Numerical methods for large algebraic systems (wi4010)

Dr.ir. C. Vuik

2004

Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Department of Applied Mathematical Analysis

Contents

1 Direct solution methods for linear systems

1.1 Introduction

In this chapter we give direct solution methods to solve a linear system of equations. The idea is based on elimination and given in Section 1.2. However using this method in practice shows that it has some drawbacks. First round off errors can spoil the result, second for simple problems the method may break down. To study this and give a better method we first define some basic notions on distances in $I\!R^n$ and floating point numbers (Section 1.3). In Section 1.4 we start with a theoretical result: how the solution may be changed if the matrix and right-hand side are transformed from reals to floating point numbers. Thereafter the properties of the Gaussian elimination process with respect to rounding errors are given. This leads to a more stable process where pivoting is introduced. Furthermore, the solution can be improved by a small number of iterations. Systems of equations arising from partial differential equations have certain properties which can be used to optimize the direct solution method. In Section 1.6 symmetric positive definite systems are considered. Finally, matrices originated from discretized PDE's are very sparse, which means that the number of nonzeroes in a row is small with respect to the dimensions of the matrix. Special methods to use this are given in Section 1.7 for structured problems (finite differences or finite volumes) and in Section 1.8 for unstructured problems (finite elements).

1.2 The Gaussian elimination method

In many numerical computations one has to solve a system of linear equations $Ax = b$. In this chapter we describe the method of Gaussian elimination, the algorithm of choice when A is square nonsingular, dense, and relatively small.

Computing the LU decomposition

In this section we show how Gauss transformations $M_1, \dots, M_{n-1}$ can be found such that the matrix U given by

$$M_{n-1}M_{n-2}\dots M_2M_1A = U$$

is upper triangular. In deriving the algorithm for computing the M_i we suppose that the Gauss transformations $M_1, \dots, M_{k-1}$ are determined such that

$$A^{(k-1)} = M_{k-1}\dots M_1A = \begin{array}{cc} \left[\begin{array}{cc} A_{11}^{(k-1)} & A_{12}^{(k-1)} \\ & \\ 0 & A_{22}^{(k-1)} \end{array}\right] & \begin{array}{c} k-1 \\ \\ n-k+1 \end{array} \\ \begin{array}{cc} k-1 & n-k+1 \end{array} & \end{array}$$

where $A_{11}^{(k-1)}$ is upper triangular. If

$$A_{22}^{(k-1)} = \left[\begin{array}{ccc} a_{kk}^{(k-1)} & \dots & a_{kn}^{(k-1)} \\ \vdots & & \vdots \\ a_{nk}^{(k-1)} & \dots & a_{nn}^{(k-1)} \end{array}\right]$$

and $a_{kk}^{(k-1)}$ is nonzero, then the multipliers

$$\ell_{ik} = a_{ik}^{(k-1)}/a_{kk}^{(k-1)} \qquad i = k+1, \dots, n$$

are defined. It follows that if $M_k = I - \alpha^{(k)}\ e_k^T$ where

$$\alpha^{(k)} = (0, \dots, 0, \ell_{k+1,k}, \dots, \ell_{nk})^T,$$

then

$$A^{(k)} = M_k\ A^{(k-1)} = \begin{array}{cc} \left[\begin{array}{cc} A_{11}^{(k)} & A_{12}^{(k)} \\ & \\ 0 & A_{22}^{(k)} \end{array}\right] & \begin{array}{c} k \\ \\ n-k \end{array} \\ \begin{array}{cc} k & n-k \end{array} & \end{array}$$

with $A_{11}^{(k)}$ upper triangular. This illustrates the k-th step of Gaussian elimination. If the pivots $a_{kk}^{(k-1)} \neq 0$ for $k = 1, \dots, n-1$ then

$$A^{(n-1)} = M_{n-1} \dots M_1\ A = U.$$

The inverse of $M_{n-1} \dots M_1$ can be given by

$$L = (M_{n-1} \dots M_1)^{-1} = \prod_{i=1}^{n-1} \left(I + \alpha^{(i)} e_i^T \right) = I + \sum_{i=1}^{n-1} \alpha^{(i)} e_i^T.$$

This implies that $A = LU$ and the matrix L is lower triangular and diag $(L) = I$.

The solution of the system $Ax = b$ is easy, if the LU decomposition is obtained. The solution of $LUx = b$ can be splitted into two parts: first the solution of the lower triangular system $Ly = b$ and then the solution of the upper triangular system $Ux = y$.

Algorithms

In a computer program, the entries in A can be overwritten with the corresponding entries of L and U as they are produced.

Gaussian elimination algorithm

Given $A \in I\!\!R^{n \times n}$ the following algorithm computes the factorization $A = LU$. The element a_{ij} is overwritten by l_{ij} if $i > j$ and by u_{ij} if $i \leq j$.

```
for $k = 1, \dots, n-1$ do
    if $a_{kk} = 0$ then
        quit
    else
        for $i = k+1, \dots, n$ do
            $\eta := a_{ik}/a_{kk}$
            $a_{ik} = \eta$
            for $j = k+1, \dots, n$
                $a_{ij} := a_{ij} - \eta\ a_{kj}$
            end for
        end for
    end if
end for
```

Given an $n \times n$ nonsingular lower triangular matrix L and $b \in I\!\!R^n$, the following algorithm finds $y \in I\!\!R^n$ such that $Ly = b$.

Forward substitution algorithm

```
for $i = 1, \ldots, n$ do
    $y_i := b_i$
    for $j = 1, \ldots, i-1$ do
        $y_i := y_i - \ell_{ij}\, y_j$
    end for
    $y_i := y_i/\ell_{ii}$
end for
```

Given an $n \times n$ nonsingular upper triangular matrix U and $y \in I\!R^n$, the following algorithm finds $x \in I\!R^n$ such that $Ux = y$.

Back substitution algorithm

```
for $i = n, \ldots, 1$ do
    $x_i := y_i$
    for $j = i+1, \ldots, n$ do
        $x_i := x_i - u_{ij}\, x_j$
    end for
    $x_i := x_i/u_{ii}$.
end for
```

To quantify the amount of arithmetic in these algorithms we define the notion flop. A flop is one floating point operation. It can be shown that the Gaussian elimination algorithm costs $2n^3/3$ flops, whereas both the forward substitution and back substitution costs n^2 flops.

It now seems that every system of linear equations can be solved. However in a practical computation many problems remain. As a first example consider the following equations:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

These equations are easily solved without Gaussian elimination. However, when one uses Gaussian elimination the first Gauss transformation does not exist and the algorithm breaks down.

As a second example we consider the effect of rounding errors. Suppose the following system is given:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0.999 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2 \\ 1.999 \end{pmatrix}.$$

The exact solution is $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$. Suppose that the right-hand side $\begin{pmatrix} 2 \\ 1.999 \end{pmatrix}$ is slightly changed to $\begin{pmatrix} 2 \\ 2 \end{pmatrix}$, because the computer can only use numbers with less than three digits. The approximate solution is then given by $\begin{pmatrix} 2 \\ 0 \end{pmatrix}$ which is totally different from the exact solution.

These problems motivates us to have a closer look at the Gaussian elimination process. For this reason we state some definitions, in Section 1.3, to measure the distance between two vectors. Thereafter, in Section 1.4, we investigate the behavior of Gaussian elimination with respect to rounding errors.

1.3 Norms and floating point numbers

In order to measure the distance of a perturbed vector to the exact vector, norms are introduced. The perturbations can be originated from errors in measurements or rounding errors during a computation.

Vector norms

A vector norm on $I\!R^n$ is a function $\|.\| : I\!R^n \to I\!R$ that satisfies the following properties:

i) $\|x\| \geq 0 \qquad x \in I\!R^n$, and $\|x\| = 0 \Longleftrightarrow x = 0$,

ii) $\|x+y\| \leq \|x\| + \|y\| \qquad x, y \in I\!R^n$,

iii) $\|\alpha x\| = |\alpha| \; \|x\| \qquad \alpha \in I\!R \,,\; x \in I\!R^n$.

An important class of vector norms are the so-called p-norms (Hölder norms) defined by

$$\|x\|_p = (|x_1|^p + \ldots + |x_n|^p)^{1/p} \qquad p \geq 1.$$

The 1,2, and ∞ norms are the most commonly used

$$\begin{aligned} &\|x\|_1 = |x_1| + \ldots + |x_n|, \\ &\|x\|_2 = (x_1^2 + \ldots + x_n^2)^{1/2} = (x^T x)^{1/2}, \\ &\|x\|_\infty = \max_{1 \leq i \leq n} \quad |x_i|. \end{aligned}$$

Inner product

The inner product is a function (.,.): $I\!R^n \times I\!R^n \to I\!R$ that satisfies the following properties:

i) $(x+y, z) = (x, z) + (y, z), \qquad x, y, z \in I\!R^n$,

ii) $(\alpha x, y) = \alpha(x, y), \qquad \alpha \in I\!R,\; x, y \in I\!R^n$,

iii) $(x, x) \geq 0, \qquad x \in I\!R^n$,

iv) $(x, x) = 0 \Longleftrightarrow x = 0, \qquad x \in I\!R^n$.

Matrix norms

The analysis of matrix algorithms frequently requires use of matrix norms. For example, the quality of a linear system solver may be poor if the matrix of coefficients is "nearly singular". To quantify the notion of near-singularity we need a measure of distance on the space of matrices. Matrix norms provide that measure.

The most commonly used matrix norms in numerical linear algebra are the p-norms induced by the vector p-norms

$$\|A\|_p = \sup_{x \neq 0} \frac{\|Ax\|_p}{\|x\|_p} = \max_{\|x\|_p = 1} \quad \|Ax\|_p \quad p \geq 1.$$

Properties

The vector and matrix p-norms have the following properties

- $\|AB\|_p \leq \|A\|_p \|B\|_p \qquad A \in I\!R^{m\times n} \quad , \quad B \in I\!R^{n\times q}$
- $\|A\|_1 = \max\limits_{1\leq j\leq n} \sum\limits_{i=1}^{m} |a_{ij}| \qquad A \in I\!R^{m\times n}$ maximal absolute column sum
- $\|A\|_\infty = \max\limits_{1\leq i\leq m} \sum\limits_{j=1}^{n} |a_{ij}| \qquad A \in I\!R^{m\times n}$ maximal absolute row sum
- $\|A\|_2$ is equal to the square root of the maximal eigenvalue of A^TA.
- The 2 norm is invariant with respect to orthogonal transformations. A matrix Q is orthogonal if and only if $Q^TQ = I$, with $Q \in I\!R^{n\times n}$. So for all orthogonal Q and Z of appropriate dimensions we have

$$\|QAZ\|_2 = \|A\|_2 \ .$$

The floating point numbers

Each arithmetic operation performed on a computer is generally affected by rounding errors. These errors arise because the machine hardware can only represent a subset of the real numbers. This subset is denoted by F and its elements are called floating point numbers. The floating point number system on a particular computer is characterized by four integers: the base β, the precision t, and the exponent range $[L, U]$. F consists of all numbers f of the form

$$f = \pm .d_1 d_2 \ldots d_t \times \beta^e, \qquad 0 \leq d_i < \beta \ , \ \ d_1 \neq 0 \ , \ \ L \leq e \leq U,$$

together with zero. Notice that for a nonzero $f \in F$ we have $m \leq |f| \leq M$ where $m = \beta^{L-1}$ and $M = \beta^U(1-\beta^{-t})$. To have a model of computer arithmetic the set G is defined by

$$G = \{x \in I\!R | m \leq |x| \leq M\} \cup \{0\} \,,$$

and the operator fl(oat): $G \rightarrow F$, where fl maps a real number from G to a floating point number by rounding away from zero. The fl operator satisfies the following equation

$$fl(x) = x(1+\epsilon), \qquad |\epsilon| \leq u, \qquad x \in G,$$

where u (unit roundoff) is defined by $\qquad u = \frac{1}{2}\beta^{1-t}$

Let a and b be elements of F. If $|a * b| \notin G$ then an arithmetic fault occurs implying either overflow ($|a*b| > M$) or underflow ($0 < |a*b| < m$). If $a*b \in G$ then we assume that the computed version of $a*b$ is given by $fl(a*b)$ which is equal to $(a*b)(1+\epsilon)$ with $|\epsilon| < u$. This shows that there is a small relative error associated with individual arithmetic operations. This is not necessarily the case when a sequence of operations is involved. Then catastrophic cancellation can occur. This term refers to the extreme loss of correct significant digits when small numbers are additively computed from large numbers.

1.4 Error analysis of Gaussian elimination

Before we proceed with the error analysis of Gaussian elimination we consider how perturbations in A and b affect the solution x. The condition number $K_p(A)$, for a nonsingular matrix A, is defined by $K_p(A) = \|A\|_p \|A^{-1}\|_p$.

Theorem 1.1 *Suppose* $Ax = b$, $A \in I\!R^{n\times n}$ *and* A *is nonsingular,* $0 \neq b \in I\!R^n$, $(A+\Delta A)y = b+\Delta b$, $\Delta A \in I\!R^{n\times n}$, $\Delta b \in I\!R^n$ *with* $\|\Delta A\|_p \le \delta\|A\|_p$ *and* $\|\Delta b\|_p \le \delta\|b\|_p$. *If* $K_p(A)\delta = r < 1$ *then* $A+\Delta A$ *is nonsingular and*

$$\frac{\|x-y\|_p}{\|x\|_p} \le \frac{2\delta}{1-r}K_p(A).$$

<u>Proof</u>: see [37], p.83.

Consider the nearly ideal situation in which no round off occurs during the solution process except when A and b are stored. Thus if $fl(b) = b+\Delta b$ and the stored matrix $fl(A) = A+\Delta A$ is nonsingular, then we are assuming that the computed solution $\hat{x}$ satisfies.

$$(A+\Delta A)\hat{x} = b+\Delta b \quad \|\Delta A\|_\infty \le u\|A\|_\infty, \quad \|\Delta b\|_\infty \le u\|b\|_\infty.$$

If $K_\infty(A)u \le \frac{1}{2}$, then by using Theorem 1.1 it can be shown that

$$\frac{\|x-\hat{x}\|_\infty}{\|x\|_\infty} \le 4uK_\infty(A)\ . \tag{1}$$

No general ∞-norm error analysis of a linear equation solver that requires the storage of A and b can render sharper bounds. As a consequence, we cannot justifiably criticize an algorithm for returning an inaccurate x if A is ill conditioned relative to the machine precision, e.g., $uK_\infty(A) \approx \frac{1}{2}$. This was the case in our second example at the end of Section 1.2.
In the next theorem we quantify the round off errors associated with the computed triangular factors and the solution of the triangular systems.
We use the following conventions, if A and B are in $I\!R^{m\times n}$ then

- $B = |A|$ means $b_{ij} = |a_{ij}|$, $i = 1,\dots,m$, $j = 1,\dots,n.$
- $B \le A$ means $b_{ij} \le a_{ij}$, $i = 1,\dots,m$, $j = 1,\dots,n.$

Theorem 1.2 *Let* $\hat{L}$ *and* $\hat{U}$ *be the computed LU factors of the* $n\times n$ *floating point matrix* A. *Suppose that* $\hat{y}$ *is the computed solution of* $\hat{L}y = b$ *and* $\hat{x}$ *is the computed solution of* $\hat{U}x = \hat{y}$. *Then* $(A+\triangle A)\hat{x} = b$ *with*

$$|\Delta A| \le n(3|A| + 5|\hat{L}||\hat{U}|)u + O(u^2).$$

<u>Proof</u>: see [37], p.107.

The result of this theorem compares favorably with the bound (1.1) except the possibility of a large $|\hat{L}||\hat{U}|$ term. Such a possibility exists because there is nothing in Gaussian elimination to rule out the appearance of small pivots. If a small pivot is encountered then we can expect large numbers in $\hat{L}$ and $\hat{U}$. We stress that small pivots are not necessarily due to ill-conditioning as the example $A = \begin{bmatrix} \epsilon & 1 \\ 1 & 0 \end{bmatrix}$ bears out. Thus Gaussian elimination can give arbitrarily poor results, even for well-conditioned problems. The method may be unstable, depending on the matrix.

1.5 Pivoting and iterative improvement

In order to repair the shortcoming of the Gaussian elimination process, it may be necessary to introduce row and/or column interchanges during the elimination process with the intention of keeping the numbers that arise during the calculation suitably bounded. We start with a definition of a permutation matrix. Suppose that the i^{th} column of the identity matrix I is denoted by e_i. Column permutation matrix P consists of $P = [e_{s_1}, \dots, e_{s_n}]$ where $s_1, \dots, s_n$ is a permutation of the numbers $1, \dots, n$. The matrix AP is a column permuted version of A. A row permuted version of A is given by PA, where $P = \begin{pmatrix} e_{s_1}^T \\ \vdots \\ e_{s_n}^T \end{pmatrix}$.

