector part z^i by the block matrix A^{i,j^T} is performed and the result vector is stored into the computation buffer buf, which is added to the received result vector part j after the synchronization of the receive. The addresses of the send and receive buffer are exchanged and the next cycle starts. At the end the sending buffer contains the result vector d^i, which has passed all processors (with non-zero matrix blocks) by a ring shift.

7.4.4 The Symmetric Matrix A

A symmetric matrix A can be represented by a diagonal matrix D and a matrix B with zero entries in the main diagonal: $A = B + D + B^T$. B and B^T have a disjoint shape. To save storage it is efficient to store only the D and B. The matrix-vector multiplication is executed by a multiplication with B and B^T and D. To ensure a good load balancing the matrix blocks $B^{i,j}$ and $B^{j,i}$ should contain nearly the same number of nonzero entries.

7.4.5 The Multiplication by a Symmetric Matrix A

Naturally the multiplication with B and B^T could be executed successively. More efficient is the simultaneous processing of the entries and their transposed entries in the main diagonal block matrices $B^{i,i}$ on each processor i; then load operations of block matrix entries can be saved. For all other blocks the multiplication is split up into a multiplication by A and a multiplication by A^T. Nevertheless an optimization takes place, which is due to the communication. The buffers $bufs$ and $bufr$ contain two vector parts - y^i and d^i. This saves one communication startup per communication cycle. The procedure runs as described in Algorithm 19.

The simultaneous multiplication with a matrix and its transposed is also used in some LINSOL methods for non-symmetric matrices.

7.5 Reduce Operations

While reduce operations like dot products are processed very efficiently on vector computers, they become a critical operation on parallel computers. The reason is that the result of a dot product or of a min/max-function is a simple scalar and has to be available on all processors. Therefore, a large communication overhead is necessary while only a few data are sent.

In order to get an efficient performance the reduce operations are implemented by a cascadic operation and additionally independent reduce operations are calculated by one piece and all independent partial and final results then sent by one and only one message each.

7.5.1 Dot Products and Min/Max-Functions

As well known now, in LINSOL all vectors are distributed onto the p processors in pieces of length $\frac{l}{p}$. Let us assume for the sake of simplicity that l is a multiple of p. In order to avoid unnecessary communication startups as much vectors as possible are used for the simultaneous computation of reduce operations. In the first step the partial results of

i=j
s=pointer(bufs)
r=pointer(bufr)
bufs=$(y^i, 0)$

do k=1,njump
    if (p>1) then rcva bufr from processor $\begin{cases} i - jump(k), if\ i - jump(k) > 0 \\ p + i - jump(k), if\ i - jump(k) < 1 \end{cases}$
    if (p>1) then snda bufs to processor $\begin{cases} i + jump(k), if\ i + jump(k) < p+1 \\ i + jump(k) - p, if\ i + jump(k) > p \end{cases}$
    if (i=1) then
        $c^i = c^i + A^{i,i} * y^i$
    else
        $c^i = c^i + A^{i,j}$ * bufs(first vector)
    fi

    if (p>1) then
        if (i<njump) then
            j=j-jump(k)
            if (j<1) then j=j+p

            buf=0
            buf = buf + $A^{j,i} * z^i$
        fi

        rcvwait
        if (i<njump) then
            bufr(second vector) = bufr(second vector) + buf
        sndwait
        swap s,r
    fi
enddo

if (p>1) then $c^i = c^i$ + bufs(second vector)

Algorithm 19: Parallel matrix-vector multiplication by a symmetric matrix

the vectors x_k and y_k with $k = 1, \ldots,$number of simultaneously computed dot products

$$\alpha_k^j = \sum_{i=1}^{\frac{l}{p}} x_k(i) y_k(i) \tag{7.5}$$

for $j = 1, \ldots, p$ are calculated on each processor; j is no index of the vectors x and y, because the p different vector parts are local vectors on the p processors. Minimal and maximal partial scalars of independent vectors x_k are computed e.g. for the minimum-function by

$$\alpha_k^j = \min_{i=1,\ldots,\frac{l}{p}} x_k(i). \tag{7.6}$$

Then the vector of scalars α^j is handed over to the subroutine REDUCE.

7.5.2 The Subroutine REDUCE

This subroutine first gathers a vector of independent scalars by a cascade, where processors, which just have received a message, perform a basic function like a dot product, minimum or maximum with the local and received scalars as parameters. Second the final scalars are scattered to all processors by a cascade running in the reverse order.

Thus α_k^{2jj} is sent to processor $2jj - 1$ for $jj = 1, \ldots, \frac{p}{2}$. Let us further assume that $p = 2^m$ for some integer m. Then the local and received scalars are added on each second processor:

$$\alpha_k^{2jj-1} = \alpha_k^{2jj-1} + \alpha_k^{2jj} \tag{7.7}$$

for $jj = 1, \ldots, \frac{p}{2}$. This procedure continues until the final results α_k are obtained on processor 1, see figure 7.4 for 7 processors. The cascade is processed in the reverse order to distribute the complete results to all processors. The number of communication steps is

$$2\ ([log_2(p-1)] + 1), \tag{7.8}$$

where [z] is the next integer less or equal z. It is obvious that the reduce operations are not scalable because the communication time increases logarithmically with the number of processors.