Partial Pivoting

Before the determination of the Gauss transformation M_k we determine a permutation matrix P_k such that if z is the k-th column of $P_k A^{(k-1)}$, then $|z(k)| = \max\limits_{k \le i \le n} |z(i)|$. The new matrix $A^{(k)}$ is then given by $A^{(k)} = M_k P_k A^{(k-1)}$. Using partial pivoting we can show that no multiplier is greater than one in absolute value.

Partial pivoting only works well if the elements of the matrix are scaled in some way. So if one looks for the solution of $Ax = b$ it is in general a good idea to multiply this equation by a row scaling matrix D^{-1} where D is a diagonal matrix and $D_{ii} = \sum\limits_{j=1}^{n} |a_{ij}|$. After this scaling Gaussian elimination with partial pivoting should be used.

Theorem 1.3 *If Gaussian elimination with partial pivoting is used to compute the upper triangularization*

$$M_{n-1}P_{n-1} \dots M_1 P_1 A = U$$

then

$$PA = LU \text{ where } P = P_{n-1} \dots P_1 \text{ and } L \text{ is a unit lower triangular matrix with } l_{ij} \le 1.$$

Proof: see [37], p. 113.

Using Theorem 1.2 it is easy to show that the computed solution $\hat{x}$ satisfies $(A + \triangle A)\hat{x} = b$ where

$$|\Delta A| \le nu \ (3|A| + 5\hat{P}^T \ |L||U|) + O(u^2)$$

Since the elements of $\hat{L}$ are bounded by one $\|\hat{L}\|_\infty \le n$ which leads to

$$\|\Delta A\|_\infty \le nu \ (3\|A\|_\infty + 5n\|\hat{U}\|_\infty) + O(u^2).$$

The problem now is to bound $\|\hat{U}\|_\infty$. Define the growth factor ρ by

$$\rho = \max_{i,j,k} \frac{|\hat{a}_{ij}^{(k)}|}{\|A\|_\infty}.$$

It follows that $\|\triangle A\|_\infty \le 8n^3 \rho \|A\|_\infty u + O(u^2)$.

The growth factor ρ measures how large the numbers become during the process of elimination. In practice, ρ is usually of order 10 but it can also be as large as 2^{n-1}. Despite this the Gaussian elimination with partial pivoting can be used with confidence. Fortran codes for solving general linear systems based on the ideas give above may be found in the LAPACK package [manual].

Complete pivoting

Before the determination of the Gauss transformation M_k we determine permutation matrices P_k and F_k such that

$$|(P_k A^{(k-1)} F_k)_{kk}| = \max_{\substack{k \le i \le n \\ k \le j \le n}} |(P_k A^{(k-1)} F_k)_{ij}|$$

The matrix $A^{(k)}$ is given by $M_k P_k A^{(k-1)} F_k$.
Complete pivoting has the property that the associated growth factor bound is considerably smaller than 2^{n-1}. However, for complete pivoting one has to look for the maximum value in the $(n-k) \times (n-k)$ lower block of $A^{(k-1)}$. This makes the algorithm approximately two times as expensive than the method without pivoting. This together with the stability in practice for partial pivoting implies that there is no justification for choosing complete pivoting.

It can be shown that for certain classes of matrices it is not necessary to pivot. A matrix $A \in I\!\!R^{n \times n}$ is strictly diagonal dominant if

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \ne i}}^{n} |a_{ij}| \qquad i = 1, ..., n$$

Lemma 4
If A^T is strictly diagonal dominant then A has an LU factorization and $|l_{ij}| \le 1$.

Proof: see [37], p. 120

After the solution is computed, it is possible to improve the computed solution in a cheap way by iterative improvement.

Iterative improvement
Assume t-digit, base β arithmetic, then the computed solution $\hat{x}$ satisfies

$$(A + \Delta A)\hat{x} = b \quad , \quad \|\Delta A\|_\infty \approx u\|A\|_\infty \quad , \quad u = \frac{1}{2}\beta^{1-t}.$$

The residual of a computed solution $\hat{x}$ is the vector $b - A\hat{x}$.

Heuristic 1 Gaussian elimination produces a solution $\hat{x}$ with a relatively small residual $\|b - A\hat{x}\|_\infty \approx u\|A\|_\infty \|\hat{x}\|_\infty$.

Small residuals do not imply high accuracy. Using Theorem 1.1 we see that

$$\frac{\|\hat{x} - x\|_\infty}{\|x\|_\infty} \approx uK_\infty(A).$$

Heuristic 2 If the unit round off and condition satisfy $u \approx 10^{-d}$ and $K_\infty(A) \approx 10^q$, then Gaussian elimination produces a solution $\hat{x}$ that has about $d-q$ correct decimal digits.

Suppose $Ax = b$ has been solved via the partial pivoting factorization $PA = LU$ in t-digit arithmetic. Improvement of the accuracy of the computed solution $\hat{x}$ can be obtained by the following loop:

```
for    $i = 1, \ldots$    until $x$ is accurate enough, do
       Compute  $r = b - Ax$  in $2t -$ digit arithmetic,
       Solve  $Ly = Pr$  for  $y$,
       Solve  $Uz = y$  for  $z$,                                   (2)
       Form  $x := x + z$.
end for
```

Heuristic 3
If the machine precision u and condition number satisfy $u \approx 10^{-d}$ and $K_\infty(A) \approx 10^q$, then after k executions of (2), x has approximately $\min(d, k(d-q))$ correct digits.

If $uK_\infty(A) \leq 1$, then iterative improvement can produce a solution x that has t correct digits. Note that the process is relatively cheap in flops. Each improvement costs $O(n^2)$ flops whereas the original LU decomposition costs $O(n^3)$ flops. Drawbacks are: the implementation is machine dependent, and the memory requirements are doubled because an original copy of A should be stored in memory.

1.6 Cholesky decomposition for symmetric positive definite systems

In many applications the matrix A, used in the linear system $Ax = b$, is symmetric and positive definite. So matrix A satisfies the following rules:

- $A = A^T$,
- $x^T A x > 0 \quad , \quad x \in I\!R^n \quad , \quad x \neq 0.$

For this type of matrices, memory and CPU time can be saved. Since A is symmetric only the elements $a_{ij},\ i = j, \ldots, n \ ; \ j = 1, \ldots, n$ should be stored in memory. Furthermore, the following result can be proved:

Theorem 1.4 *If $A \in I\!R^{n \times n}$ is symmetric positive definite, then there exists a unique lower triangular $G \in I\!R^{n \times n}$ with positive diagonal entries such that $A = GG^T$.*

Proof: see [37], p. 143

The factorization $A = GG^T$ is known as the Cholesky factorization and G is referred to as the Cholesky triangle. Note that if we compute the Cholesky factorization and solve the triangular systems $Gy = b$ and $G^T x = y$, then $b = Gy = GG^T x = Ax$.

Algorithm

Cholesky Decomposition (Column version). Given a symmetric positive definite $A \in I\!R^{n \times n}$, the following algorithm computes a lower triangular $G \in I\!R^{n \times n}$ such that $A = GG^T$. The entry a_{ij} is overwritten by $g_{ij} (i \geq j)$.

```
for    $k = 1, 2, \dots, n$ do
       $a_{kk} := (a_{kk} - \sum_{p=1}^{k-1} a_{kp}^2)^{1/2}$
       for  $i = k+1, \dots, n$ do
            $a_{ik} := (a_{ik} - \sum_{p=1}^{k-1} a_{ip} a_{kp}) / a_{kk}$
       end for
end for
```

The number of flops for this algorithm is equal to $n^3/3$. Note that the amount of memory and work is halved in comparison with Gaussian elimination for a general matrix. The inequality

$$g_{ij}^2 \leq \sum_{p=1}^{i} g_{ip}^2 = a_{ii} \ ,$$

shows that the elements of the Cholesky triangle are bounded, without pivoting. This is again an advantage of this type of matrices.
The round off errors associated with the Cholesky factorization have been studied in [86]. Using the results in this paper, it can be shown that if $\hat{x}$ is the computed solution to $Ax = b$, obtained via Cholesky decomposition then $\hat{x}$ solves the perturbed system $(A+E)\hat{x} = b$ where $\|E\|_2 \leq c_n u \|A\|_2$ and c_n is a small constant depending upon n. Moreover, in [86] it is shown that if $q_n u K_2(A) \leq 1$ where q_n is another small constant, then the Cholesky process runs to completion, i.e., no square roots of negative numbers arise.

1.7 Band matrices

In many applications that involve linear systems, the matrix is banded. This is the case whenever the equations can be ordered so that each unknown x_i appears in only a few equations in a "neighborhood" of the i^{th} equation. The matrix A has upper bandwidth q where $q \geq 0$ is the smallest number such that $a_{ij} = 0$ whenever $j > i + q$ and lower bandwidth p where $p \geq 0$ is the smallest number such that $a_{ij} = 0$ whenever $i > j + p$. Typical examples are obtained after finite element or finite difference discretizations. Substantial reduction of work and memory can be realized for these systems. It can be shown that if A is banded and $A = LU$ then L and U inherits the lower and upper bandwidth of A.

Theorem 1.5 *Suppose $A \in I\!R^{n \times n}$ has an LU factorization $A = LU$. If A has upper bandwidth q and lower bandwidth p, then U has upper bandwidth q and L has lower bandwidth p.*

Proof: see [37], p.152

This result is easily checked by writing down some elimination steps for a banded system of equations.

The LU decomposition can now be obtained using $2npq$ flops if $n \gg p$ and $n \gg q$. The solution of the lower triangular system costs $2np$ flops, $n \gg p$ and the upper triangular system costs $2nq$ flops, $n \gg q$.
Gaussian elimination with partial pivoting can also be specialized to exploit band structure in A. If, however $PA = LU$ then the band properties of L and U are not quite so simple.

Theorem 1.6 *Suppose $A \in I\!R^{n\times n}$ is nonsingular and has upper and lower bandwidths q and p, respectively. If Gaussian elimination with partial pivoting is used to compute Gauss transformations*

$$M_j = I - \alpha^{(j)} e_j^T \qquad j = 1, \ldots, n-1$$

and permutations $P_1, \ldots, P_{n-1}$ such that $M_{n-1}P_{n-1}\ldots M_1P_1A = U$ is upper triangular, then U has upper bandwidth $p+q$ and $\alpha_i^{(j)} = 0$ whenever $i \le j$ or $i > j+p$.

Proof: see [37], p. 154

Thus pivoting destroys the band structure in the sense that U becomes larger than A's upper triangle, while nothing at all can be said about the bandwidth of L. However, since the j^{th} column of L is a permutation of the j^{th} Gauss vector α_j it follows that L has at most $p+1$ nonzero elements per column.
For this reason pivoting is avoided as much as possible in the solution of banded systems. Note that pivoting is not necessary for symmetric positive definite systems, so Cholesky decomposition can be safely applied to banded system without destroying the band structure.

1.8 General sparse matrices

Using a finite difference method the matrix of the linear system can in general adequately be described with a band structure. For the more general finite element method the matrix can be better described using a profile structure. The profile of a matrix can be defined as follows: in the lower triangle all the elements in the row from the first non zero to the main diagonal, and the upper triangle all the elements in the column from the first non zero to the main diagonal belong to the profile. All other elements are lying outside the profile. An example is given in Figure 1.1. In this example the profile of the matrix is symmetric. Only

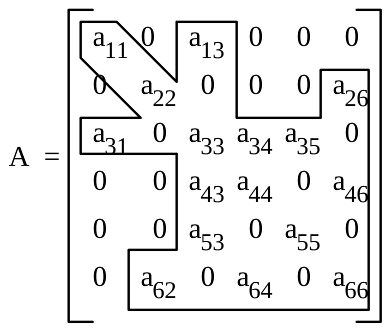


Figure 1: Example of a sparse matrix with its profile structure

the elements of the matrix within its profile are stored in memory. As can be seen from the example this can lead to a large saving, also with respect to a band structure. To give an idea how to store this profile we give an example for the matrix of Figure 1.1. This matrix requires the following arrays:

diag : $[a_{11}, a_{22}, a_{33}, a_{44}, a_{55}, a_{66}]$,

row : $[a_{31}, 0, a_{43}, a_{53}, 0, a_{62}, 0, a_{64}, 0]$,

column : $[a_{13}, 0, a_{34}, a_{35}, 0, a_{26}, 0, a_{46}, 0]$,

position: $[1, 1, 1, 3, 4, 6, 10]$.

The array diag contains all the diagonal elements, the array row contains the lower triangular part row-wise, whereas the array column contains the upper triangular part column-wise. The length of row (column) i is given by position$(i+1)$-position(i). The contents of a nonzero row i starts at row (position(i)) (the same for a column).
It can easily be seen that if $A = LU$, where L and U are constructed by Gaussian elimination without pivoting, $L + U$ has the same profile as A. So the above given storage scheme can also be used to store L and U. Renumbering of the equations and unknowns can be used to minimize the profile (band) of a given matrix. For finite element discretizations we refer to [48] and [34].
If pivoting is necessary, or the non zero elements in the profile of original matrix are not stored, the memory to store L and U can be much larger than the memory used to store A. For this kind of applications we refer to the general purpose programs for sparse LU decomposition: MA28 [23] and [22] and Y12M [88]. A survey of sparse matrix storage schemes is given in [8], Section 4.3.1.

1.9 Exercises

1. Show that if $A \in I\!R^{n \times n}$ has an LU decomposition and is nonsingular, then L and U are unique.

2. Show that $\sup\limits_{x \neq 0} \frac{\|Ax\|_p}{\|x\|_p} = \max\limits_{\|x\|_p=1} \|Ax\|_p$ for $p \geq 1$.

3. Show that $\|A\|_1 = \max\limits_{1 \leq j \leq n} \sum\limits_{i=1}^{m} |a_{ij}| \qquad A \in I\!R^{m \times n}$ (the maximal absolute column sum).

4. Show that $\|A\|_2 = \sqrt{\lambda_{max}(A^T A)}$.

5. Show that for every nonsingular matrix A, partial pivoting leads to an LU decomposition of PA so: $PA = LU$.

6. Show that if $A \in I\!R^{n \times n}$ has an LDM^T decomposition and is nonsingular, then L, D, and M are unique.

7. Suppose $A = LU$ with $L = (l_{ij})$, $U = (u_{ij})$ and $l_{ii} = 1$. Derive an algorithm to compute l_{ij} and u_{ij} by comparing the product LU with A.

8. Suppose that A is symmetric and positive definite. Show that the matrix A_k consisting of the first k rows and columns of A is also symmetric and positive definite.

9. Suppose that A is a symmetric and positive definite tridiagonal matrix. Give an algorithm to compute the LDL^T decomposition, where $l_{ii} = 1$.

10. For a symmetric and positive definite matrix A, we define the numbers $f_i(A)$, $i = 1, \ldots, n$ as follows:
$$f_i(A) = min\{j | a_{ij} \neq 0\}.$$
Show that for the Cholesky decomposition $A = LL^T$ the equality $f_i(L) = f_i(A)$ holds for $i = 1, \ldots, n$.

2 Basic iterative solution methods for linear systems

2.1 Introduction and model problem

Problems coming from discretized partial differential equations lead in general to large sparse systems of equations. Direct solution methods can be impractical if A is large and sparse, because the factors L and U can be dense. This is especially the case for 3D problems. So a considerable amount of memory is required and even the solution of the triangular system costs many floating point operations.

In contrast to the direct methods are the iterative methods. These methods generate a sequence of approximate solutions $\{x^{(k)}\}$ and essentially involve the matrix A only in the context of matrix-vector multiplication. The evaluation of an iterative method invariably focuses on how quickly the iterates $x^{(k)}$ converge. The study of round off errors is in general not very well developed. A reason for this is that the iterates are only approximations of the exact solution, so round off errors in general only influence the speed of convergence but not the quality of the final approximation.

The use of iterative solution methods is very attractive in time dependent and nonlinear problems. For these problems the solution of the linear system is part of an outer loop: time stepping for a time dependent problem and Newton Raphson (or other nonlinear methods) for a nonlinear problem. So good start vectors are in general available: the solution of the preceding outer iteration, whereas the required accuracy is in general low for these problems. Both properties lead to the fact that only a small number of iterations is sufficient to obtain the approximate solution of the linear system. Before starting the description and analysis of iterative methods we describe a typical model problem obtained from a discretized partial differential equation. The properties and definitions of the given methods are illustrated by this problem.

Model problem
Consider the discrete approximation of the Poisson equation

$$\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} = G(x,y) \quad , \quad 0 < x < 1 \quad , \quad 0 < y < 1,$$

and boundary conditions

$$w(x,y) = g(x,y) \quad , \quad x \in \{0,1\} \quad , \quad \text{or} \quad y \in \{0,1\}\,.$$

In the sequel v_{ij} is an approximation of $w(x_i, y_j)$ where $x_i = ih$ and $y_j = jh$, $0 \le i \le m+1$, $0 \le j \le m+1$. The finite difference approximation may be written as:

$$4v_{i,j} - v_{i+1,j} - v_{i-1,j} - v_{i,j+1} - v_{i,j-1} = -h^2\ G(x_i, y_j).$$

The ordering of the nodal points is given in Figure 2 for $m = 3$. The k^{th} component u_k of the vector u is the unknown corresponding to the grid point labeled k. When all the boundary

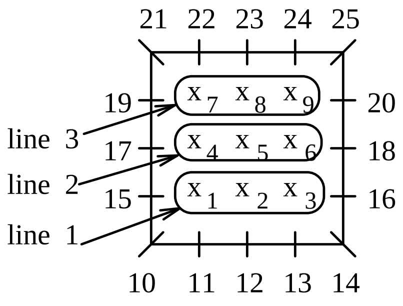


Figure 2: The natural (lexicographic) ordering of the nodal points

terms are moved to the right-hand side, the system of difference equations can be written as:

$$\begin{bmatrix} 4 & -1 & 0 & -1 & & & & & \\ -1 & 4 & -1 & 0 & -1 & & \emptyset & & \\ 0 & -1 & 4 & 0 & 0 & -1 & & & \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & & \\ & -1 & 0 & -1 & 4 & -1 & 0 & -1 & \\ & & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ & & & -1 & 0 & 0 & 4 & -1 & 0 \\ & \emptyset & & & -1 & 0 & -1 & 4 & -1 \\ & & & & & -1 & 0 & -1 & 4 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \\ u_9 \end{bmatrix} = \begin{bmatrix} g_{11} & + & g_{15} & - & h^2G_1 \\ g_{12} & & & - & h^2G_2 \\ g_{13} & + & g_{16} & - & h^2G_3 \\ & & g_{17} & - & h^2G_4 \\ & & & - & h^2G_5 \\ & & g_{18} & - & h^2G_6 \\ g_{19} & + & g_{22} & - & h^2G_7 \\ & & g_{23} & - & h^2G_8 \\ g_{20} & + & g_{24} & - & h^2G_9 \end{bmatrix}$$

If the unknowns on lines parallel to the x-axis are grouped together the given system can be written as:

$$\begin{bmatrix} A_{1,1} & A_{1,2} & 0 \\ A_{2,1} & A_{2,2} & A_{2,3} \\ 0 & A_{3,2} & A_{3,3} \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ F_3 \end{bmatrix},$$

where the unknowns on line k are denoted by U_k. The lines and grid points are given in Figure 2.1. The matrices $A_{i,j}$ are given by:

$$A_{k,k} = \begin{bmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{bmatrix} \quad \text{and} \quad A_{k+1,k} = A_{k,k+1} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}.$$

The submatrix $A_{k,l}$ gives the coupling of the unknowns from line k to those on line l.

2.2 Basic iterative methods

The basic idea behind iterative methods for the solution of a linear system $Ax = b$ is: starting from a given $x^{(k)}$, obtain a better approximation $x^{(k+1)}$ of x in a cheap way. Note that $b - Ax^{(k)}$ is small if $x^{(k)}$ is close to x. This motivates the iteration process

$$x^{(k+1)} = x^{(k)} + M^{-1}(b - Ax^{(k)}) \tag{3}$$

One immediately verifies that if this process converges, x is a possible solution.
So if $\|b - Ax^{(k)}\|_2$ is large we get a large change of $x^{(k)}$ to $x^{(k+1)}$. The choice of M is crucial in order to obtain a fast converging iterative method. Rewriting of (3) leads to:

$$Mx^{(k+1)} = Nx^{(k)} + b \tag{4}$$

where the matrix N is given by $N = M - A$. The formula $A = M - N$ is also known as a splitting of A. It can easily be seen that if $x^{(k+1)} \to x$ the vector x should satisfy

$$Mx = Nx + b \quad \Leftrightarrow \quad Ax = (M - N)x = b.$$

As a first example we describe the point Gauss Jacobi method. First we define $D = \mathit{diag}\ (A)$ and L and U which are the strictly lower respectively the strictly upper part of A. So $A = D + L + U$.

Gauss Jacobi (point).
The choice $M = D$ and $N = -(L + U)$ leads to the point Gauss Jacobi iteration. This algorithm can be described by the following formula:

```
for  $i = 1, \ldots, n$ do
    $x_i^{(k+1)} = \left( b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)} \right) / a_{ii}$        (5)
end for
```

One immediately sees that only memory is required to store the matrix A, the right-hand side vector b and the approximation vector $x^{(k)}$ which can be overwritten in the next step. For our model problem it is sufficient to store 7 vectors in memory. Furthermore, one iteration costs approximately as much work as a matrix vector product.

Whether or not the iterates obtained by formula (4) converge to $x = A^{-1}b$ depends upon the eigenvalues of $M^{-1}N$. The set of eigenvalues of A is denoted as the spectrum of A. The spectral radius of a matrix A is defined by

$$\rho(A) = \max\ \{|\lambda|, \text{ where } \lambda \in \text{ spectrum of } A\}.$$

The size of $\rho(M^{-1}N)$ is critical to the convergence behavior of (4).

Theorem 2.1 *Suppose $b \in I\!R^n$ and $A = M - N \in I\!R^{n \times n}$ is nonsingular. If M is nonsingular and the spectral radius of $M^{-1}N$ is less than 1, then the iterates $x^{(k)}$ defined by (4) converge to $x = A^{-1}b$ for any starting vector $x^{(0)}$.*

Proof: Let $e^{(k)} = x^{(k)} - x$ denote the error in the k^{th} iterate. Since $Mx = Nx + b$ it follows that

$$M(x^{(k+1)} - x) = N(x^{(k)} - x)$$

and thus the error in $x^{(k+1)}$ given by $e^{(k+1)}$ satisfies:

$$e^{(k+1)} = M^{-1}Ne^{(k)} = (M^{-1}N)^k e^{(0)} \tag{6}$$

From [37], p. 336, Lemma 7.3.2 it follows that $(M^{-1}N)^k \to 0$ for $k \to \infty$ if $\rho(M^{-1}N) < 1$, so $e^{(k+1)} \to 0$ for $k \to \infty$. $\square$

As an example we note that point Gauss Jacobi is convergent if the matrix A is strictly diagonal dominant. To show this we note that

$$\rho(M^{-1}N) \leq \|M^{-1}N\|_\infty = \|D^{-1}(L + U)\|_\infty = \max_{1 \leq i \leq n} \sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{|a_{ij}|}{|a_{ii}|}\ .$$

Since a strictly diagonal dominant matrix has the property that $\sum\limits_{\substack{j=1\\j\neq i}}^{n} |a_{ij}| < |a_{ii}|$ it follows that $\rho(M^{-1}N) < 1$.
In many problems an increase of the diagonal dominance leads to a decrease of the number of iterations.

Summarizing the results given above we see that a study of basic iterative methods proceeds along the following lines:

- a splitting $A = M - N$ is given where systems of the form $Mz = d$ (d given and z unknown) are cheaply solvable in order to obtain a practical iteration method by (4),
- classes of matrices are identified for which the iteration matrix $M^{-1}N$ satisfies $\rho(M^{-1}N) < 1$.
- further results about $\rho(M^{-1}N)$ are established to gain intuition about how the error $e^{(k)}$ tends to zero.

For Gauss Jacobi we note that M is a diagonal matrix so that systems of the form $Mz = d$ are easily solvable. We have specified a class of matrices such that convergence occurs. The final point given above is in general used to obtain new methods with a faster rate of convergence or a wider class of matrices such that $\rho(M^{-1}N) < 1$.

Below we first give a block variant of the Gauss Jacobi method. Thereafter other methods are specified. In the remainder of this section we suppose that $Ax = b$ is partitioned in the form

$$\begin{bmatrix} A_{1,1} & \dots & A_{1,q} \\ \vdots & & \vdots \\ A_{q,1} & \dots & A_{q,q} \end{bmatrix} \begin{bmatrix} X_1 \\ \vdots \\ X_q \end{bmatrix} = \begin{bmatrix} B_1 \\ \vdots \\ B_q \end{bmatrix},$$

where $A_{i,j}$ is an $n_i \times n_j$ submatrix and $n_1 + n_2 \ldots + n_q = n$. Here the X_i and B_i represent subvectors of order n_i. Furthermore, we define

$$D = \begin{bmatrix} A_{1,1} & & \oslash \\ & \ddots & \\ \oslash & & A_{q,q} \end{bmatrix}, \quad L = \begin{bmatrix} \bigcirc & & & \\ A_{2,1} & & \oslash & \\ \vdots & \ddots & & \\ A_{q,1} & & A_{q,q-1} & \bigcirc \end{bmatrix} \text{ and } U = \begin{bmatrix} \bigcirc & A_{1,2} & A_{1,q} \\ & \ddots & \\ \oslash & & A_{q-1,q} \\ & & \bigcirc \end{bmatrix}$$

<u>Gauss Jacobi</u> (block)
For the given matrices D, L and U the Gauss Jacobi method is given by $M = D$ and $N = -(L + U)$. The iterates can be obtained by the following algorithm:

```
for $i = 1, \ldots, q$ do
    $X_i^{(k+1)} = A_{i,i}^{-1} \left( B_i - \sum_{j=1, j\neq i}^{q} A_{ij} X_j^{(k)} \right)$.
end for
```

For the special case $n_i = 1$, $i = 1, \ldots, n$ we get the point Gauss Jacobi back. In our example a natural block ordering is obtained if we take $n_i = 3$. In that case the diagonal block matrices $A_{i,i}$ are tridiagonal matrices for which cheap methods exist to solve systems of the form

$A_{i,i}z = d.$

Note that in the Gauss Jacobi iteration one does not use the most recently available information when computing $X_i^{(k+1)}$. For example $X_1^{(k)}$ is used in the calculation of $X_2^{(k+1)}$ even though component $X_1^{(k+1)}$ is already known. If we revise the Gauss Jacobi iteration so that we always use the most current estimate of the exact X_i then we obtain:

Gauss Seidel (block)
The Gauss Seidel method is given by $M = D + L$ and $N = -U$. The iterates can be obtained by

```
for  $i =$  $1, \dots, q$ do
        $X_i^{(k+1)} = A_{i,i}^{-1}\left(B_i - \sum_{j=1}^{i-1} A_{i,j}X_j^{(k+1)} - \sum_{j=i+1}^{q} A_{i,j}X_j^{(k)}\right).$
end for
```

Note that again the solution of systems of the form $Mz = d$ are easily obtained since $M = D + L$ and L is a strictly lower triangular block matrix. As an example for the convergence results that can be proved for the point Gauss Seidel method ($A_{i,i} = a_{i,i}$), we give the following theorem:

Theorem 2.2 *If $A \in \mathbb{R}^{n\times n}$ is symmetric and positive definite, then the point Gauss Seidel iteration converges for any $x^{(0)}$.*

Proof: see [37], p. 512, Theorem 10.1.2.

This result is frequently applicable because many of the matrices that arise from discretized elliptic partial differential equations are symmetric positive definite. In general Gauss Seidel converges faster than Gauss Jacobi. Furthermore, block versions converge faster than point versions. For these results and further detailed convergence proofs we refer to [79], [87], [43], and [8].

The Gauss Seidel iteration is in general a slowly converging process. Inspections of the iterates show that in general the approximations are monotonous increasing or decreasing. Hence we may expect an improvement of the rate of convergence if we use an extrapolation in the direction of the expected change. This leads to the so called successive over-relaxation (SOR) method:

SOR (block)
The successive over-relaxation method is obtained by choosing a constant $\omega \in \mathbb{R}$ and compute the iterates by

$$M_\omega x^{(k+1)} = N_\omega x^{(k)} + \omega b,$$

where $M_\omega = D + \omega L$ and $N_\omega = (1-\omega)D - \omega U$. Note that $\omega A = M_\omega - N_\omega$. The following algorithm can be used:

```
for  $i =$  $1, \dots, q$ do
        $X_i^{(k+1)} = \omega A_{i,i}^{-1}\left(B_i - \sum_{j=1}^{i-1} A_{i,j}X_j^{(k+1)} - \sum_{j=i+1}^{q} A_{i,j}X_j^{(k)}\right) + (1-\omega)X_i^{(k)}.$
end for
```

It can be shown that for $0 < \omega < 2$ the SOR method converges if A is symmetric and positive definite. For $\omega < 1$ we have underrelaxation and for $\omega > 1$ we have overrelaxation. In most examples overrelaxation is used. For a few structured (but important) problems such as our model problem, the value of the relaxation parameter ω that minimizes $\rho(M_\omega^{-1}N_\omega)$ is known. Moreover, a significant reduction of $\rho(M_{\omega_{opt}}^{-1}N_{\omega_{opt}})$ with respect to $\rho(M_1^{-1}N_1)$ can be obtained. Note that for $\omega = 1$ we get the Gauss Seidel method. As an example it appears that for the model problem the number of Gauss Seidel iterations is proportional to $\frac{1}{h^2}$ whereas the number of SOR iterations with optimal ω is proportional to $\frac{1}{h}$. So for small values of h a considerable gain in work results. However, in more complicated problems it may be necessary to perform a fairly sophisticated eigenvalue analysis in order to determine an appropriate ω. A complete survey of "SOR theory" appeared in [87]. Some practical schemes for estimating the optimum ω are discussed in [10], [15], and [84].

Chebyshev

The SOR method is presented as an acceleration of the Gauss Seidel method. Another method to accelerate the convergence of an iterative method is the Chebyshev method. Suppose $x^{(1)}, \ldots, x^{(k)}$ have been obtained via (4), and we wish to determine coefficients $\mu_j(k)$, $j = 0, \ldots, k$ such that

$$y^{(k)} = \sum_{j=0}^{k} \mu_j(k) x^{(j)} \tag{7}$$

is an improvement of $x^{(k)}$. If $x^{(0)} = \ldots = x^{(k)} = x$, then it is reasonable to insist that $y^{(k)} = x$. Hence we require

$$\sum_{j=0}^{k} \mu_j(k) = 1 \tag{8}$$

and consider how to choose the $\mu_j(k)$ so that the error $y^{(k)} - x$ is minimized. It follows from the proof of Theorem 2.1 that $e^{(k+1)} = (M^{-1}N)^k e^{(0)}$ where $e^{(k)} = x^{(k)} - x$. This implies that

$$y^{(k)} - x = \sum_{j=0}^{k} \mu_j(k)(x^{(j)} - x) = \sum_{j=0}^{k} \mu_j(k)(M^{-1}N)^j e^{(0)}. \tag{9}$$

Using the 2-norm we look for $\mu_j(k)$ such that $\|y^{(k)} - x\|_2$ is minimal. To simplify this minimization problem we use the following inequality:

$$\|y^{(k)} - x\|_2 \leq \|p_k(M^{-1}N)\|_2 \|x^{(0)} - x\|_2 \tag{10}$$

where $p_k(z) = \sum\limits_{j=0}^{k} \mu_j(k) z^j$ and $p_k(1) = 1$. We now try to minimize $\|p_k(M^{-1}N)\|_2$ for all polynomials satisfying $p_k(1) = 1$. Another simplification is the assumption that $M^{-1}N$ is symmetric with eigenvalues λ_i that satisfy $\alpha \leq \lambda_n \ldots \leq \lambda_1 \leq \beta < 1$. Using these assumptions we see that

$$\|p_k(M^{-1}N)\|_2 = \max_{\lambda_i} |p_k(\lambda_i)| \leq \max_{\alpha < \lambda < \beta} |p_k(\lambda)|.$$

So to make the norm of $p_k(M^{-1}N)$ small we need a polynomial $p_k(z)$ that is small on $[\alpha, \beta]$ subject to the constraint that $p_k(1) = 1$. This is a minimization problem of polynomials on the real axis. The solution of this problem is obtained by Chebyshev polynomials. These polynomials $c_j(z)$ can be generated by the following recursion

$$\begin{aligned}
&c_0(z) = 1, \\
&c_1(z) = z, \\
&c_j(z) = 2zc_{j-1}(z) - c_{j-2}(z).
\end{aligned}$$

These polynomials satisfy $|c_j(z)| \leq 1$ on $[-1, 1]$ but grow rapidly in magnitude outside this interval. As a consequence the polynomial

$$p_k(z) = \frac{c_k\left(-1 + 2\frac{z-\alpha}{\beta-\alpha}\right)}{c_k\left(1 + 2\frac{1-\beta}{\beta-\alpha}\right)}$$

satisfies $p_k(1) = 1$, since $-1 + 2\frac{1-\alpha}{\beta-\alpha} = 1 + 2\frac{1-\beta}{\beta-\alpha}$, and tends to be small on $[\alpha, \beta]$. The last property can be explained by the fact that

$$-1 \leq -1 + 2\frac{z-\alpha}{\beta-\alpha} \leq 1 \qquad \text{for } \ z \in [\alpha, \beta] \ \text{ so the}$$

numerator is less than 1 in absolute value, whereas the denominator is large in absolute value since $1 + 2\frac{1-\beta}{\beta-\alpha} > 1$. This polynomial combined with (10) leads to

$$\|y^{(k)} - x\|_2 \leq \frac{\|x - x^{(0)}\|_2}{|c_k\left(1 + 2\frac{1-\beta}{\beta-\alpha}\right)|}. \tag{11}$$

Calculation of the approximation $y^{(k)}$ by formula (7) costs much time and memory, since all the vectors $x^{(0)}, \ldots, x^{(k)}$ should be kept in memory. Furthermore, to calculate $y^{(k)}$ one needs to add $k+1$ vectors, which for the model problem costs for $k \geq 5$ more work than one matrix vector product. Using the recursion of the Chebyshev polynomials it is possible to derive a three term recurrence among the $y^{(k)}$. It can be shown that the vectors $y^{(k)}$ can be calculated as follows:

$$y^{(0)} = x^{(0)}$$

solve $z^{(0)}$ from $Mz^{(0)} = b - Ay^{(0)}$ then $y^{(1)}$ is given by

$$y^{(1)} = y^{(0)} + \tfrac{2}{2-\alpha-\beta} z^{(0)}$$

solve $z^{(k)}$ from $Mz^{(k)} = b - Ay^{(k)}$ then $y^{(k+1)}$ is given by

$$y^{(k+1)} = \frac{4 - 2\beta - 2\alpha}{\beta - \alpha} \frac{c_k\left(1 + 2\frac{1-\beta}{\beta-\alpha}\right)}{c_{k+1}\left(1 + 2\frac{1-\beta}{\beta-\alpha}\right)} \left(y^{(k)} - y^{(k-1)} + \frac{2}{2-\alpha-\beta} z^{(k)}\right) + y^{(k-1)}\ .$$

We refer to this scheme as the Chebyshev semi-iterative method associated with $My^{(k+1)} = Ny^{(k)} + b$. Note that only 4 vectors are needed in memory and the extra work consists of the addition of 4 vectors. In order that the acceleration is effective it is necessary to have good lower and upper bounds of α and β. These parameters may be difficult to obtain. Chebyshev semi-iterative methods are extensively analyzed in [79], [38] and [43].