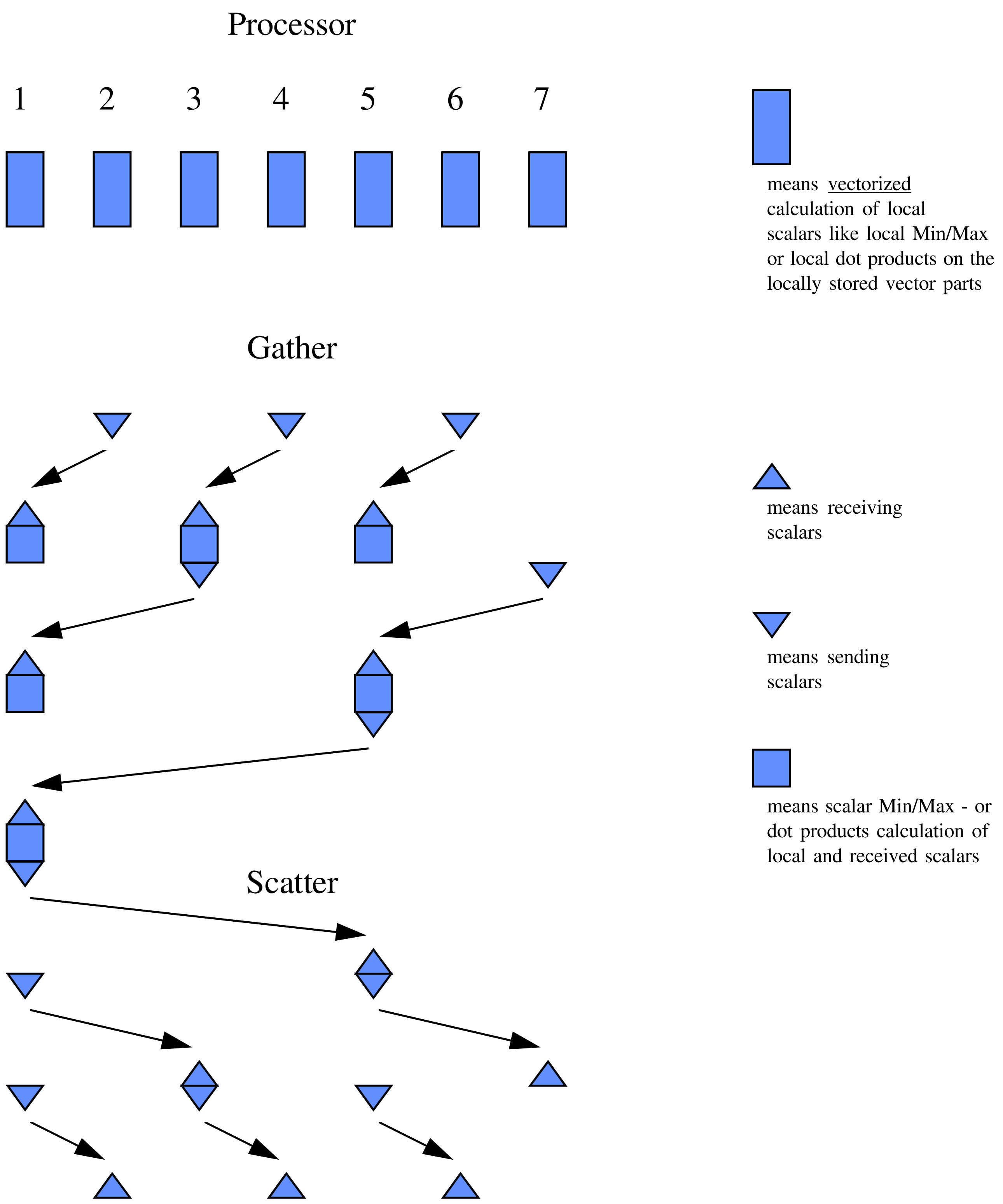


Figure 7.4: Computation of dot products on 7 processors

Chapter 8

Data Structures

If the global matrix is sparse, the block matrices $A^{i,j}$ are sparse matrices, too. There is no difference between the global matrix and the block matrices regarding the data structures; looking at the whole matrix or at one block matrix is the nearly the same. So we must look for suitable storage schemes for sparse matrices, which lead to optimal data structures for the processing on superscalar and vector processors. We pay attention to the three most important rules for efficient programming on superscalar and vector processors [93]:

- long inner loops,
- contiguous storage access,
- no recursion.

The processors of the most existing parallel computers have superscalar processors, which can be programmed optimally with the same strategies as the vector computers. The problem of cache reuse concerning only superscalar processors is discussed at the end of this chapter.

The $n_i \times n_j$ block matrix $A^{i,j}$ and the global matrix A respectively is logically decomposed in $nstopa$ (= number of storage patterns) $n_i \times n_j$ submatrices $A_s^{i,j}$ and A_s respectively.

$$A^{i,j} = \sum_{s=1}^{nstopa} A_s^{i,j}, i,j = 1,\ldots,p. \tag{8.1}$$

The submatrices $A_s^{i,j}$ have disjoint shapes of their nonzero elements. They have one of the special shapes described below, i.e. they contain one and only one special storage pattern; but they can contain many "vector terms" (stands for vectors within the storage patterns like diagonals, full rows, etc.) of one special storage pattern. The matrix-vector multiplication with the block matrix $A^{i,j}$ and its transposed $A^{j,i}$ is now performed by sequential matrix-vector multiplications with the matrices $A_s^{i,j}$ and the operations are very simple:

$$A^{i,j}y^j = \sum_{s=1}^{nstopa} A_s^{i,j}y^j, {A^{i,j}}^T z^i = \sum_{s=1}^{nstopa} A_s^{j,i} z^i. \tag{8.2}$$

In the following we will present the basic operation

$$c = c + A_s^{i,j} y \tag{8.3}$$

for the different storage patterns, induced by the submatrix $A_s^{i,j}$. It is important to stress that in contrast to Sparse BLAS [24] the storage patterns *have not* to be the same for all block matrices $A^{i,j}$.

LINSOL supports the following storage patterns:

1. main diagonal (MAIN),
2. diagonals (FULLDIAG),
3. packed diagonals (PDIAG),
4. indexed column (IXCOL),
5. indexed row (IXROW),
6. starry sky (SKY),
7. full row (FULLROW),
8. full column (FULLCOL),
9. packed row (PROW),
10. packed column (PCOL),

In LINSOL the number of supported storage patterns is 10 ($nstopa = 10$) and each block matrix $A^{i,j}$ can be represented as sum of 10 submatrices $A_s^{i,j}$:

$$\begin{aligned} A^{i,j} = \sum_{s=1}^{10} A_s^{i,j} \quad = \quad & 1 * MAIN + nvt_2^{i,j} * FULLDIAG + nvt_3^{i,j} * PDIAG \\ & + nvt_4^{i,j} * IXCOL + nvt_5^{i,j} * IXROW + nvt_6^{i,j} * SKY \\ & + nvt_7^{i,j} * FULLROW + nvt_8^{i,j} * FULLCOL + nvt_9^{i,j} * PROW \\ & + nvt_{10}^{i,j} * PCOL. \end{aligned}$$

The number of vector terms within each $A_s^{i,j}$ is stored in $nvt_s^{i,j}$. In LSOLPP this variable is stored in the two-dimensional array $ptrinf(s,j)$. Index i can be dropped, because it stands for the processor, on which the **local** array $ptrinf(s,j)$ is used. The variable iac and iar respectively defines the starting index regarding the column and row respectively

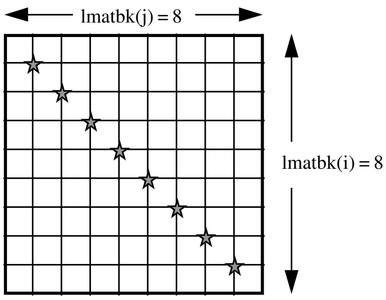


Figure 8.1: The storage pattern *Main Diagonal*

of a vector term. The variables iac, iar and the array $index$ are **global** and relate to the **global** matrix A. The variables l and lvt are **local** and relate to the **local** matrix $A^{i,j}$. The variables $iacl$ and $iarl$ and the array $indexl$ correspond to the variables iac and iar and the array $index$, but they are **local**.

Now the different storage patterns are described.

8.1 Main Diagonal

A_1 is the main diagonal of the matrix A. $A_1^{i,i}$ contains one diagonal of full length and thus has elements, which can be zero, too. The main diagonal contains $lmatbk(i) = l$ (= length of the matrix block) entries $a_{ii,jj}$ with $ii = jj, iar = iac, iarl = iacl$, where $a_{iar+1,iac+1}$ is the first nonzero entry and $a_{iar+l,iac+l}$ is the last nonzero entry (see Figure 8.1). Gaps with no entries have to be filled by zeroes. Operation 8.3 runs then as follows:

$$\begin{aligned} &\text{for } k = 1 \text{ to } l \text{ do} \\ &\qquad c_k = c_k + a_{iar+k,iac+k} * y_k \\ &\text{enddo} \end{aligned} \tag{8.4}$$

This operation is a special FULLDIAG-operation, which is explained now.