In deriving the Chebyshev acceleration we assumed that the iteration matrix $M^{-1}N$ was symmetric. Thus our simple analysis does not apply to the SOR iteration matrix $M_\omega^{-1}N_\omega$ because this matrix is not symmetric. To repair this Symmetric SOR (SSOR) is proposed. In SSOR one SOR step is followed by a backward SOR step. In this backward step the unknowns are updated in reversed order. For further details see [37], Section 10.1.5.
Finally we present some theoretical results for the Chebyshev method [1]. Suppose that the matrix $M^{-1}A$ is symmetric and positive definite and that the eigenvalues μ_i are ordered as follows $0 < \mu_1 \leq \mu_2 \ldots \leq \mu_n$. It is then possible to prove the following theorem:

Theorem 2.3 *If the Chebyshev method is applied and $M^{-1}A$ is symmetric positive definite then*

$$\|y^{(k)} - x\|_2 \leq 2 \left(\frac{\sqrt{K_2(M^{-1}A)} - 1}{\sqrt{K_2(M^{-1}A)} + 1} \right)^k \|x^{(0)} - x\|_2.$$

<u>Proof</u> Since $M^{-1}A = M^{-1}(M-N) = I - M^{-1}N$ we see that the eigenvalues satisfy the following relation:

$$\mu_i = 1 - \lambda_i \quad \text{or} \quad \lambda_i = 1 - \mu_i.$$

This combined with (11) leads to the inequality:

$$\|y^{(k)} - x\|_2 \leq \frac{\|x - x^{(0)}\|_2}{|c_k\left(1 + 2\frac{(1-(1-\mu_1))}{(1-\mu_1)-(1-\mu_n)}\right)|}. \tag{12}$$

So it remains to estimate the denominator. Note that

$$c_k\left(1 + \frac{2(1-(1-\mu_1))}{(1-\mu_1)-(1-\mu_n)}\right) = c_k\left(\frac{\mu_n + \mu_1}{\mu_n - \mu_1}\right) = c_k\left(\frac{1+\frac{\mu_1}{\mu_n}}{1-\frac{\mu_1}{\mu_n}}\right).$$

The Chebyshev polynomial can also be given by

$$c_k(z) = \frac{1}{2}\left\{\left(z + \sqrt{z^2-1}\right)^k + \left(z - \sqrt{z^2-1}\right)^k\right\} \quad \text{[4], p. 180.}$$

This expression can be used to show that

$$\begin{aligned} c_k\left(\frac{1+\frac{\mu_1}{\mu_n}}{1-\frac{\mu_1}{\mu_n}}\right) &> \frac{1}{2}\left(\frac{1+\frac{\mu_1}{\mu_n}}{1-\frac{\mu_1}{\mu_n}} + \sqrt{\left(\frac{1+\frac{\mu_1}{\mu_n}}{1-\frac{\mu_1}{\mu_n}}\right)^2 - 1}\right)^k = \\ &= \frac{1}{2}\left(\frac{1+\frac{\mu_1}{\mu_n}+2\sqrt{\frac{\mu_1}{\mu_n}}}{1-\frac{\mu_1}{\mu_n}}\right)^k = \frac{1}{2}\left(\frac{1+\sqrt{\frac{\mu_1}{\mu_n}}}{1-\sqrt{\frac{\mu_1}{\mu_n}}}\right)^k. \end{aligned} \tag{13}$$

The condition number $K_2(M^{-1}A)$ is equal to $\frac{\mu_n}{\mu_1}$. Together with (12) and (13) this leads to

$$\|y^{(k)} - x\|_2 \leq 2 \left(\frac{\sqrt{K_2(M^{-1}A)} - 1}{\sqrt{K_2(M^{-1}A)} + 1} \right)^k \|x^{(0)} - x\|_2.$$

$\square$

Chebyshev type methods which are applicable to a wider range of matrices are given in the literature. In [52] a Chebyshev method is given for matrices with the property that their eigenvalues are contained in an ellipse in the complex plane, and the origin is no element of this ellipse. For a general theory of semi-iterative methods of Chebyshev type we refer to [24].

[1]These results are used in the following chapters to analyze the converge behavior of other iterative methods

2.3 Starting vectors and termination criteria

Starting vectors

All the given iterative solution methods used to solve $Ax = b$ start with a given vector $x^{(0)}$. In this subsection we shall give some ideas how to choose a good starting vector $x^{(0)}$. These choices depend on the problem to be solved. If no further information is available one always starts with $x^{(0)} = 0$. The solution of a nonlinear problem is in general approximated by the solution of a number of linear systems. In such a problem the final solution of the iterative method at a given outer iteration can be used as a starting solution for the iterative method used to solve the next linear system.

Suppose we have a time dependent problem. The solution of the continuous problem is denoted by $u^{(n)}$. In every time step this solution is approximated by a discrete solution $u_h^{(n)}$ satisfying the following linear system

$$A^{(n)} u_h^{(n)} = b^{(n)}.$$

These systems are approximately solved by an iterative method where the iterates are denoted by $x^{(n,k)}$. An obvious choice for the starting vector is $x^{(n+1,0)} = x^{(n,\boldsymbol{k}_n)}$ where $\boldsymbol{k}_n$ denotes the number of iterations in the n^{th} time step. A better initial estimate can be obtained by the following extrapolation:

$$u^{(n+1)} \cong u^{(n)} + \triangle t \frac{du^{(n)}}{dt},$$

where $\frac{du^{(n)}}{dt}$ is approximated by $\frac{x^{(n,\boldsymbol{k}_n)} - x^{(n-1,\boldsymbol{k}_{n-1})}}{\triangle t}$. This leads to the following starting vector

$$x^{(n+1,0)} = 2x^{(n,\boldsymbol{k}_n)} - x^{(n-1,\boldsymbol{k}_{n-1})}\ .$$

Finally starting vectors can sometimes be obtained by solution of related problems, e.g., analytic solution of a simplified problem, a solution computed by a coarser grid, a numerical solution obtained by a small change in one of the parameters etc.

Termination criteria

In Subsection 2.2 we have specified iterative methods to solve $Ax = b$. However, no criteria to stop the iterative process have been given. In general, the iterative method should be stopped if the approximate solution is accurate enough. A good termination criterion is very important for an iterative method, because if the criterion is too weak the approximate solution is useless, whereas if the criterion is too severe the iterative solution method never stops or costs too much work.

We start by giving a termination criterion for a linear convergent process. An iterative method is linear convergent if the iterates satisfy the following equation:

$$\|x^{(k)} - x^{(k-1)}\|_2 \approx r\|x^{(k-1)} - x^{(k-2)}\|_2, \quad r < 1 \tag{14}$$

and $x^{(k)} \to A^{-1}b$ for $k \to \infty$. Relation (14) is easily checked during the computation. In general initially (14) is not satisfied but after some iterations the quantity $\frac{\|x^{(k)} - x^{(k-1)}\|_2}{\|x^{(k-1)} - x^{(k-2)}\|_2}$ converges to r. The Gauss Jacobi, Gauss Seidel and SOR method all are linear convergent.

Theorem 2.4 *For a linear convergent process we have the following inequality*

$$\|x - x^{(i)}\|_2 \le \frac{r}{1-r}\|x^{(i)} - x^{(i-1)}\|_2.$$

Proof
Using (14) we obtain the following inequality for $k \ge i+1$.

$$\|x^{(k)} - x^{(i)}\|_2 \le \sum_{j=i}^{k-1} \|x^{(j+1)} - x^{(j)}\|_2 \le \sum_{j=1}^{k-i} r^j \|x^{(i)} - x^{(i-1)}\|_2$$

$$= r\tfrac{1-r^{k-i-1}}{1-r}\|x^{(i)} - x^{(i-1)}\|_2 \ .$$

Since $\lim\limits_{k\to\infty} x^{(k)} = x$ this implies that

$$\|x - x^{(i)}\|_2 \le \frac{r}{1-r}\|x^{(i)} - x^{(i-1)}\|_2.$$

□

The result of this theorem can be used to give a stopping criterion for linear convergent methods. Sometimes the iterations are stopped if $\|x^{(i)} - x^{(i-1)}\|_2$ is small enough. If r is close to one this may lead to inaccurate results since $\frac{r}{1-r}\|x^{(i)} - x^{(i-1)}\|_2$ and thus $\|x - x^{(i)}\|_2$ may be large. A safe stopping criterion is: stop if

$$\frac{r}{1-r}\frac{\|x^{(i)} - x^{(i-1)}\|_2}{\|x^{(i)}\|_2} \le \epsilon.$$

If this condition holds then the relative error is less than ε:

$$\frac{\|x - x^{(i)}\|_2}{\|x\|_2} \cong \frac{\|x - x^{(i)}\|_2}{\|x^{(i)}\|_2} \le \frac{r}{1-r}\frac{\|x^{(i)} - x^{(i-1)}\|_2}{\|x^{(i)}\|_2} \le \epsilon.$$

Furthermore, Theorem 2.4 yields the following result:

$$\|x - x^{(i)}\|_2 \le \frac{r^i}{1-r}\|x^{(1)} - x^{(0)}\|_2 \ . \tag{15}$$

So assuming that the expression (15) can be replaced by an equality

$$\log\ \|x - x^{(i)}\|_2 \ = \ i \log\ (r) + \log\ \left(\frac{\|x^{(1)} - x^{(0)}\|_2}{1-r}\right) \ . \tag{16}$$

This implies that the curve log $\|x - x^{(i)}\|_2$ is a straight line as function of i. This was the motivation for the term linear convergent process. Given the quantity r, which is also known as the rate of convergence, or reduction factor, the required accuracy and $\|x^{(1)} - x^{(0)}\|_2$ it is possible to estimate the number of iterations to achieve this accuracy. In general r may be close to one and hence a small increase of r may lead to a large increase of the required number of iterations.

For iterative methods, which have another convergence behavior most stopping criteria are based on the norm of the residual. Below we shall give some of these criteria and give comment on their properties.

Criterion 1 $\|b - Ax^{(i)}\|_2 \le \epsilon$.

The main disadvantage of this criterion is that it is not scaling invariant. This implies that if $\|b - Ax^{(i)}\|_2 < \epsilon$ this does not hold for $\|100(b - Ax^{(i)})\|_2$. Although the accuracy of $x^{(i)}$ remains the same. So a correct choice of ϵ depends on properties of the matrix A.

The remaining criteria are all scaling invariant.

Criterion 2 $\frac{\|b-Ax^{(i)}\|_2}{\|b-Ax^{(0)}\|_2} \le \epsilon$

The number of iterations is independent of the initial estimate $x^{(0)}$. This may be a drawback since a better initial estimate does not lead to a decrease of the number of iterations.

Criterion 3 $\frac{\|b-Ax^{(i)}\|_2}{\|b\|_2} \le \epsilon$

This is a good termination criterion. The norm of the residual is small with respect to the norm of the right-hand side. Replacing ϵ by $\epsilon/K_2(A)$ we can show that the relative error in x is less than ϵ. It follows (compare Theorem 1.1) that:

$$\frac{\|x - x^{(i)}\|_2}{\|x\|_2} \le K_2(A)\frac{\|b - Ax^{(i)}\|_2}{\|b\|_2} \le \epsilon.$$

In general $\|A\|_2$ and $\|A^{-1}\|_2$ are not known. Some iterative methods gives approximations of these quantities.

Criterion 4 $\frac{\|b-Ax^{(i)}\|_2}{\|x^{(i)}\|_2} \le \epsilon/\|A^{-1}\|_2$

This criterion is closely related to Criterion 3. In many cases this criterion also implies that the relative error is less then ϵ:

$$\frac{\|x - x^{(i)}\|_2}{\|x\|_2} \cong \frac{\|x - x^{(i)}\|_2}{\|x^{(i)}\|_2} = \frac{\|A^{-1}(b - Ax^{(i)})\|_2}{\|x^{(i)}\|_2} \le \frac{\|A^{-1}\|_2\|b - Ax^{(i)}\|_2}{\|x^{(i)}\|_2} \le \epsilon$$

Sometimes physical relations lead to other termination criteria. This is the case if the residual has a physical meaning, for instance the residual is equal to some energy or the deviation from a divergence free vector field etc.

In Theorem 1.1 we have seen that due to rounding errors the solution x represented on a computer has a residual which may be of the magnitude of $u\|b\|_2$, where u is the machine precision. So we cannot expect that a computed solution by an iterative solution method has a smaller norm of the residual. In a good implementation of an iterative method, a warning is given if the required accuracy is too high. If for instance the termination criterion is $\|b-Ax^{(i)}\|_2 \le \epsilon$ and ϵ is chosen less then $1000u\|b\|_2$ a warning should be given and ε should be replaced by $\epsilon = 1000u\|b\|_2$. The arbitrary constant 1000 is used for safety reasons.

2.4 Exercises

1. Suppose $\|E\| < 1$ for some matrix $E \in I\!R^{n \times n}$. Show that

$$(I - E)^{-1} = \sum_{k=0}^{\infty} E^k \text{ and } \|(I - E)^{-1}\| \leq \frac{1}{1 - \|E\|}.$$

2. Show that if A is strictly diagonal dominant then the Gauss Seidel method converges.

3. Suppose that A is symmetric and positive definite.

   (a) Show that one can write $A = D - L - L^T$ where D is diagonal with $d_{ii} > 0$ for each $1 \leq i \leq n$ and L is strictly lower triangular. Further show that $D - L$ is nonsingular.

   (b) Let $T_g = (D - L)^{-1} L^T$ and $P = A - T_g^T A T_g$. Show that P is symmetric.

   (c) Show that T_g can also be written as $T_g = I - (D - L)^{-1} A$.

   (d) Let $Q = (D - L)^{-1} A$. Show that $T_g = I - Q$ and

$$P = Q^T (A Q^{-1} - A + (Q^T)^{-1} A) Q.$$

   (e) Show that $P = Q^T D Q$ and P is symmetric and positive definite.

   (f) Let λ be an eigenvalue of T_g with eigenvector x. Use part (b) to show that $x^T P x > 0$ implies that $|\lambda| < 1$.

   (g) Show that the Gauss Seidel method converges.

4. Extend the method of proof in Exercise 3 to the SOR method with $0 < \omega < 2$.

5. Suppose that $\tilde{\mu}_1$ is an estimate for μ_1 and $\tilde{\mu}_n$ for μ_n.

   (a) Show that in general the Chebyshev method converges slower if $0 < \tilde{\mu}_1 < \mu_1$ and $\tilde{\mu}_n > \mu_n$ if $\tilde{\mu}_1$ and $\tilde{\mu}_n$ are used in the Chebyshev method.

   (b) Show that divergence can occur if $\tilde{\mu}_n < \mu_n$.

6. (a) Do two iterations with Gauss Jacobi to the system:

$$\begin{pmatrix} 2 & 0 \\ -2 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$

   Note that the second iterate is equal to the exact solution.

   (b) Is the following claim correct?

   The Gauss Jacobi method converges in mostly n iterations if A is a lower triangular matrix

3 A Krylov subspace method for systems with a symmetric positive definite matrix

3.1 Introduction

In the basic iterative solution methods we compute the iterates by the following recursion:

$$x_{i+1} = x_i + M^{-1}(b - Ax_i) = x_i + M^{-1}r_i$$

Writing out the first steps of such a process we obtain:

$$\begin{array}{rcl} x_0 & & , \\ x_1 & = & x_0 + (M^{-1}r_0), \\ x_2 & = & x_1 + (M^{-1}r_1) = x_0 + M^{-1}r_0 + M^{-1}(b - Ax_0 - AM^{-1}r_0) \\ & = & x_0 + 2M^{-1}r_0 - M^{-1}AM^{-1}r_0, \\ & \vdots & \end{array}$$

This implies that

$$x_i \in x_0 + \text{ span}\left\{M^{-1}r_0, M^{-1}A(M^{-1}r_0), \ldots, (M^{-1}A)^{i-1}(M^{-1}r_0)\right\}.$$

The subspace $K^i(A; r_0) := \text{span}\left\{r_0, Ar_0, \ldots, A^{i-1}r_0\right\}$ is called the Krylov-space of dimension i corresponding to matrix A and initial residual r_0. An x_i calculated by a basic iterative method is an element of $x_0 + K^i(M^{-1}A; M^{-1}r_0)$.

In the preceding chapter we tried to accelerate convergence by the Chebyshev method. In this method one approximates the solution x by a vector $x_i \in x_0 + K^i(M^{-1}A; M^{-1}r_0)$ such that $\|x - x_i\|_2$ is minimized in a certain way. One of the drawbacks of that method is that information on the eigenvalues of $M^{-1}A$ should be known. In this chapter we shall describe the Conjugate Gradient method. This method minimizes the error $x - x_i$ in an adapted norm, without having to know any information about the eigenvalues. In Section 3.3 we give theoretical results concerning the convergence behavior of the CG method.

3.2 The Conjugate Gradient (CG) method

In this section we assume that $M = I$, and $x_0 = 0$ so $r_0 = b$. These assumptions are only needed to facilitate the formula's. They are not necessary for the CG method itself. Furthermore, we assume that A satisfies the following condition.

Condition 3.2.1
The matrix A is symmetric ($A = A^T$) and positive definite ($x^TAx > 0$ for $x \neq 0$).

This condition is crucial for the derivation and success of the CG method. Later on we shall derive extensions to non-symmetric matrices.

The first idea could be to construct a vector $x_i \in K^i(A, r_0)$ such that $\|x - x_i\|_2$ is minimal. The first iterate x_1 can be written as $x_1 = \alpha_0 r_0$ where α_0 is a constant which has to be chosen such that $\|x - x_1\|_2$ is minimal. This leads to

$$\|x - x_1\|_2^2 = (x - \alpha_0 r_0)^T (x - \alpha_0 r_0) = x^T x - 2\alpha_0 r_0^T x + \alpha_0^2 r_0^T r_0 \ . \tag{17}$$

The norm given in (17) is minimized if $\alpha_0 = \frac{r_0^T x}{r_0^T r_0}$. Since x is unknown this choice cannot be determined, so this idea does not lead to a useful method. Note that $Ax = b$ is known so using an adapted inner product implying A could lead to an α_0 which is easy to calculate. To follow this idea we define the following inner product and related norm.

<u>Definition 3.2.2</u>
The A-inner product is defined by

$$(y, z)_A = y^T A z,$$

and the A-norm by $\|y\|_A = \sqrt{(y, y)_A} = \sqrt{y^T A y}$.

It is easy to show that if A satisfies Condition 3.2.1 $(.,.)_A$ and $\|.\|_A$ satisfy the rules for inner product and norm (see Section 1.3) respectively. In order to obtain x_1 such that $\|x - x_1\|_A$ is minimal we note that

$$\|x - x_1\|_A^2 = x^T A x - 2\alpha_0 r_0^T A x + \alpha_0^2 \ r_0^T A r_0,$$

so $\alpha_0 = \frac{r_0^T A x}{r_0^T A r_0} = \frac{r_0^T b}{r_0^T A r_0}$. We see that this new inner product leads to a minimization problem, which can be easily solved. In the next iterations we compute x_i such that

$$\|x - x_i\|_A = \min_{y \in K^i(A; r_0)} \|x - y\|_A \tag{18}$$

The solution of this minimization problem leads to the conjugate gradient method. First we specify the CG method, thereafter we summarize some of its properties.

<u>Conjugate Gradient method</u>

```
$k = 0$  ;   $x_0 = 0$ ; $r_0 = b$                                        initialization
while        $r_k \neq 0$ do                                               termination criterion
             $k := k + 1$                                                  $k$ is the iteration number
             if $k = 1$ do
                 $p_1 = r_0$
             else
                 $\beta_k = \frac{r_{k-1}^T r_{k-1}}{r_{k-2}^T r_{k-2}}$     $p_k$ is the search direction vector
                 $p_k = r_{k-1} + \beta_k p_{k-1}$                           to update $x_{k-1}$ to $x_k$
             end if
             $\alpha_k = \frac{r_{k-1}^T r_{k-1}}{p_k^T A p_k}$
             $x_k = x_{k-1} + \alpha_k p_k$                                  update iterate
             $r_k = r_{k-1} - \alpha_k A p_k$                                update residual
end while
```

The first description of this algorithm is given in [44]. For recent results see [76]. Besides the two vectors x_k, r_k and matrix A only one extra vector p_k should be stored in memory. Note that the vectors from the previous iteration can be overwritten. One iteration of CG costs one matrix vector product and 10 n flops for vector calculations. If the CG algorithm is used in a practical application the termination criterion should be replaced by one of the criteria given in Section 2.3. In this algorithm r_k is computed from r_{k-1} by the equation $r_k = r_{k-1} - \alpha_k A p_k$. This is done in order to save one matrix vector product for the original calculation $r_k = b - Ax_k$. In some applications the updated residual obtained from the CG algorithm can deviate much from the exact residual $b - Ax_k$ due to rounding errors. So it is strongly recommended to recompute $b - Ax_k$ after the termination criterion is satisfied for the updated residual and compare the norm of the exact and updated residual. If the exact residual does no satisfy the termination criterion the CG method should be restarted with x_k as its starting vector.

The vectors defined in the CG method have the following properties:

Theorem 3.1

$$1. \qquad \textit{span}\ \{p_1, \ldots, p_k\} = \textit{span}\,\{r_0, \ldots, r_{k-1}\} = K^k(A; r_0), \tag{19}$$

$$2. \qquad r_j^T r_i = 0 \quad i = 0, \ldots, j-1 \quad ; \quad j = 1, \ldots, k \quad , \tag{20}$$

$$3. \qquad r_j^T p_i = 0 \quad i = 1, \ldots, j \quad ; \quad j = 1, \ldots, k \quad , \tag{21}$$

$$4. \qquad p_j^T A p_i = 0 \quad i = 1, \ldots, j-1 \quad ; \quad j = 2, \ldots, k \tag{22}$$

$$5. \qquad \|x - x_k\|_A = \min_{y \in K^k(A; r_0)} \|x - y\|_A . \tag{23}$$

Proof: see [37], Section 10.2.

Remarks on the properties given in Theorem 3.1

- It follows from (19) and (20) that the vectors $r_0, \ldots, r_{k-1}$ form an orthogonal basis of $K^k(A; r_0)$.
- In theory the CG method is a finite method. After n iterations the Krylov subspace is identical to $\mathbb{R}^n$. Since $\|x - y\|_A$ is minimized over $K^n(A; r_0) = \mathbb{R}^n$ the norm is equal to zero and $x_n = x$. However in practice this property is never utilized for two reasons: firstly in many applications n is very large so that it is not feasible to do n iterations, secondly even if n is small, rounding errors can spoil the results such that the properties given in Theorem 3.1 do not hold for the computed vectors.
- The sequence $\|x - x_k\|_A$ is monotone decreasing, so

  $$\|x - x_{k+1}\|_A \leq \|x - x_k\|_A .$$

  This follows from (23) and the fact that $K^k(A; r_0) \subset K^{k+1}(A; r_0)$. In practice $\|x - x_k\|_A$ is not easy to compute since x is unknown. The norm of the residual is given by $\|r_k\|_2 = \|x - x_k\|_{A^T A}$. This sequence is not necessarily monotone decreasing. In applications it may occur that $\|r_{k+1}\|_2$ is larger than $\|r_k\|_2$. This does not mean that the CG process becomes divergent. The inequality

$$\|r_k\|_2 = \|Ax_k - b\|_2 \le \sqrt{\|A\|_2}\|x - x_k\|_A$$

shows that $\|r_k\|_2$ is less than the monotone decreasing sequence $\sqrt{\|A\|_2}\|x - x_k\|_A$, so after some iterations the norm of the residual decreases again.

- The direction vector p_j is A-orthogonal or A-conjugate to all p_i with index i less than j. This is the motivation for the name of the method: the directions or gradients of the updates are mutually conjugate.

- In the algorithm we see two ratios, one to calculate β_k and the other one for α_k. If the denominator is equal to zero, the CG method breaks down. With respect to β_k this implies that $r_{k-2}^T r_{k-2} = 0$, which implies $r_{k-2} = 0$ and thus $x_{k-2} = x$. The linear system is solved. The denominator of α_k is zero if $p_k^T A p_k = 0$ so $p_k = 0$. Using property (19) this implies that $r_{k-1} = 0$ so again the problem is already solved.
  Conclusion: If the matrix A satisfies Condition 3.2.1 then the CG method is robust.

In the following chapter we shall give CG type methods for general matrices A. But first we shall extend Condition 3.2.1 in such a way that also singular matrices are permitted. If the matrix A is symmetric and positive semi definite ($x^T Ax \ge 0$) the CG method can be used to solve the linear system $Ax = b$, provided b is an element of the column space of A (range(A)). This is a natural condition because if it does not hold there is no vector x such that $Ax = b$. For further details and references see [47].

3.3 The convergence behavior of the CG method

An important research topic is the rate of convergence of the CG method. The optimality property enables one to obtain easy to calculate upper bounds of the distance between the k^{th} iterate and the exact solution.

Theorem 3.2 *The iterates x_k obtained from the CG algorithm satisfy the following inequality:*

$$\|x - x_k\|_A \le 2\left(\frac{\sqrt{K_2(A)} - 1}{\sqrt{K_2(A)} + 1}\right)^k \|x - x_0\|_A.$$

Proof
We shall only give a sketch of the proof. It is easily seen that $x - x_k$ can be written as a polynomial, say $p_k(A)$ with $p_k(0) = 1$, times the initial residual (compare the Chebyshev method)

$$\|x - x_k\|_A = \|p_k(A)(x - x_0)\|_A.$$

Due to the minimization property every other polynomial $q_k(A)$ with $q_k(0) = 1$ does not decrease the error measured in the A-norm:

$$\|x - x_k\|_A \le \|q_k(A)(x - x_0)\|_A.$$

The right-hand side can be written as

$$\|q_k(A)(x - x_0)\|_A = \|q_k(A)\sqrt{A}(x - x_0)\|_2 \le \|q_k(A)\|_2\|\sqrt{A}(x - x_0)\|_2 = \|q_k(A)\|_2\|x - x_0\|_A$$

Taking $q_k(A)$ equal to the Chebyshev polynomial gives the desired result. $\square$

Note that the rate of convergence of CG is comparable to that of the Chebyshev method, however it is not necessary to estimate or calculate eigenvalues of the matrix A. Furthermore, increasing diagonal dominance leads to a better rate of convergence.

Initially the CG method was not very popular. The reason for this is that the convergence can be slow for systems where the condition number $K_2(A)$ is very large. On the other hand the fact that the solution is found after n iteration is also not useful in practice. Firstly n may be very large, secondly the property does not hold in the presence of rounding errors. To illustrate this we consider the following classical example:

<u>Example 1</u>
The linear system $Ax = b$ should be solved where $n = 40$ and $b = (1, 0, \ldots, 0)^T$. The matrix A is given by

$$A = \begin{bmatrix} 5 & -4 & 1 & & & & & & \\ -4 & 6 & -4 & 1 & & & & \emptyset & \\ 1 & -4 & 6 & -4 & 1 & & & & \\ & \ddots & \ddots & \ddots & \ddots & & & & \\ & & \ddots & \ddots & \ddots & \ddots & & & \\ & & & & 1 & -4 & 6 & -4 & 1 \\ & \emptyset & & & & 1 & -4 & 6 & -4 \\ & & & & & & 1 & -4 & 5 \end{bmatrix}.$$

This can be seen as a finite difference discretization of the bending beam equation:
$u'''' = f$. The eigenvalues of this matrix are given by:

$$\lambda_k = 16 \sin^4 \frac{k\pi}{82} \quad k = 1, \ldots, 40.$$

The matrix A is symmetric positive definite so the CG method can be used to solve the linear system. The condition number of A is approximately equal to $\left(\frac{82}{\pi}\right)^4$. The resulting rate of convergence given by

$$\frac{\sqrt{K_2(A)} - 1}{\sqrt{K_2(A)} + 1} \cong 0.997$$

is close to one. This explains a slow convergence of the CG method for the first iterations. However after 40 iterations the solution should be found. In Figure 3 the convergence behavior is given where the rounding error is equal to 10^{-16}, [35]. This example suggests that CG has only a restricted range of applicability. These ideas however changed after the publication of [63]. Herein it is shown that the CG method can be very useful for a class of linear systems, not as a direct method, but as an iterative method. These problems originate from discretized partial differential equations. It appears that not the size of the matrix is important for convergence but the extreme eigenvalues of A.

One of the results which is based on the extreme eigenvalues is given in Theorem 3.2. This inequality is an upper bound for the error of the CG iterates, and suggests that the CG method is a linearly convergent process (see Figure 4). However, in practice the convergence behavior looks like the one given in Figure 5. This is called superlinear convergence behavior.

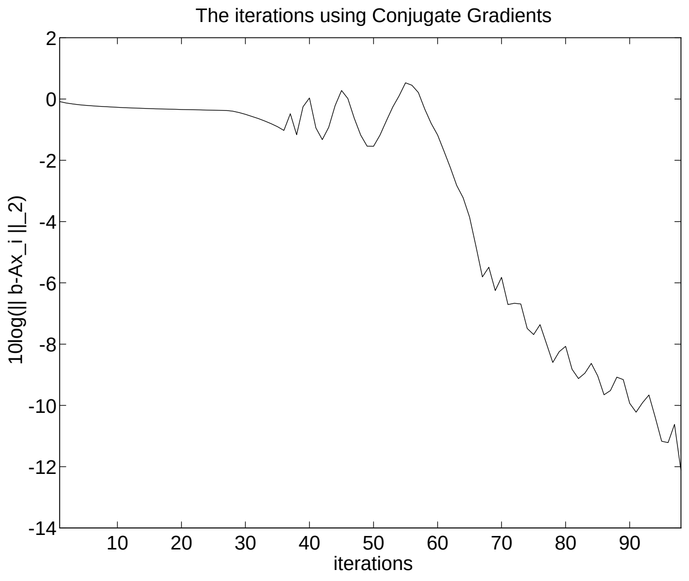


Figure 3: The convergence behavior of CG applied to Example 1.