8.2 Diagonals

$A_2^{i,j}$ contains only diagonals with elements, which can be zero, too. One diagonal contains lvt (= length of vector term) entries $a_{ii,jj}$ with $ii - jj = iar - iac = iarl - iacl = const$, where $a_{iar+1,iac+1}$ is the first nonzero entry and $a_{iar+lvt,iac+lvt}$ is the last nonzero entry (see Figure 8.2). Gaps with no entries have to be filled by zeroes. Operation 8.3 runs

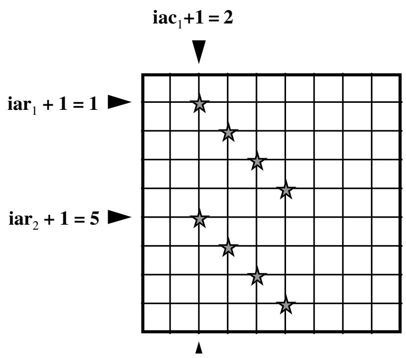


Figure 8.2: The storage pattern *Full Diagonal*

then as follows:

$$
\begin{aligned}
&\text{for } k = 1 \text{ to } lvt \text{ do} \\
&\qquad c_{iarl+k} = c_{iarl+k} + a_{iar+k,iac+k} * y_{iacl+k} \\
&\text{enddo}
\end{aligned}
\tag{8.5}
$$

This is a contracting vector triad operation [93]. Three load and one store operation are necessary for each processed vector term. The access to the submatrix elements of A_2 is not visible in the above described loop; we will explain it at the end of this chapter.

If there are too many gaps with zero elements in a diagonal, it is more efficient to pack the diagonal; now the submatrix $A_3^{i,j}$ contains packed diagonals with only nonzero elements; the integer vector *index* marks the position of the nonzero elements in the diagonal (see Figure 8.3). It will never contain twice the same indices and therefore the modification of loop 8.5 is free of recursion:

$$
\begin{aligned}
&\text{for } k = 1 \text{ to } lvt \text{ do} \\
&\qquad c_{iarl+index_k} = c_{iarl+index_k} + a_{iar+index_k,iac+index_k} * y_{iacl+index_k} \\
&\text{enddo}
\end{aligned}
\tag{8.6}
$$

Here we need one additional load operation for the index vector and the load and store operations for the input and output vector are indexed, which includes the danger of bank conflicts. The access to the index vector elements is not visible in the above described loop; we will explain it at the end of this chapter. These braking effects are compensated by the lower loop length *lvt*. According to our experiences a switch from full to packed storaging is favorable with regard to the performance for less than 35% nonzero elements in the diagonal on a vector processor and about 75% on a superscalar processor. The optimal value depends on the arithmetic performance of the processor and the number of load/store paths connected to the processor.

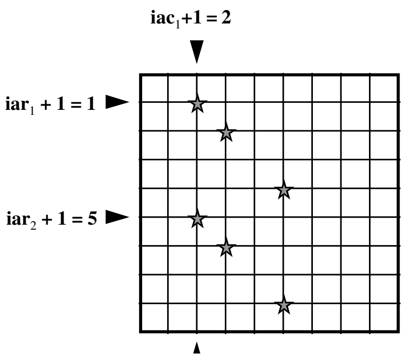


Figure 8.3: The storage pattern *Packed Diagonal*

For the multiplication with A^T only the offsets $iacl$ and $iarl$ of input vector y and output vector c have to be exchanged in the loop 8.5 and the loop 8.6. The full and packed diagonal are the optimal storage patterns for a vector computer, if the sparse matrix consists of a band of diagonals.

8.3 Indexed Column and Indexed Row

If the length of one vector term stored in the diagonal storaging scheme is too short, it is much better to gather many short diagonals into a new storage pattern. $A_4^{i,j}$ can contain only vector terms with entries stored in contiguous rows. For all rows $iar < ii \leq iar + lvt$ a element $a_{ii,index_k}$ in the block matrix is selected by the column index $index_k$ (see Figure 8.4). We call this pattern *Indexed Column.* The operation 8.3 is then

$$\begin{aligned} &\text{for } k = 1 \text{ to } lvt \text{ do} \\ &\qquad c_{iarl+k} = c_{iarl+k} + a_{iar+k,index_k} * y_{indexl_k} \\ &\text{enddo} \end{aligned} \tag{8.7}$$

As for the packed diagonal, this operation requires four loads and one store operation, but it seems to be faster, since the loading and storing of the result vector works without an index.

Before we discuss the efficiency, we want to look to the related storage pattern using a row index: $A_5^{i,j}$ can contain only vector terms with entries stored in contiguous columns. For all columns $iac < ii \leq iac + lvt$ a element $a_{index_k,ii}$ in the block matrix is selected by the row index $index_k$. We call this pattern *Indexed Row.* The matrix-vector multiplication runs as follows:

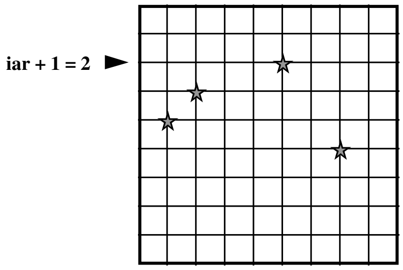


Figure 8.4: The storage pattern *Indexed Column*

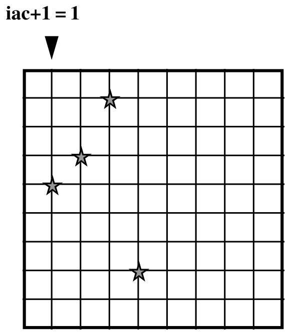


Figure 8.5: The storage pattern *Indexed Row*

$$\begin{aligned}&\text{for } k = 1 \text{ to } lvt \text{ do}\\ &\quad c_{indexl_k} = c_{indexl_k} + a_{index_k,iac+k} * y_{iacl+k} \qquad (8.8)\\ &\text{enddo}\end{aligned}$$

According to the *Indexed Column* pattern four load and one store operations are needed, but now the loading of the input vector is contiguous. This operation should not be used on a vector processor, since the row indices *may* contain equal values, which means that a recursion takes place.

To get the multiplication with A_4^T for the *Indexed Column* pattern the arguments of the input vector y and the output vector c in loop 8.7 have to be exchanged, so that logically the *Indexed Row* pattern is used. Analogously the multiplication by the *Indexed Row* pattern uses logically the *Index Column* pattern for the multiplication with A_5^T.

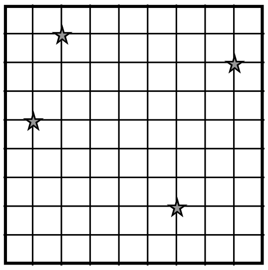


Figure 8.6: The storage pattern *Starry Sky*

For the execution on a vector processor (not on superscalar processors!) the index vector **must** be free of recursion for the *Indexed Row* as well as for the *Indexed Column* pattern, since we need multiplications with A *and* A^T. Additionally the data access by the index vector can produce bank conflicts. The creation of the index vectors requires a lot of attention and therefore it is very difficult to use these both storage patterns in practical applications. In LSOLPP it can be checked, if the index vectors are free of recursion.