So the upper bound is only sharp for the initial iterates. It seems that after some iterations the condition number in Theorem 3.2 is replaced by a smaller "effective" condition number. To illustrate this we give the following example:

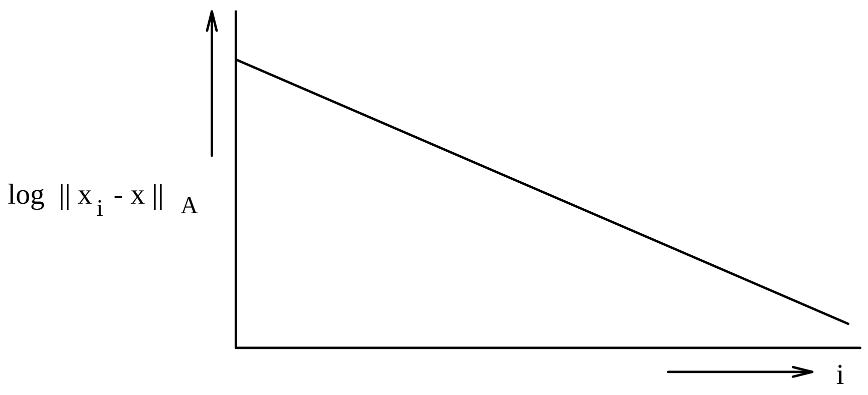


Figure 4: A linear convergent behavior

Example 2

The matrix A is the discretized Poisson operator. The physical domain is the two-dimensional unit square. The grid used consists of an equidistant distribution of 30×30 grid points. The dimension of A is equal to 900 and the eigenvalues are given by

$$\lambda_{k,l} = 4 - 2\cos\frac{\pi k}{31} - 2\cos\frac{\pi l}{31} \quad , \quad 1 \le k, l \le 30.$$

Using Theorem 3.2 it appears that 280 iteration are necessary to ensure that

$$\frac{\|x - x_i\|_A}{\|x - x_0\|_A} \le 10^{-12}.$$

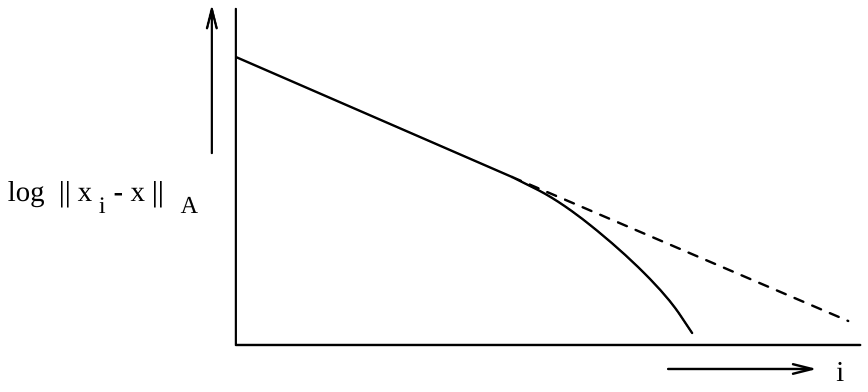


Figure 5: A super linear convergent behavior

Computing the solution it appears that CG iterates satisfy the given termination criterion after 120 iterations. So in this example the estimate given in Theorem 3.2 is not sharp.

To obtain a better idea of the convergence behavior we have a closer look to the CG method. We have seen that CG minimizes $\|x - x_i\|_A$ on the Krylov subspace. This can also be seen as the construction of a polynomial q_i of degree i and $q_i(0) = 1$ such that

$$\|x - x_i\|_A = \|q_i(A)(x - x_0)\|_A = \min_{\substack{\tilde{q_i}, \\ \tilde{q}_i(0)=1}} \|\tilde{q}_i(A)(x - x_0)\|_A \ .$$

Suppose that the orthonormal eigen system of A is given by: $\{\lambda_j, y_j\}_{j=1,\ldots,n}$ where $Ay_j = \lambda_j y_j$, $\|y_j\|_2 = 1$, $y_j^T y_i = 0, j \neq i$, and $0 < \lambda_1 \leq \lambda_2 \ldots \leq \lambda_n$. The initial errors can be written as $x - x_0 = \sum\limits_{j=1}^{n} \gamma_j y_j$, which implies that

$$x - x_i = \sum_{j=1}^{n} \gamma_j q_i(\lambda_j) y_j \ . \tag{24}$$

If for instance $\lambda_1 = \lambda_2$ and $\gamma_1 \neq 0$ and $\gamma_2 \neq 0$ it is always possible to change y_1 and y_2 in $\tilde{y}_1$ and $\tilde{y}_2$ such that $\tilde{\gamma}_1 \neq 0$ but $\tilde{\gamma}_2 = 0$. This combined with equation (24) implies that if $q_i(\lambda_j) = 0$ for all different λ_j then $x_i = x$. So if there are only $m < n$ different eigenvalues the CG method stops at least after m iterations. Furthermore, the upper bound given in Theorem 3.2 can be sharpened.

<u>Remark</u>

For a given linear system $Ax = b$ and a given x_0 (note that $x - x_0 = \sum\limits_{j=1}^{n} \gamma_j y_j$) the quantities α and β are defined by:

$$\alpha = \min\{\lambda_j | \gamma_j \neq 0\} \,,$$
$$\beta = \max\{\lambda_j | \gamma_j \neq 0\} \,.$$

It is easy to show that the following inequality holds:

$$\|x - x_i\|_A \leq 2 \left(\frac{\sqrt{\frac{\beta}{\alpha}} - 1}{\sqrt{\frac{\beta}{\alpha}} + 1} \right)^i \|x - x_0\|_A. \tag{25}$$

The ratio $\frac{\beta}{\alpha}$ is called the effective condition number of A.

It follows from Theorem 3.1 that $r_0, \ldots, r_{k-1}$ forms an orthogonal basis for $K^k(A; r_0)$. So the vectors $\tilde{r}_i = r_i / \|r_i\|_2$ form an orthonormal basis for $K^k(A; r_0)$. We define the following matrices

$$\begin{aligned} &R_k \in I\!R^{n \times k} \text{ and the } j^{th} \text{ column of } R_k \text{ is } \tilde{r}_j, \\ &T_k = R_k^T A R_k \text{ where } T_k \in I\!R^{k \times k}. \end{aligned}$$

The matrix T_k can be seen as the projection of A on $K^k(A; r_0)$. It follows from Theorem 3.1 that T_k is a tridiagonal symmetric matrix. The coefficients of T_k can be calculated from the α_i's and β_i's of the CG process. The eigenvalues θ_i of the matrix T_k are called Ritz values of A with respect to $K^k(A; r_0)$. If z_i is an eigenvector of T_k so that $T_k z_i = \theta_i z_i$ and $\|z_i\|_2 = 1$ then $R_k\ z_i$ is called a Ritzvector of A. Ritzvalues and Ritzvectors are approximations of eigenvalues and eigenvectors and play an important role in a better understanding of the convergence behavior of CG. The properties of the Ritzvalues are given in more detail in Chapter 6. Some important properties are:

- the rate of convergence of a Ritzvalue to its limit eigenvalue depends on the distance of this eigenvalue to the rest of the spectrum
- in general the extreme Ritzvalues converge the fastest and their limits are α and β.

In practical experiments we see that, if Ritzvalues approximate the extreme eigenvalues of A, then the rate of convergence seems to be based on a smaller effective condition number (the extreme eigenvalues seem to be absent). We first give an heuristic explanation. Thereafter an exact result from the literature is cited.

From Theorem 3.1 it follows that $r_k = A(x - x_k)$ is perpendicular to the Krylov subspace $K^k(A; r_0)$. If a Ritzvector is a good approximation of an eigenvector y_j of A this eigenvector is nearly contained in the subspace $K^k(A; r_0)$. These two combined yields that $(A(x - x_k))^T y_j \cong 0$. The exact solution and the approximation can be written as

$$x = \sum_{i=1}^{n} (x^T y_i) y_i \quad \text{and} \quad x_k = \sum_{i=1}^{n} (x_k^T y_i) y_i.$$

From $(A(x - x_k))^T y_j = (x - x_k)^T \lambda_j y_j \cong 0$ it follows that $x^T y_j \cong x_k^T y_j$. So the error $x - x_k$ has a negligible component in the eigenvector y_j. This suggest that λ_j does no longer influence the convergence of the CG process.

For a more precise result we define a comparison process. The iterates of this process are comparable to that of the original process, but its condition number is less than that of original process.

Definition
Let x_i be the i-th iterate of the CG process for $Ax = b$. For a given integer i let $\boldsymbol{x}_j$ denote the j-th iterate of the comparison CG process for this equation, starting with $\boldsymbol{x}_0$ such that $x - \boldsymbol{x}_0$ is the projection of $x - x_i$ on span$\{y_2, \ldots, y_n\}$.

Note that for the comparison process the initial error has no component in the y_1 eigenvector.

Theorem 3.3 *[72]*
Let x_i be the i-th iterate of CG, and $\boldsymbol{x}_j$ the j-th iterate of the comparison process. Then for any j there holds:

$$\|x - x_{i+j}\|_A \leq F_i \|x - \boldsymbol{x}_j\|_A \leq F_i \frac{\|x - \boldsymbol{x}_j\|_A}{\|x - \boldsymbol{x}_0\|_A} \|x - x_i\|_A$$

with $F_i = \frac{\theta_1^{(i)}}{\lambda_1} \max_{k \geq 2} \frac{|\lambda_k - \lambda_1|}{|\lambda_k - \theta_1^{(i)}|}$ *, where $\theta_1^{(i)}$ is the smallest Ritz value in the i-th step of the CG process.*

<u>Proof</u>: see [72], Theorem 3.1.

The theorem shows that from any stage i on for which $\theta_1^{(i)}$ does not coincide with an eigenvalue λ_k, the error reduction in the next j steps is at most the fixed factor F_i worse than the error reduction in the first j steps of the comparison process in which the error vector has no y_1-component. As an example we consider the case that $\lambda_1 < \theta_1^{(i)} < \lambda_2$ we then have

$$F_i = \frac{\theta_1^{(i)}}{\lambda_1} \frac{\lambda_2 - \lambda_1}{\lambda_2 - \theta_1^{(i)}},$$

which is a kind of relative convergence measure for $\theta_1^{(i)}$ relative to λ_1 and $\lambda_2 - \lambda_1$. If $\frac{\theta_1^{(i)} - \lambda_1}{\lambda_1} < 0.1$ and $\frac{\theta_1^{(i)} - \lambda_1}{\lambda_2 - \lambda_1} < 0.1$ then we have $F_i < 1.25$. Hence, already for this modest degree of convergence of $\theta_1^{(i)}$ the process virtually converges as well as the comparison process (as if the y_1-component was not present). For more general results and experiments we refer to [72].

3.4 Exercises

1. Show that $(y, z)_A = \sqrt{y^T A z}$ is an inner product if A is symmetric and positive definite.

2. Give the proof of inequality (25).

3. (a) Show that an A-orthogonal set of nonzero vectors associated with a symmetric and positive definite matrix is linearly independent.

   (b) Show that if $\{v^{(1)}, v^{(2)}, \ldots, v^{(n)}\}$ is a set of A-orthogonal vectors in $I\!R^n$ and $z^T v^{(i)} = 0$ for $i = 1, \ldots, n$ then $z = 0$.

4. Define
$$t_k = \frac{(v^{(k)}, b - Ax^{(k-1)})}{(v^{(k)}, Av^{(k)})}$$
and $x^{(k)} = x^{(k-1)} + t_k v^{(k)}$, then $(r^{(k)}, v^{(j)}) = 0$ for $j = 1, \ldots, k$, if the vectors $v^{(j)}$ form an A-orthogonal set. To prove this, use the following steps using mathematical induction:

   (a) Show that $(r^{(1)}, v^{(1)}) = 0$.

   (b) Assume that $(r^{(k)}, v^{(j)}) = 0$ for each $k \le l$ and $j = 1, \ldots, k$ and show that this implies that
$$(r^{(l+1)}, v^{(j)}) = 0 \text{ for each } j = 1, \ldots, l.$$

   (c) Show that $(r^{(l+1)}, v^{(l+1)}) = 0$.

5. Take $A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{pmatrix}$ and $b = \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix}$. We are going to solve $Ax = b$.

   (a) Show that Conjugate Gradients applied to this system should convergence in 1 or 2 iterations (using the convergence theory).

   (b) Choose $x^{(0)} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ and do 2 iterations with the Conjugate Gradients method.

6. Suppose that A is nonsingular, symmetric, and indefinite. Give an example to show that the Conjugate Gradients method can break down.

4 Preconditioning of Krylov subspace methods

We have seen that the convergence behavior of Krylov subspace methods depends strongly on the eigenvalue distribution of the coefficient matrix. A preconditioner is a matrix that transforms the linear system such that the transformed system has the same solution but the transformed coefficient matrix has a more favorable spectrum. As an example we consider a matrix M which resembles the matrix A. The transformed system is given by

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

and has the same solution as the original system $Ax = b$. The requirements on the matrix M are the following:

- the eigenvalues of $M^{-1}A$ should be clustered around 1,
- it should be possible to obtain $M^{-1}y$ with low cost.

Most of this chapter contains preconditioners for symmetric positive definite systems (Section 4.1). For non-symmetric systems the ideas are analogously, so in Section 4.2 we give some details, which can be used only for non-symmetric systems.

4.1 The Preconditioned Conjugate Gradient (PCG) method

In Section 3.3 we observed that the rate of convergence of CG depends on the eigenvalues of A. Initially the condition number $\frac{\lambda_n}{\lambda_1}$ determines the decrease of the error. After a number of iterations the $\frac{\lambda_n}{\lambda_1}$ is replaced by the effective condition number $\frac{\lambda_n}{\lambda_2}$ etc. So the question arises, is it possible to change the linear system $Ax = b$ in such a way that the eigenvalue distribution becomes more favorable with respect to the CG convergence? This is indeed possible and the approach is known as: the preconditioning of a linear system. Consider the $n \times n$ symmetric positive definite linear system $Ax = b$. The idea behind Preconditioned Conjugate Gradients is to apply the "original" Conjugate Gradient method to the transformed system

$$\tilde{A}\tilde{x} = \tilde{b} \ ,$$

where $\tilde{A} = P^{-1}AP^{-T}$, $x = P^{-T}\tilde{x}$ and $\tilde{b} = P^{-1}b$, and P is a nonsingular matrix. The matrix M defined by $M = PP^T$ is called the preconditioner. The resulting algorithm can be rewritten in such a way that only quantities without a ˜ sign occurs.

Preconditioned Conjugate Gradient method

```
k = 0 ;       $x_0 = 0$ ;      $r_0 = b$ ;   initialization
while         $(r_k \neq 0)$ do             termination criterion
              $z_k = M^{-1} r_k$             preconditioning
              $k := k + 1$
              if $k = 1$ do
                 $p_1 = z_0$
              else
                 $\beta_k = \frac{r_{k-1}^T z_{k-1}}{r_{k-2}^T z_{k-2}}$          update of $p_k$
                 $p_k = z_{k-1} + \beta_k p_{k-1}$
              end if
              $\alpha_k = \frac{r_{k-1}^T z_{k-1}}{p_k^T A p_k}$
              $x_k = x_{k-1} + \alpha_k p_k$        update iterate
              $r_k = r_{k-1} - \alpha_k A p_k$      update residual
end while
```

Observations and properties for this algorithm are:

- it can be shown that the residuals and search directions satisfy:

$$\begin{aligned} r_j^T M^{-1} r_i &= 0 \quad , \quad i \neq j \ , \\ p_j^T (P^{-1} A P^{-T}) p_i &= 0 \quad , \quad i \neq j \ . \end{aligned}$$

- The denominators $r_{k-2}^T z_{k-2} = z_{k-2}^T M z_{k-2}$ never vanish for $r_{k-2} \neq 0$ because M is a positive definite matrix.

With respect to the matrix P we have the following requirements:

- the multiplication of $P^{-T} P^{-1}$ by a vector should be cheap. (comparable with a matrix vector product using A). Otherwise one iteration of PCG is much more expensive than one iteration of CG and hence preconditioning leads to a costlier algorithm.
- The matrix $P^{-1} A P^{-T}$ should have a favorable distribution of the eigenvalues. It is easy to show that the eigenvalues of $P^{-1} A P^{-T}$ are the same as for $P^{-T} P^{-1} A$ and $A P^{-T} P^{-1}$. So we can choose one of these matrices to study the spectrum.

In order to give more details on the last requirement we note that the iterate x_k obtained by PCG satisfies

$$x_k \in x_0 + K^k(P^{-T} P^{-1} A \ ; \ P^{-T} P^{-1} r_0), \text{ and} \tag{26}$$

$$\|x - x_k\|_A \leq 2 \left(\frac{\sqrt{K_2(P^{-1} A P^{-T})} - 1}{\sqrt{K_2(P^{-1} A P^{-T})} + 1} \right)^k \|x - x_0\|_A \ . \tag{27}$$

So a small condition number of $P^{-1} A P^{-T}$ leads to fast convergence. Two extreme choices of P show the possibilities of PCG. Choosing $P = I$ we get the original CG method back, whereas if $P^T P = A$ the iterate x_1 is equal to x so PCG converges in one iteration. For a classical paper on the success of PCG we refer to [53]. In the following pages some typical preconditioners are discussed.

Diagonal scaling

A simple choice for P is a diagonal matrix with diagonal elements $p_{ii} = \sqrt{a_{ii}}$. In [71] it has been shown that this choice minimizes the condition number of $P^{-1}AP^{-T}$ if P is restricted to be a diagonal matrix. For this preconditioner it is advantageous to apply CG to $\tilde{A}\tilde{x} = \tilde{b}$. The reason is that $P^{-1}AP^{-T}$ is easily calculated. Furthermore, $diag\ (\tilde{A}) = 1$ which saves n multiplications in the matrix vector product.