8.4 Starry Sky

To catch all nonzero entries in the block matrix, which can not be processed efficiently by one of the remaining storage patterns, the *Starry Sky* pattern is used: For $k = 1, \ldots, lvt$ $a_{index_{k(row)}, index_{k(col)}}$ is nonzero. The matrix-vector multiplication is

$$\begin{array}{l} \text{for } k = 1 \text{ to } lvt \text{ do} \\ \quad c_{indexl_{k(row)}} = c_{indexl_{k(row)}} + a_{index_{k(row)}, index_{k(col)}} * y_{index_{k(col)}} \\ \text{enddo} \end{array} \tag{8.9}$$

This operation requires five load and one store operation. For the multiplication with A_6^T $index_{k(row)}$ and $index_{k(col)}$ are exchanged. For the vectorized processing $index_{k(row)}$ as well as $index_{k(row)}$ have to be free of recursion, so that in general the loop has to be split up. In LSOLPP it can be checked, if the index vectors are free of recursion.

8.5 Full Row and Full Column

In some applications the global matrix A is bordered by columns and/or rows, which are produced by secondary conditions, e.g. normalizations. Then a row $iar + 1$ in the

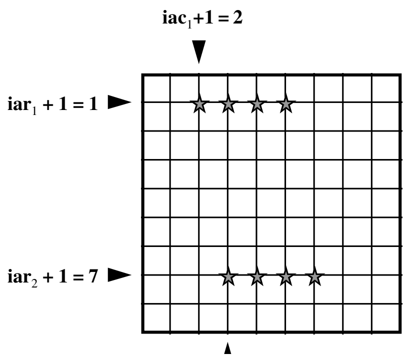


Figure 8.7: The storage pattern *Full Row*

matrix contains no nonzero elements and is stored optimally in the following pattern: for all columns $iac < ii \leq iac + lvt$ matrix element $a_{iar+1,ii}$ is stored (see Figure 8.7). We have to execute

$$\begin{aligned}&\text{for } k = 1 \text{ to } lvt \text{ do}\\ &\qquad c_{iarl+1} = c_{iarl+1} + a_{iar+1,iac+k} * y_{iacl+k}\\ &\text{enddo}\end{aligned} \tag{8.10}$$

which is a scalar product.

Naturally we have to introduce a *Full Column* pattern: for all rows $iar < ii \leq iar + lvt$ $a_{ii,iac+1}$ is stored and we get the following equation for the multiplication with A_8:

$$\begin{aligned}&\text{for } k = 1 \text{ to } lvt \text{ do}\\ &\qquad c_{iarl+k} = c_{iarl+k} + a_{iar+k,iac+1} * y_{iacl+1}\\ &\text{enddo}\end{aligned} \tag{8.11}$$

This is a contracting linked triad. For the multiplication with A_7^T the *Full Column* pattern is applied for the *Full Row* storage scheme and the other way round.

8.6 Packed Row and Packed Column

If there are too many gaps with zero elements in a row (column), it is more efficient to pack the row (column); now the submatrix $A_9^{i,j}$ contains packed rows with only nonzero elements; the integer vector *index* marks the position of the nonzero elements in the row (see Figure 8.9). It will never contain twice the same indices and therefore the modification of loop 8.10 is free of recursion:

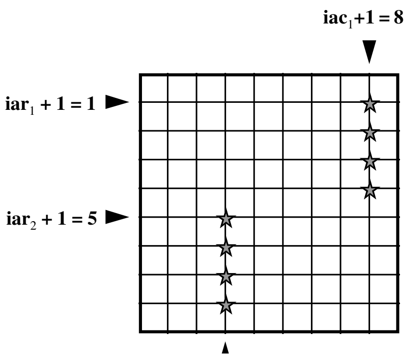


Figure 8.8: The storage pattern *Full Column*

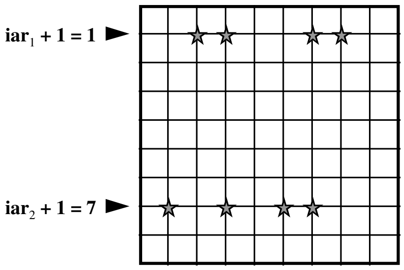


Figure 8.9: The storage pattern *Packed Row*

$$\begin{aligned}&\text{for } k = 1 \text{ to } lvt \text{ do}\\ &\qquad c_{iarl+1} = c_{iarl+1} + a_{iar+1,index_k} * y_{indexl_k}\\ &\text{enddo}\end{aligned} \tag{8.12}$$

which is a scalar product with indicated access.

Naturally we have to introduce a *Packed Column* pattern; the submatrix $A_1 0^{i,j}$ contains packed columns with only nonzero elements; the integer vector *index* marks the position of the nonzero elements in the columns (see Figure 8.10). It will never contain twice the same indices and therefore the modification of loop 8.11 is free of recursion:

$$\begin{aligned}&\text{for } k = 1 \text{ to } lvt \text{ do}\\ &\qquad c_{indexl_k} = c_{indexl_k} + a_{index_k,iac+1} * y_{iacl+1}\\ &\text{enddo}\end{aligned} \tag{8.13}$$

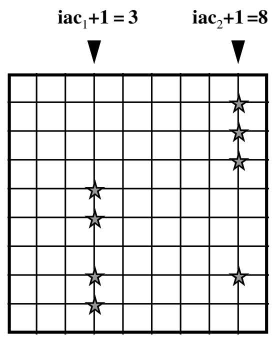


Figure 8.10: The storage pattern *Packed Column*

This is a contracting linked triad with indicated access. For the multiplication with A_9^T the *Packed Column* pattern is applied for the *Packed Row* storage scheme and the other way round.

8.7 The Storage Scheme for the Matrix and the Index Vectors

The matrix A is stored in the one-dimensional array MAT of length LMAT; all index vectors are stored in the one-dimensional integer array INDEX of length LINDEX. The pointer ADDA points to a single vector term within the array MAT and the pointers INDR and INDC point to the relating indices within the array INDEX. All information relating to the matrix A is stored in the two-dimensional integer array INFO of length (NVT,7), where NVT is the number of vector terms of the stored matrix A (see page 6 of Appendix MATRIX).

8.8 Optimization

The strategy to find the optimal storage pattern for a given matrix is to minimize the data access by index vectors and to maximize the operation length lvt. Naturally it is allowed to fill patterns by zeroes to get a efficient storage pattern, but too much additional zero entries will reduce the efficiency of the whole matrix-vector multplication. A good strategy is:

- look for all full rows and full columns,
- look for all 'long' diagonals; if a diagonal has a 'low' number of nonzero elements, use the storage pattern *Packed Diagonal*,

- look for all 'long' indexed rows,
- gather remaining entries into the storage pattern *Starry Sky*.

The actual meaning of 'long' and 'low' depends strictly on the used hardware configuration.

Here it is neccessary to make some remarks for the execution on cache architectures. The contiguous access to the matrix array MAT and the index array INDEX allows the optimal use of the loaded cache lines for these data. This is also true for the storage patterns with indexed access. But there is an additional effect: the input and result vector can be reused, if they are short enough to be kept in the cache. Then we also get a fast data access for the indexed operations. Thus on a parallel computer the number of unknowns per processor should be selected so that the complete input and result vector can be stored in the cache.