Basic iterative method

The basic iterative methods described in Section 2.2 use a splitting of the matrix $A = M - N$. In the beginning of Section 3.2 we show that the k-th iterate y_k from a basic method is an element of $x_0 + K^k(M^{-1}A, M^{-1}r_0)$. Using this matrix M in the PCG method we see that the iterate x_k obtained by PCG satisfies the following inequality:

$$\|x - x_k\|_A = \min_{z \in K^k(M^{-1}A; M^{-1}r_0)} \|x - z\|_A \ .$$

This implies that $\|x - x_k\|_A \leq \|x - y_k\|_A$, so measured in the $\| \ . \ \|_A$ norm the error of a PCG iterate is less than the error of a corresponding result of a basic iterative method. The extra costs to compute a PCG iterate with respect to the basic iterate are in general negligible. This leads to the notion that any basic iterative method based on the splitting $A = M - N$ can be accelerated by the Conjugate Gradient method so long as M (the preconditioner) is symmetric and positive definite.

Incomplete decomposition

This type of preconditioner is a combination of an iterative method and an approximate direct method. As illustration we use the model problem defined in Section 2.1. The coefficient matrix of this problem $A \in I\!R^{n \times n}$ is a matrix with at most 5 nonzero elements per row. Furthermore, the matrix is symmetric and positive definite. The nonzero diagonals are numbered as follows: m is number of grid points in the x-direction.

$$A = \begin{bmatrix} a_1 & b_1 & & c_1 & & & & \\ b_1 & a_2 & b_2 & & c_2 & & & \\ \vdots & \ddots & \ddots & & & \ddots & & \emptyset \\ c_1 & & b_m & a_{m+1} & b_{m+1} & & c_{m+1} & \\ & \ddots & \emptyset & \ddots & \ddots & \ddots & \emptyset & \ddots \\ & \emptyset & & & & & & \end{bmatrix} \qquad (28)$$

An optimal choice with respect to converge is take a lower triangular matrix L such that $A = L^T L$ and $P = L$ (L is the Cholesky factor). However it is well known that the zero elements in the band of A become non zero elements in the band of L. So the amount of work to construct L is large. With respect to memory we note that A can be stored in $3n$ memory positions, whereas L needs $m\ .\ n$ memory positions. For large problems the memory requirements are not easily fulfilled.

If the Cholesky factor L is calculated one observes that the absolute value of the elements in the band of L decreases considerably if the "distance" to the non zero elements of A increases. The non zero elements of L on positions where the elements of A are zero are called fill-in (elements). The observation of the decrease of fill-in motivates to discard fill in elements entirely, which leads to an incomplete Cholesky decomposition of A. Since the Cholesky decomposition is very stable this is possible without break down for a large class of matrices. To specify this in a precise way we use the following definition:

Definition 5.1.1
The matrix $A = (a_{ij})$ is an M-matrix if $a_{ij} \leq 0$ for $i \neq j$, the inverse A^{-1} exists and has positive elements $(A^{-1})_{ij} \geq 0$.

The matrix of our model problem is an M-matrix.
Furthermore, we give a notation for these elements of L which should be kept to zero. The set of all pairs of indices of off-diagonal matrix entries is denoted by

$$Q_n = \{(i,j) |\ i \neq j\ ,\ 1 \leq i \leq n\ ,\ 1 \leq j \leq n\ \}\ .$$

The subset Q of Q_n are the places (i,j) where L should be zero. Now the following theorem can be proved:

Theorem 4.1 *If A is a symmetric M-matrix, there exists for each $Q \subset Q_n$ having the property that $(i,j) \in Q$ implies $(j,i) \in Q$, a uniquely defined lower triangular matrix L and a symmetric nonnegative matrix R with $l_{ij} = 0$ if $(i,j) \in Q$ and $r_{ij} = 0$ if $(i,j) \notin Q$, such that the splitting $A = LL^T - R$ leads to a convergent iterative process*

$$LL^T x_{i+1} = Rx_i + b \quad \text{for every choice } x_0\ ,$$

where $x_i \rightarrow x = A^{-1}b$.

Proof (see [53]; p.151.)

After the matrix L is constructed it is used in the PCG algorithm. Note that in this algorithm multiplications by L^{-1} and L^{-T} are necessary. This is never done by forming L^{-1} or L^{-T}. It is easy to see that L^{-1} is a full matrix. If for instance one wants to calculate $z = L^{-1}r$ we compute z by solving the linear system $Lz = r$. This is cheap since L is a lower triangular matrix so the forward substitution algorithm can be used.

Example 5.1.3
We consider the model problem and compute a slightly adapted incomplete Cholesky decomposition: $A = LD^{-1}L^T - R$ where the elements of the lower triangular matrix L and diagonal matrix D satisfy the following rules:

a) $l_{ij} = 0$ for all (i,j) where $a_{ij} = 0 \quad i > j$,

b) $l_{ii} = d_{ii}$,

c) $(LD^{-1}L^T)_{ij} = a_{ij}$ for all (i,j) where $a_{ij} \neq 0 \quad i \geq j$.

In this example $Q^0 = \{(i,j) |\ \ |i-j| \neq 0, 1, m\}$
If the elements of L are given as follows:

$$L = \begin{bmatrix} \tilde{d}_1 & & & & \\ \tilde{b}_1 & \tilde{d}_2 & & & \\ & \ddots & \ddots & \emptyset & \\ \tilde{c}_1 & & \tilde{b}_m & \tilde{d}_{m+1} & \\ & \ddots & \emptyset & \ddots & \ddots \\ \emptyset & & & & \end{bmatrix} \tag{29}$$

it is easy to see that (using the notation as given in (28))

$$\left.\begin{array}{rcc} \tilde{d}_i &=& a_i - \frac{b_{i-1}^2}{\tilde{d}_{i-1}} - \frac{c_{i-m}^2}{\tilde{d}_{i-m}} \\ \tilde{b}_i &=& b_i \\ \tilde{c}_i &=& c_i \end{array}\right\} \quad i = 1, ..., n \; . \tag{30}$$

where elements that are not defined should be replaced by zeros. For this example the amount of work for $P^{-T}P^{-1}$ times a vector is comparable to the work to compute A times a vector. The combination of this incomplete Cholesky decomposition process with Conjugate Gradients is called the ICCG(0) method ([53]; p. 156). The 0 means that no extra diagonals are used for fill in. Note that this variant is very cheap with respect to memory: only one extra vector to store D is needed.

Another successfull variant is obtained by a smaller set Q. In this variant the matrix L has three more diagonals than the lower triangular part of the original matrix A. This preconditioning is obtained for the choice

$$Q^3 = \{(i,j) | \; |i-j| \neq 1, 2, m-2, m-1, m\}$$

For the formula's to obtain the decomposition we refer to ([53]; p. 156). This preconditioner combined with PCG is known as the ICCG(3) method. A drawback is that all the elements of L are different from the corresponding elements of A so 6 extra vectors are needed to store L in memory.

To give an idea of the power of the ICCG methods we have copied some results from [53]. As a first example we consider the model problem, where the boundary conditions are somewhat different:

$$\begin{array}{lll} \frac{\partial u}{\partial x}(x,y) = 0 & \text{for} & \left\{ \begin{array}{ll} x = 0 \; , & y \in [0,1] \\ x = 1 \; , & y \in [0,1] \end{array} \right. \quad , \\ \frac{\partial u}{\partial y}(x,y) = 0 & \text{for} & y = 1 \; , \; x \in [0,1] \; , \\ u(x,y) = 1 & \text{for} & y = 0 \; , \; x \in [0,1] \; . \end{array}$$

The distribution of the grid points is equidistant with $h = \frac{1}{31}$. The results for CG, ICCG(0) and ICCG(3) are plotted in Figure 6.

From inequality (27) it follows that the rate of convergence can be bounded by

$$r = \frac{\sqrt{K_2(P^{-1}AP^{-T})} - 1}{\sqrt{K_2(P^{-1}AP^{-T})} + 1}. \tag{31}$$

To obtain a better insight in the fast convergence of ICCG(0) and ICCG(3) the eigenvalues of $A, (L_0 L_0^T)^{-1}A$ and $(L_3 L_3^T)^{-1}A$ are computed and given in Figure 7. For this result given in [53] a small matrix of order $n = 36$ is used, so all eigenvalues can be calculated.
The eigenvalues as given in Figure 7 can be substituted in formula (31). We then obtain

$$\begin{array}{lll} r = 0.84 & \text{for} & \text{CG} \; , \\ r = 0.53 & \text{for} & \text{ICCG}(0) \; , \\ r = 0.23 & \text{for} & \text{ICCG}(3) \; , \end{array} \tag{32}$$

which explains the fast convergence of the PCG variants. In our explanation of the convergence behavior we have also used the notion of Ritz values. Applying these ideas to the given methods we note the following:

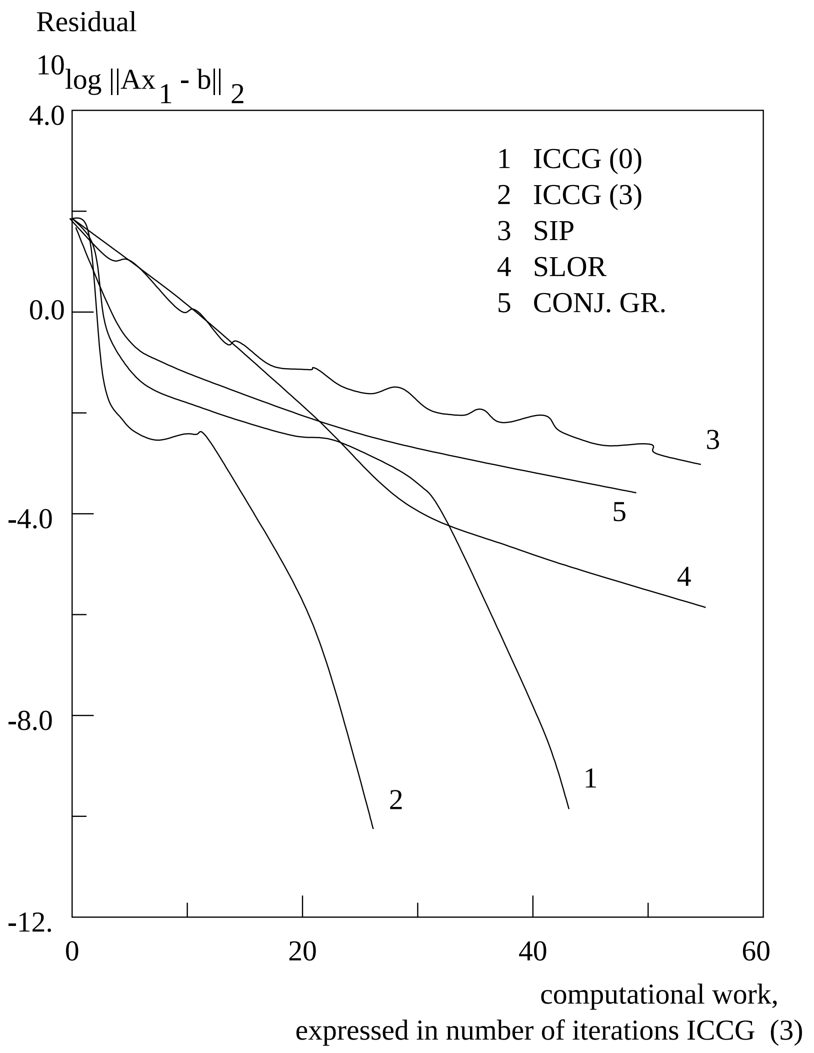


Figure 6: The results for the CG, ICCG(0) and ICCG(3) methods, compared with SIP (Strongly Implicit Procedure) and SLOR (Successive Line Over Relaxation method)

- For CG the eigenvalues of A are more or less equidistantly distributed. So if a Ritzvalue has converged we only expect a small decrease in the rate of convergence. This agrees with the results given in Figure 6, the CG method has a linear convergent behavior.

- For the PCG method the eigenvalue distribution is very different. Looking to the spectrum of $(L_3L_3^T)^{-1}A$ we see that $\lambda_{36} = 0.446$ is the smallest eigenvalue. The distance between λ_{36} and the other eigenvalues is relatively large which implies that there is a fast convergence of the smallest Ritz-value to λ_{36}. Furthermore, if the smallest Ritzvalue is a reasonable approximation of λ_{36} the effective condition number is much less than the original condition number. Thus super linear convergence is expected. This again agrees very well with the results given in Figure 6.

So the faster convergence of ICCG(3) comes from a smaller condition number and a more favorable distribution of the internal eigenvalues.

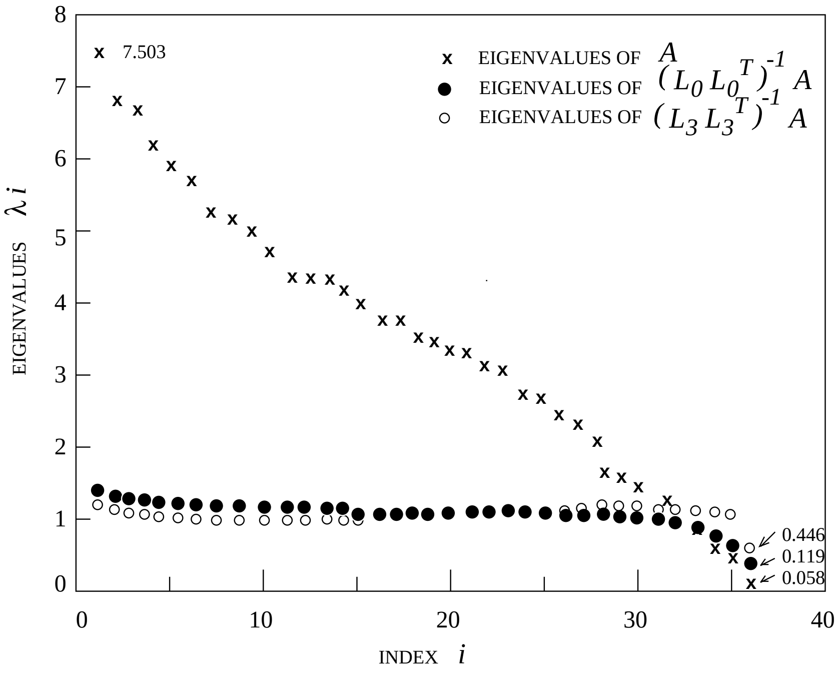


Figure 7: The eigenvalues of A, $(L_0L_0^T)^{-1}A$ and $(L_3L_3^T)^{-1}A$.

Finally, the influence of the order of the matrix on the number of iterations required to reach a certain precision was checked for both ICCG(0) and ICCG(3). Therefore several uniform rectangular meshes have been chosen, with mesh spacings varying from $\sim 1/10$ up to $\sim 1/50$. This resulted in linear systems with matrices of order 100 up to about 2500. In each case it was determined how many iterations were necessary, in order that the magnitude of each entry of the residual vector was below some fixed small number ε. In Figure 8 the number of iterations are plotted against the order of the matrices for $\varepsilon = 10^{-2}$, $\varepsilon = 10^{-6}$ and $\varepsilon = 10^{-10}$. It can be seen that the number of iterations, necessary to get the residual vector sufficiently small, increases only slowly for increasing order of the matrix. The dependence of $K_2(A)$ for this problem is $O(\frac{1}{h^2})$. For ICCG preconditioning it can be shown that there is a cluster of large eigenvalues of $(L_0L_0^T)^{-1}A$ in the vicinity of 1, whereas the small eigenvalues are of order $O(h^2)$ and the gaps between them are relatively large. So also for ICCG(0) the condition number is $O(\frac{1}{h^2})$. Faster convergence can be explained by the fact that the constant before $\frac{1}{h^2}$ is less for the ICCG(0) preconditioned system than for A and the distribution of the internal eigenvalues is much better so super linear convergence sets in after a small number of iterations.

The success of the ICCG method has led to many variants. In the following we describe two of them MICCG(0) given in [42] (MICCG means Modified ICCG) and RICCG(0) given in [5] (RICCG means Relaxed ICCG).