Bibliography

[1] R.E. Alcouffe, A. Brandt, J.E. Dendy, Jr., and J. W. Painter, *The multi-grid method for the diffusion equation with strongly discontinuous coefficients*, SIAM J. Sci. Stat. Comput. **2** (1981), no. 4, 430–454.

[2] G.M. Amdahl, *Validity of the single-processor approach to achieving large scale computing capabilities*, AFIPS Conference Proceedings, vol. 30, AFIP Press, Reston, Va., 1967, pp. 483–485.

[3] O. Axelsson, *A survey of preconditioned iterative methods for linear systems of algebraic equations*, BIT **1** (1985), 166–187.

[4] O. Axelsson and G. Lindskog, *On the rate of convergence of the preconditioned conjugate gradient method*, Numer. Math. **48** (1986), 499–523.

[5] O. Axelsson and P.S. Vassilevski, *A survey of multilevel preconditioned iterative methods*, BIT **29** (1989), 769–793.

[6] R. Beauwens, *Incomplete factorizations with S/P and modified S/P consistently ordered M-factors*, Incomplete Decompositions (ILU) - Algorithms, Theory, and Applications, Proceedings of the 8th GAMM-Seminar, Notes on Num. Fluid Mech., Vol. 41, January 24-26, Kiel, Germany (W. Hackbusch and G. Wittum, eds.), Vieweg, Braunschweig, 1993, pp. 22–31.

[7] ———, *Approximate factorizations with modified S/P consistently ordered M-factors*, J. of Num. Lin. Alg. with Appl. **1** (1994), no. 1, 3–17.

[8] ———, *Dynamic factorization algorithms*, Numerical Algebra (Er. Jiang, ed.), 1994, 92 Shangai International Numerical Algebra and its Applications Meeting, Shangai, October 26-30, 1992, pp. 40–47.

[9] H. Böpple, *Theoretical and experimental investigation of the DARE (data reduction) method*, Internal report 49/93, University of Karlsruhe, Computing Center, Postfach 6980, D-76128 Karlsruhe, Germany, 1993.

[10] A. Brandt, *Multi-level adaptive solutions to boundary-value problems*, Mathematics of Computation **31** (1977), no. 138, 333–390.

[11] C. Brezinski and M. Redivo-Zaglia, *Look-ahead in Bi-CGSTAB and other product methods for linear systems*, BIT **35** (1995), 169–201.

[12] C. Brezinski, M. Redivo-Zaglia, and H. Sadok, *Avoiding breakdown and near-breakdown in Lanczos type algorithms*, Numerical Algorithms **1** (1991), no. 3, 261–284.

[13] ———, *A breakdown-free Lanczos type algorithm for solving linear systems*, Numer. Math. **63** (1992), 29–38.

[14] C. Brezinski and H. Sadok, *Avoiding breakdown in the CGS algorithm*, Numerical Algorithms **1** (1991), no. 2, 199–206.

[15] W.L. Briggs, *A multigrid tutorial*, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1987.

[16] T.F. Chan, E. Gallopoulos, V. Simoncini, T. Szeto, and C.H. Tong, *QMR-CGSTAB: a quasi-minimal residual variant of the Bi-CGSTAB algorithm for non-symmetric systems*, SIAM J. Sci. Comput. **15** (1994), 338–347.

[17] E. Chow and Y. Saad, *Approximate inverse preconditioners for general sparse matrices*, Research Report UMSI 94/101, University of Minnesota, Supercomputer Institute, 1200 Washington Avenue South, Minneapolis, Minnesota 55415, 1994.

[18] J.D.F. Cosgrove, J.C. Diaz, and A. Griewank, *Approximate inverse preconditionings for sparse linear systems*, Intern. J. Computer Math. **44** (1992), 91–110.

[19] E.J. Craig, *The n-step iteration procedures*, Math. Phys. **34** (1955), 64–73.

[20] L.G.C. Crone and H.A. van der Vorst, *Communication aspects of the conjugate gradient method on distributed-memory machines*, Supercomputer **58** (1993), 4–9.

[21] E. de Sturler, *A parallel variant of GMRES(m)*, IMACS '91, Proceedings of the 13th IMACS World Congress on Computation and Applied Mathematics, July 22-26, Dublin, Ireland (R. Vichnevetsky and J.J.H. Miller, eds.), 1991, pp. 682–683.

[22] S. Demko, W.F. Moss, and P.W. Smith, *Decay rates for inverses of band matrices*, Mathematics of Computation **43** (1984), no. 168, 491–499.

[23] J. E. Dendy, Jr., *Black box multigrid for nonsymmetric problems*, Applied Mathematics and Computation **13** (1983), 261–283.

[24] I.S. Duff, M. Marrone, and G. Radicati, *A proposal for user level sparse BLAS*, Tech. report, Rutherford Appleton Lab., Oxfordshire, 1992.

[25] M. Eiermann, X. Li, and R.S. Varga, *On hybrid semi-iterative methods*, SIAM J. Numer. Anal. **26** (1989), no. 1, 152–168.

[26] M. Eiermann and W. Niethammer, *On the construction of semiiterative methods*, SIAM J. Numer. Anal. **20** (1983), no. 6, 1153–1160.

[27] M. Eiermann, W. Niethammer, and R.S. Varga, *A study of semiiterative methods for nonsymmetric systems of linear equations*, Numer. Math. **43** (1985), 505–533.

[28] M. Eiermann, R.S. Varga, and W. Niethammer, *Iterationsverfahren für nichtsymmetrische Gleichungssysteme und Approximationsmethoden im Komplexen*, Jber. d. Dt. Math.-Verein **89** (1987), 1–32.

[29] V. Faber and T. A. Manteuffel, *Orthogonal error methods*, SIAM J. Numer. Anal. **24** (1987), no. 1, 170–187.

[30] B. Fischer and R.W. Freund, *On adaptive weighted polynomial preconditioning for Hermitian positive definite matrices*, SIAM J. Sci. Comput. **15** (1994), 408–426.

[31] R. Fletcher, *Conjugate gradient methods for indefinite systems*, Proc. Dundee Biennial Conf. on Num. Anal. (G.A. Watson, ed.), Springer-Verlag, 1975, pp. 73–89.

[32] R.W. Freund, *On conjugate gradient type methods and polynomial preconditioners for a class of non-Hermitian matrices*, Numer. Math. **57** (1990), 285–312.

[33] ______, *Quasi-kernel polynomials and convergence results for quasi-minimal residual iterations*, Numerical Methods of Approximation Theory **9** (1992), 77–95.

[34] ______, *A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems*, SIAM J. Sci. Comput. **14** (1993), no. 2, 470–482.

[35] R.W. Freund, G.H. Golub, and N.M. Nachtigal, *Iterative solution of linear systems*, Acta Numerica (1992), 57–100.

[36] R.W. Freund, M.H. Gutknecht, and N.M. Nachtigal, *An implementation of the Look-Ahead Lanczos algorithm for non-Hermitian matrices*, SIAM J. Sci. Comput. **14** (1993), no. 1, 137–158.

[37] R.W. Freund and N.M. Nachtigal, *QMR: a quasi-minimal residual method for non-Hermitian linear systems*, Numer. Math. **60** (1991), 315–339.

[38] R.W. Freund and T. Szeto, *A quasi-minimal residual squared algorithm for non-hermitian linear systems*, RIACS Technical Report 91/26, Research Institute for Advanced Computer Science, NASA Ames Research Center, 1991, Proceedings of the Copper Mountain Conference on Iterative Methods, 1992.

[39] V.M. Fridman, *The method of minimum iterations with minimum errors for a system of linear algebraic equations with a symmetrical matrix*, USSR Comput. Math. and Math. Phys. **2** (1963), 362–363.

[40] S. Fujino, *Memory bank conflict on some vector supercomputers*, Proc. of the 6th Annual European Computer Conference on Computer System and Software Engineering, Hague, The Netherlands, IEEE Computer Society, 1992, pp. 359–364.

[41] S. Fujino and S. Doi, *Optimizing multicolor ICCG methods on some vectorcomputers*, Iterative Methods in linear Algebra (R. Beauwens and P. de Groen, eds.), Elservier Science Publishers B.V. (North-Holland), 1992, pp. 349–358.

[42] S. Fujino, M. Mori, and T. Takeuchi, *Performance of hyperplane ordering on vector computers*, Journal of Computational and Applied Mathematics **38** (1991), 125–136.

[43] S. Fujino and T. Takeuchi, *ILU factorization well suited to the vector processor using a variant of the 5-point difference scheme*, Comp. Phys. Comm. **85** (1995), no. 3, 371–381.

[44] G. Golubovici and C. Popa, *Interpolation and related coarsening techniques for the algebraic multigrid methods*, Proceedings of the Fourth European Conference on Multigrid Methods, July 1993, Amsterdam (Berlin), Intern. Series of Numer. Math, vol. 116, Birkhäuser Verlag, 1994, pp. 201–213.

[45] A. Greenbaum, *Comparison of splittings used with the conjugate gradient algorithm*, Numer. Math. **33** (1979), 181–194.

[46] L. Gross, C. Roll, and W. Schönauer, *Chances and limits for the application of the FEM-solver VECFEM to general partial differential equations*, Advances in Computer Methods for Partial Differential Equations VII (R. Vichnevetsky, Knight, and Richter, eds.), IMACS, 1992, pp. 306–312.

[47] ______, *VECFEM for mixed finite elements*, Internal report 50/93, University of Karlsruhe, Computing Center, Postfach 6980, D-76128 Karlsruhe, Germany, 1993.

[48] M. Grote and H.D. Simon, *Parallel preconditioning and approximate inverses on the connection machine*, Research Report RNR-92-020, NAS Systems Division, NASA Ames Research Center, 1992, Proceedings of the Scalable High Performance Computer Conference 1992, IEEE Computer Society Press, 1992, pp. 76-83.

[49] J.L. Gustafson, *Reevaluating Amdahl's law*, Communications of the ACM **31** (1988), no. 5, 532–533.

[50] I. Gustafsson, *A class of first order factorization methods*, BIT **18** (1978), no. 1, 142–156.

[51] M.H. Gutknecht, *A completed theory of the unsymmetric Lanczos process and related algorithms, part 1*, SIAM J. Matrix Anal. **13** (1992), no. 2, 594–639.

[52] ———, *Changing the norm in conjugate gradient type algorithms*, SIAM J. Numer. Anal. **30** (1993), no. 1, 40–56.

[53] ———, *Variants of BiCGStab for matrices with complex spectrum*, SIAM J. Sci. Comput. **14** (1993), 1020–1033.

[54] W. Hackbusch, *Multigrid methods and applications*, Springer-Verlag, Berlin, Heidelberg, New York, 1985.

[55] W. Hackbusch and U. Trottenberg (eds.), *Multigrid methods, Proceedings of the conference held at Köln-Porz, 1981*, Lecture Notes in Mathematics, vol. 960, Berlin, Heidelberg, New York, Springer-Verlag, 1982.

[56] W. Hackbusch and U. Trottenberg (eds.), *Multigrid methods iii*, International Series of Numerical Mathematics, vol. 98, Birkhäuser Verlag, Basel, Boston, Berlin, 1991.

[57] H. Häfner and W. Schönauer, *Micro-mesurements of a supercomputer and models for memory contention*, High Performance Computing II (M. Durand and F. El Dabaghi, eds.), Elsevier Science Publishers B. V. (North-Holland), 1991, pp. 169–180.

[58] ———, *Kommunikation auf verschiedenen Parallelrechnern, Auswirkung auf die Programmierung*, Tagungsband zum 3. ODIN-Symposium Karlsruhe: Softwareentwicklung für Supercomputer, 1994, pp. 195–206.

[59] M. Hanke, M. Hochbruck, and W. Niethammer, *Experiments with Krylov subspace methods on a massively parallel computer*, Appl. of Math. **38** (1993), 440–451.

[60] D.R. Hare, C.R. Johnson, D.D. Olesky, and P. van den Driessche, *Sparsity analysis of the QR factorization*, SIAM J. Matrix Anal. Appl. **14** (1993), no. 3, 655–669.

[61] M.R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, J. Res. Nat. Bur. Standards **49** (1952), 409–435.

[62] T. Huckle and M. Grote, *A new approach to parallel preconditioning with sparse approximate inverses*, Internal Report SCCM94-03, Stanford University, 1994.

[63] V. P. Il'in, *Iterative incomplete factorization methods*, World Scientific Publishing Co. Pte. Ltd, Singapore, New Jersey, London, Hong Kong, 1992.

[64] O.G. Johnson, C.A. Micchelli, and G. Paul, *Polynomial preconditioners for conjugate gradient calculations*, SIAM J. Numer. Anal. **20** (1983), no. 2, 362–376.

[65] M.T. Jones and P.E. Plassmann, *Scalable iterative solution of sparse linear systems*, Parallel Computing **20** (1994), 753–773.

[66] I.E. Kaporin, L.Y. Kolotilina, and A.Y. Yeremin, *Block SSOR preconditionings for high-order 3D FE systems. II. incomplete BSSOR preconditionings*, Linear Algebra and its Applications **154–156** (1991), 647–674.

[67] L.Y. Kolotilina and A.Y. Yeremin, *Factorized sparse approximate inverse preconditionings i. theory*, SIAM J. Matrix Anal. Appl. **14** (1993), no. 1, 45–58.

[68] C. Lanczos, *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*, J. Res. Nat. Bur. Standards **45** (1950), 255–282.

[69] ———, *Solution of systems of linear equations by minimized iterations*, J. Res. Nat. Bur. Standards **49** (1952), 33–53.

[70] G. I. Marchuk, *Methods for calculating the nuclear reactors*, Atomizdat, Moskva, 1958, in Russian.

[71] U. Meier Yang, *Preconditioned conjugate gradient-like methods for nonsymmetric linear systems*, Tech. Report CSRD 1210, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, 465 CSRL-1308 West Main Street, Urbana, IL 61801-2307, 1992.

[72] J.A. Meijerink and H.A. van der Vorst, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix*, Math. Comp. **31** (1977), 148–162.

[73] W. Niethammer and W. Schempp, *On the construction of iteration methods for linear equations in Banach spaces by summation methods*, Aequat. Math. **5** (1970), 273–284.

[74] W. Niethammer and R.S. Varga, *The analysis of k-step iterative methods for linear systems from summability theory*, Numer. Math. **41** (1983), 177–206.

[75] Y. Notay, *Ordering methods for approximate factorization preconditioning*, Preprint, Université Libre de Bruxelles, Service de Métrologie Nucléaire, 1993.

[76] ———, *DRIC: A dynamic version of the RIC method*, J. of Num. Lin. Alg. with Appl. **1** (1994), 511–532.

[77] T. Oliphant, *An extrapolation procedure for solving linear systems*, Quart. Appl. Math. **20** (1962), 257–267.

[78] C.C. Paige and M.A. Saunders, *Solution of sparse indefinite systems of linear equations*, SIAM J. Numer. Anal. **12** (1975), no. 4, 617–629.

[79] C. Popa, *A coarsening algorithm for the algebraic multigrid method*, Studii si Cercetari Matematice **46** (1994), no. 6, 613–624.

[80] ———, *Preconditioning for the fulfilment of the approximation assumption in the algebraic multigrid method*, Tech. report, University of Karlsruhe, Computing Center, Postfach 6980, D-76128 Karlsruhe, Germany, 1994, to appear in Studia Univ. Babes-Bolyai, Cluj-Napoca, Romania, vol. 40.

[81] L. Portugal, F. Bastos, J. Judice, J. Paixão, and T. Terlaky, *An investigation of interior point algorithms for the linear tranportation problem*, Report 93-100, Delft University of Technology, Faculty of Technical Mathematics and Informatics, 1993.

[82] J.K. Reid, *On the method of conjugate gradients for the solution of large sparse systems of linear equations*, Large Sparse Sets of Linear Equations (J.K. Reid, ed.), Academic Press, New York, 1971, pp. 231–254.

[83] M. Rozložník and R. Weiss, *On stable implementations of the generalized minimal error method*, Internal Report 56/95, University of Karlsruhe, Computing Center, Postfach 6980, D-76128 Karlsruhe, Germany, 1995.

[84] P. Rózsa, R. Bevilacqua, F. Romani, and P. Favati, *On band matrices and their inverses*, Linear Algebra and its Applications **150** (1991), 287–295.

[85] Y. Saad, *Practical use of polynomial preconditionings for the conjugate gradient method*, Research Report YALEU/DCS/RR-282, Yale University, Department of Computer Science, 1983.

[86] ———, *Preconditioning techniques for nonsymmetric and indefinite linear systems*, J. Comput. Appl. Math. **24** (1988), 89–105.

[87] ———, *ILUM: A parallel multi-elimination ILU preconditioner for general sparse matrices*, Research Report UMSI 92/241, University of Minnesota, Supercomputer Institute, 1200 Washington Avenue South, Minneapolis, Minnesota 55415, 1992.

[88] ———, *ILUT: A dual threshold incomplete LU factorization*, Research Report UMSI 92/38, University of Minnesota, Supercomputer Institute, 1200 Washington Avenue South, Minneapolis, Minnesota 55415, 1992.

[89] ———, *A flexible inner-outer preconditioned GMRES algorithm*, SIAM J. Sci. Comput. **14** (1993), no. 2, 461–469.

[90] Y. Saad and M.H. Schultz, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Stat. Comp. **7** (1986), no. 3, 856–869.

[91] M. Schmauder and W. Schönauer, *The CADSOL 3-D report*, Internal report 51/94, University of Karlsruhe, Computing Center, Postfach 6980, D-76128 Karlsruhe, Germany, 1994.

[92] M. Schmauder, R. Weiss, and W. Schönauer, *The CADSOL program package (version 1.1)*, Internal report 46/92, University of Karlsruhe, Computing Center, Postfach 6980, D-76128 Karlsruhe, Germany, 1992.

[93] W. Schönauer, *Scientific computing on vector computers*, North-Holland, Amsterdam, New York, Oxford, Tokyo, 1987.

[94] W. Schönauer and H. Häfner, *Supercomputers: The hardware , the architecture*, Proceedings of the 13th IMACS World Congress on Computation and Applied Mathematics, Dublin, Ireland, July 22-26 (R. Vichnevetsky and J.J.H. Miller, eds.), 1991, pp. 725–727.

[95] ———, *Explaining the gap between theoretical peak performance and real performance for supercomputer architectures*, Scientific Programming **3** (1994), 157–168.

[96] ———, *Supercomputer architectures and their bottlenecks*, Computer & Fluids **15** (1994), no. 1, 411–417.

[97] W. Schönauer, H. Häfner, L. Gross, M. Schmauder, and R. Weiss, *Parallelization of black-box software for elliptic and parabolic PDEs*, Preprint, University of Karlsruhe, Computing Center, Postfach 6980, D-76128 Karlsruhe, Germany, 1995.

[98] W. Schönauer, H. Müller, and E. Schnepf, *Pseudo-residual type methods for the iterative solution of large linear systems on vector computers*, Parallel Computing 85 (M. Feilmeier, J. Joubert, and U. Schendel, eds.), Elsevier Science Publishers B.V., North-Holland, 1986, pp. 193–198.

[99] W. Schönauer and K. Raith, *A polyalgorithm with diagonal storing for the solution of very large indefinite linear banded systems on vector computers*, Proceedings of the 10th IMACS World Congress on System Simulation and Scientific Computation, Montreal, 1982, pp. 326–328.

[100] W. Schönauer, M. Schlichte, and R. Weiss, *Wrong ways and promising ways towards robust and efficient iterative linear solvers*, Advances in Computer Methods for Partial Differential Equations VI, Publ. IMACS, 1987, pp. 7–14.

[101] ———, *Numerical experiments with data reduction (DARE) methods for the solution of large linear systems on vector computers*, IMACS 1988, 12th World Congress on Scientific Computation, Proceedings, Paris (R. Vichnevetsky, P. Borne, and J. Vignes, eds.), 1988, pp. 227–229.

[102] W. Schönauer, E. Schnepf, and H. Müller, *The FIDISOL program package*, Internal report 27/85, University of Karlsruhe, Computing Center, Postfach 6980, D-76128 Karlsruhe, Germany, 1985.

[103] W. Schönauer and R. Weiss, *An engineering approach to generalized conjugate gradient methods and beyond*, Appl. Numer. Math. **19** (1995), no. 3, 175–206, Special Issue on Iterative Methods for Linear Equations.

[104] W. Schönauer, R. Weiss, and M. Schlichte, *The basic ideas of the data reduction (DARE) method for the solution of large linear systems on vector computers*, ZAMM **69** (1989), T 184–T 185.

[105] W. Schönauer, R. Weiss, and P. Sternecker, *Polyalgorithms with automatic method selection for the iterative solution of linear equations and eigenproblems*, Programming Environments for High-Level Problem Solving (P.W. Gaffney and E.N. Houstis, eds.), North-Holland, Amsterdam, 1992, pp. 57–67.

[106] J. R. Shewchuk, *Conjugate gradient method without the agonizing pain*, Internal Report CMU-CS-94-125, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA, 1994.

[107] H.D. Simon, *Parallel preconditioners for the connection machine*, Proceedings on the Fourth SIAM Conference on Parallel Processing for Scientific Computing (J. Dongarra, P. Messina, D.C. Sorensen, and R.G. Voigt, eds.), 1990, pp. 389–390.

[108] G.L.G. Sleijpen and D.R. Fokkema, *BiCGStab(l) for linear equations involving unsymmetric matrices with complex spectrum*, Electronic Transactions on Numer. Anal. (ETNA) **1** (1993), 11–32.

[109] G.L.G. Sleijpen, H.A. van der Vorst, and D.R. Fokkema, *BiCGSTAB(l) and other hybrid Bi-CG methods*, Numerical Algorithms **7** (1994), 75–109.

[110] P. Sonneveld, *CGS, a fast Lanczos-type solver for nonsymmetric linear systems*, SIAM J. Sci. Stat. Comp. **10** (1989), no. 1, 36–52.

[111] P. Sternecker and W. Schönauer, *VECFEM: A finite element kernel program with optimal data structures for vector computers*, Internal report 34/88, University of Karlsruhe, Computing Center, Postfach 6980, D-76128 Karlsruhe, Germany, 1988.

[112] J. Stoer and R.W. Freund, *On the solution of large indefinite systems of linear equations by conjugate gradient algorithms*, Proceedings of the International Symposium on Computing Methods in Applied Sciences and Engineering V. (R. Glowinski and J. L. Lions, eds.), North-Holland, 1982, pp. 35–53.

[113] G. Temple, *The general theory of relaxation methods applied to linear systems*, Proc. Roy. Soc. (London) **A169** (1939), no. A939, 476–500.

[114] H.A. van der Vorst, *Guidelines for the usage of incomplete decompositions in solving sets of linear equations as they occur in practical problems*, J. Comput. Phys. **44** (1981), 134–155.

[115] ______, *Preconditioning by incomplete decompositions*, Ph.D. thesis, University of Utrecht, The Netherlands, 1982, ACCU Series 32.

[116] ______, *High performance preconditioning*, SIAM J. Sci. Stat. Comp. **10** (1989), no. 6, 1174–1185.

[117] ______, *ICCG and related methods for 3D problems on vector computers*, Comp. Phys. Comm. **53** (1989), 223–235.

[118] ______, *BI-CGSTAB: A fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput. **13** (1992), no. 2, 631–644.

[119] R.S. Varga, *Factorization and normalized iterative methods*, Boundary Problems in Differential Equations (R. Langer, ed.), University of Wisconsin Press, Englewood Cliffs, 1960, pp. 121–142.

[120] P.K.W. Vinsome, *ORTHOMIN, an iterative method for solving sparse sets of simultaneous linear equations, paper SPE 5739*, Symp. Numer. Simulation of Reservoir Performance of SPE of AIME, 4th, Los Angeles, California, 1976.

[121] C. Vuik, *Further experiences with GMRESR*, Supercomputer **55** (1993), 13–27.

[122] H.F. Walker, *Implementations of the GMRES method*, Computer Physics Communications **53** (1989), 311–320.

[123] ______, *Residual smoothing and peak/plateau behavior in Krylov subspace methods*, Appl. Numer. Math. **19** (1995), no. 3, 279–286, Special Issue on Iterative Methods for Linear Equations.

[124] R. Weiss, *Convergence behavior of generalized conjugate gradient methods*, Internal report 43/90, University of Karlsruhe, Computing Center, Postfach 6980, D-76128 Karlsruhe, Germany, 1990.

[125] ______, *Error-minimizing Krylov subspace methods*, SIAM J. Sci. Comput. **15** (1994), no. 3, 511–527.

[126] ______, *Properties of generalized conjugate gradient methods*, Num. Lin. Alg. with Appl. **1** (1994), no. 1, 45–63.

[127] ______, *Relations between smoothing and QMR*, Internal report 53/94, University of Karlsruhe, Computing Center, Postfach 6980, D-76128 Karlsruhe, Germany, 1994.

[128] R. Weiss, H. Häfner, and W. Schönauer, *Tuning the matrix-vector multiplication in diagonal form*, Proceedings of Parallel Computing 89, Leiden (G.R. Joubert, D.J. Evans, and F.J. Peters, eds.), Elsevier Science Publishers B. V., North-Holland, 1990, pp. 93–98.

[129] R. Weiss, A. Meroth, and P. Levin, *Iterative solution of boundary element approximations for electric and magnetic fields*, Intern. J. of Num. Modelling: Electronic Networks, Devices and Fields **7** (1994), 357–370.

[130] R. Weiss, C. Roll, H. Häfner, L. Gross, M. Schmauder, and W. Schönauer, *Parallelisierung von ODIN-Software für iterative Gleichungslöser*, Tagungsband zum 3. ODIN-Symposium Karlsruhe: Softwareentwicklung für Supercomputer (A. Schreiner and E. Schnepf, eds.), 1994, pp. 117–128.

[131] R. Weiss and W. Schönauer, *Data reduction (DARE) preconditioning for generalized conjugate gradient methods*, Preconditioned Conjugate Gradient Methods (A. Dold, B. Eckmann, and F. Takens, eds.), Lecture Notes in Mathematics, vol. 1457, Springer-Verlag, Berlin, Heidelberg, 1990, pp. 137–153.

[132] ———, *Solution of partial differential equations by means of data reduction (DARE)*, Multigrid Methods: Special Topics and Applications II, GMD-Studien Nr. 189, GMD, St. Augustin, 1991, pp. 361–372.

[133] ———, *Preconditioned conjugate gradient methods: What do we have and what do we need?*, Advances in Computer Methods for Partial Differential Equations VII (R. Vichnevetsky, D. Knight, and G. Richter, eds.), IMACS, 1992, pp. 806–812.

[134] G. Wittum and F. Liebau, *On truncated incomplete decompositions*, BIT **29** (1989), 719–740.

[135] F. Yamamoto and S. Takahashi, *Vectorized LUdecomposition algorithms for large-scale circuit simulation*, IEEE Transactions on CAD **CAD-4** (1985), no. 3, 232–239.

[136] Q. Ye, *A breakdown-free variation of the nonsymmetric Lanczos algorithms*, Math. of Comput. **62** (1994), no. 205, 179–207.

[137] D.M. Young, *Iterative solution of large linear systems*, Academic Press, New York, San Francisco, London, 1971.

[138] S.-L. Zhang, *GPBi-CG: Generalized product-type methods based on Bi-CG for solving nonsymmetric linear systems*, SIAM J. Sci. Comput. **17** (1996), no. 6, To appear.

[139] L. Zhou and H.F. Walker, *Residual smoothing techniques for iterative methods*, SIAM J. Sci. Comput. **15** (1994), no. 2, 297–312.

[140] M. Zimmermann, *Projektionsverfahren zur Präkonditionierung von verallgemeinerten CG-Verfahren*, Internal report 55/95, University of Karlsruhe, Computing Center, Postfach 6980, D-76128 Karlsruhe, Germany, 1995, Diploma thesis.

List of Figures

List of Tables

List of Algorithms

Index