<u>MICCG</u>
In the MICCG method the MIC preconditioner is constructed by slightly adapted rules. Again A is splitted as follows $A = LD^{-1}L^T - R$, where L and D satisfy the following rules:

a) $l_{ij} = 0$ for all (i,j) where $a_{ij} = 0 \;\; i > j$,

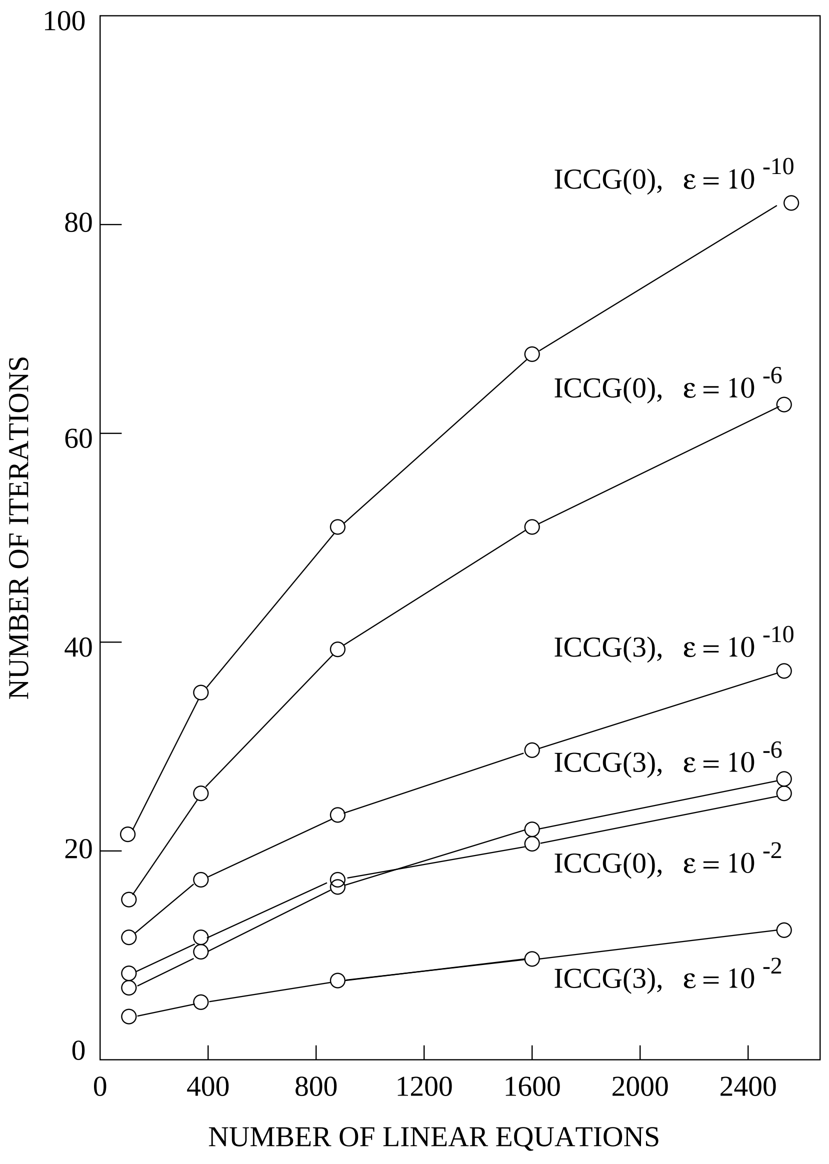


Figure 8: Effect of number of equations on the rate of convergence

b) $l_{ii} = d_{ii}$,

c) rowsum $(LD^{-1}L^T)$=rowsum(A) for all rows and $(LD^{-1}L^T)_{ij} = a_{ij}$ for all (i,j) where $a_{ij} \neq 0 \;\; i > j$.

A consequence of c) is that $LD^{-1}L^T \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} = A \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$ so $(LD^{-1}L^T)^{-1}A \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$.
this means that if $Ax = b$ and x and/or b are slowly varying vectors this incomplete Cholesky decomposition is a very good approximation for the inverse of A with respect to x and/or b. Using the same notation of L as given in (29) we obtain

$$\left.\begin{array}{lcl} \tilde{d}_i & = & a_i - (b_{i-1} + c_{i-1})\frac{b_{i-1}}{\tilde{d}_{i-1}} - (b_{i-m} + c_{i-m})\frac{c_{i-m}}{\tilde{d}_{i-m}} \\ \tilde{b}_i & = & b_i \\ \tilde{c}_i & = & c_i \end{array}\right\} \quad i = 1,..,n \tag{33}$$

It can be proved that for this preconditioning there is a cluster of small eigenvalues in the vicinity of 1 and the largest eigenvalues are of order $\frac{1}{h}$ and have large gap ratio's. So the condition number is $O(1/h)$.

In many problems the initial iterations of MICCG(0) converge faster than ICCG(0). Thereafter for both methods super linear convergence sets in. Using MICCG the largest Ritz values are good approximations of the largest eigenvalues of the preconditioned matrix. A drawback of MICCG is that due to rounding errors components in eigenvectors related to large eigenvalues return after some iterations. This deteriorates the rate of convergence. So if many iterations are needed ICCG can be better than MICCG.

In order to combine the advantage of both methods the RIC preconditioner is proposed in [5], which is an average of the IC and MIC preconditioner. For the details we refer to [5]. Only the algorithm is given: choose the average parameter $\alpha \in [0,1]$ then $\tilde{d}_i, \tilde{b}_i$ and $\tilde{c}_i$ are given by:

$$\left.\begin{array}{lcl} \tilde{d}_i & = & a_i - (b_{i-1} + \alpha c_{i-1})\frac{b_{i-1}}{\tilde{d}_{i-1}} - (\alpha b_{i-m} + c_{i-m})\frac{c_{i-m}}{\tilde{d}_{i-m}} \\ \tilde{b}_i & = & b_i \\ \tilde{c}_i & = & c_i \end{array}\right\} \quad i = 1,...,n \tag{34}$$

However the question remains: how to choose α? In Figure 9 which is copied from [74] a typical convergence behavior as function of α is given. This motivates the choice $\alpha = 0.95$, which leads to a very good rate of convergence on a wide range of problems.

Diagonal scaling
The above given preconditioners IC, MIC and RIC can be optimized with respect to work. One way to do this is to look at the explicitly preconditioned system:

$$D^{1/2}L^{-1}AL^{-T}D^{1/2}y = D^{1/2}L^{-1}b \tag{35}$$

Applying CG to this system one has to solve lower triangular systems of equations with matrix $LD^{-1/2}$. The main diagonal of this matrix is equal to $D^{1/2}$. It saves time if we can change this in such a way that the main diagonal is equal to the identity matrix. One idea could be to replace (35) by

$$D^{1/2}L^{-1}D^{1/2}D^{-1/2}AD^{-1/2}D^{1/2}L^{-T}D^{1/2}y = D^{1/2}L^{-1}D^{1/2}D^{-1/2}b \; .$$

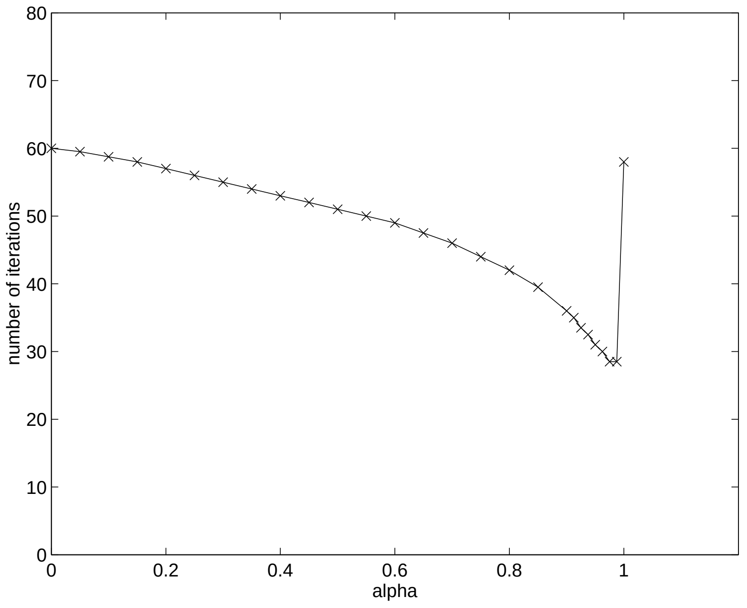


Figure 9: Convergence in relation with α

with $\tilde{A} = D^{-1/2}AD^{-1/2}$, $\tilde{L} = D^{-1/2}LD^{-1/2}$ and $\tilde{b} = D^{-1/2}b$ we obtain

$$\tilde{L}^{-1}\tilde{A}\tilde{L}^{-T}y = \tilde{L}\tilde{b} \, . \tag{36}$$

Note that $\tilde{L}_{ii} = 1$ for $i = 1, ..., n$. PCG now is the application of CG to this preconditioned system.

Eisenstat implementation

In this section we restrict ourselves to the IC(0), MIC(0) and RIC(0) preconditioner. We have already noted that the amount of work of one PCG iteration is approximately 2 times as large than a CG iteration. In [25] it is shown that much of the extra work can be avoided. If CG is applied to (36) products of the following form are calculated: $v_{j+1} = \tilde{L}^{-1}\tilde{A}\tilde{L}^{-T}v_j$. For the preconditioners used, the off diagonal part of $\tilde{L}$ is equal to the off-diagonal part of $\tilde{A}$. Using this we obtain:

$$\begin{aligned} v_{j+1} &= \tilde{L}^{-1}\tilde{A}\tilde{L}^{-T}v_j = \tilde{L}^{-1}(\tilde{L} + \tilde{A} - \tilde{L} - \tilde{L}^T + \tilde{L}^T)\tilde{L}^{-T}v_j \\ &= \tilde{L}^{-T}v_j + \tilde{L}^{-1}(v_j + (diag\ (\tilde{A}) - 2I)\tilde{L}^{-T}v_j) \end{aligned} \tag{37}$$

So v_{j+1} can be calculated by a multiplication by $\tilde{L}^{-T}$ and $\tilde{L}^{-1}$ and some vector operations. The saving in CPU time is the time to calculate the product of A times a vector. Now one iteration of PCG costs approximately the same as one iteration of CG.

General stencils

In practical problems the stencils in finite element methods may be larger or more irregular distributed than for finite difference methods. The same type of preconditioners can be used. However there are some differences. We restrict ourselves to the IC(0) preconditioner. For the five point stencil we see that the off diagonal part of L is equal to the strictly lower triangular part of A. For general stencils this property does not hold. Drawbacks are: All the elements of L should be stored, so the memory requirements of PCG are two times as large as for CG. Furthermore, the Eisenstat implementation can no longer be used. This motivates another preconditioner constructed by the following rules:

ICD (Incomplete Cholesky restricted to the Diagonal).
A is again splitted as $A = LD^{-1}L^T - R$ and L and D satisfy the following rules:

a) $l_{ij} = 0$ for all (i, j) where $a_{ij} = 0 \quad i > j$

b) $l_{ii} = d_{ii}, \quad i = 1, ..., n$

c) $l_{ij} = a_{ij}$ for all (i, j) where $a_{ij} \neq 0 \;\; i > j$
   $(LD^{-1}L^T)_{ii} = a_{ii} \quad i = 1, ..., n.$

This enables us to save memory (only the diagonal D should be stored) and CPU time (since now Eisenstat implementation can be used) per iteration. For large problems the rate of convergence of ICD is worse than for IC. Also MICD and RICD preconditioners can be given.

4.2 Preconditioning for general matrices

The preconditioning for non-symmetric matrices goes along the same lines as for symmetric matrices. There is a large amount of literature for generalization of the incomplete Cholesky decompositions. In general it is much more difficult to prove that the decomposition does not break down or that the resulting preconditioned system has a spectrum which leads to a fast convergence. Since symmetry is no longer important the number of possible preconditioners is much larger. Furthermore, if we have an incomplete LU decomposition of A, we can apply the iterative methods from 5.3.3 to the following three equivalent systems of equations:

$$U^{-1}L^{-1}Ax = U^{-1}L^{-1}b \; , \tag{38}$$

$$L^{-1}AU^{-1}y = L^{-1}b \; , \quad x = U^{-1}y \; , \tag{39}$$

or

$$AU^{-1}L^{-1}y = b \; , \quad x = U^{-1}L^{-1}y \; . \tag{40}$$

The rate of convergence is approximately the same for all variants. When the Eisenstat implementation is applied one should use (39). Otherwise we prefer (40) because then the stopping criterion is based on $\|r\|_2 = \|b - Ax_k\|_2$ whereas for (38) it is based on $\|U^{-1}L^{-1}r_k\|_2$, and for (39) it is based on $\|L^{-1}r_k\|_2$.

4.3 Exercises

1. Derive the preconditioned CG method using the CG method applied to $\tilde{A}\tilde{x} = \tilde{b}$.

2. (a) Show that the formula's given in (30) are correct.

   (b) Show that the formula's given in (33) are correct.

3. (a) Suppose that $a_i = 4$ and $b_i = -1$. Show that $\lim_{i\to\infty} \tilde{d}_i = 2 + \sqrt{3}$, where $\tilde{d}_i$ is as defined in (30).

   (b) Do the same for $a_i = 4$, $b_i = -1$ and $c_i = -1$ with $m = 10$, and show that $\lim_{i\to\infty} \tilde{d}_i = 2 + \sqrt{2}$.

   (c) Prove that the $LD^{-1}L^T$ decomposition (30) exists if $a_i = a, b_i = b, c_i = c$ and A is diagonally dominant.

4. **A practical exercise**
   Use as test matrices:

$$[a, f] = poisson(30, 30, 0, 0,' central')$$

   (a) Adapt the matlab cg algorithm such that preconditioning is included. Use a diagonal preconditioner and compare the number of iterations with cg without preconditioner.

   (b) Use the formula's given in (30) to obtain an incomplete $LD^{-1}L^T$ decomposition of A. Make a plot of the diagonal elements of D. Can you understand this plot?

   (c) Use the $LD^{-1}L^T$ preconditioner in the method given in (a) and compare the convergence behavior with that of the diagonal preconditioner.

5 Krylov subspace methods for general matrices

5.1 Introduction

In the preceding chapter we discuss the Conjugate Gradient method. This Krylov subspace method can only be used if the coefficient matrix is symmetric and positive definite. In this chapter we discuss Krylov subspace methods for an increasing class of matrices. For these we give different iterative methods, and at this moment there is no method which is the best for all cases. This is in contrast with the symmetric positive definite case. In Subsection 5.3.4 we give some guidelines for choosing an appropriate method for a given system of equations. In Section 5.2 we consider symmetric indefinite systems. General real matrices are the subject of Section 5.3. We end this chapter with a section containing iterative methods for complex linear systems.

5.2 Indefinite symmetric matrices

In this section we relax the condition that A should be positive definite (Chapter 3), and only assume that A is symmetric. This means that $x^TAx > 0$ for some x and possibly $y^TAy < 0$ for some y. For the real eigenvalues this implies that A has positive and negative eigenvalues. For this type of matrices $\|.\|_A$ defines no longer a norm. Furthermore, CG may have a serious break down since $p_k^TAp_k$ may be zero whereas $\|p_k\|_2$ is not zero. In this section we give two different (but related) methods to overcome these difficulties. These methods are defined in [57].

<u>SYMMLQ</u>

In the CG method we have defined the orthogonal matrix $R_k \in I\!R^{n\times k}$ where the j^{th} column of R_k is equal to the normalized residual $r_j/\|r_j\|_2$. It appears that the following relation holds

$$AR_k = R_{k+1}\bar{T}_k, \tag{41}$$

where $\bar{T}_k \in I\!R^{k+1\times k}$ is a tridiagonal matrix. This decomposition is also known as the Lanczos algorithm for the tridiagonalisation of A ([37]; p.477). This decomposition is always possible for symmetric matrices, also for indefinite ones. The CG iterates are computed as follows:

$$\begin{aligned} x_k &= R_k y_k, \quad \text{where} &(42)\\ R_k^TAR_ky_k &= T_ky_k = R_k^Tb. &(43)\end{aligned}$$

Note that T_k consists of the first k rows of $\bar{T}_k$. If A is positive definite then T_k is positive definite. It appears further that in the CG process an LDL^T factorization of T_k is used to solve (43). This can lead to break down because T_k may be indefinite in this section (compare [37]; Section 9.3.1, 9.3.2, 10.2.6). In the SYMMLQ method [57] problem (43) is solved in a stable way by using an LQ decomposition. So T_k is factorized in the following way:

$$T_k = \bar{L}_kQ_k \quad \text{where} \quad Q_k^TQ_k = I \tag{44}$$

with $\bar{L}_k$ lower triangular. For more details to obtain an efficient code we refer to [57] section 5.

MINRES

In SYMMLQ we have solved (43) in a stable way and obtain the "CG" iteration. However since $\|.\|_A$ is no longer a correct norm, the optimality properties of CG are lost. To get these back we can try to use the decomposition

$$AR_k = R_{k+1}\bar{T}_k$$

and minimize the error in the $\|.\|_{A^TA}$ norm. This is a norm if A is nonsingular. This leads to the following approximation:

$$x_k = R_k y_k, \tag{45}$$

where y_k is such that

$$\|Ax_k - b\|_2 = \min_{y\in I\!R^k} \|AR_k y - b\|_2. \tag{46}$$

Note that $\|AR_k y - b\|_2 = \|R_{k+1}\bar{T}_k y - b\|_2$ using (41).

Starting with $x_0 = 0$ implies that $r_0 = b$. Since R_{k+1} is an orthogonal matrix and $b = R_{k+1}\|r_0\|_2 e_1$, where $e_1 = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \in I\!R^{k+1}$, we have to solve the following least squares problem

$$\min_{y\in I\!R^k} \|\bar{T}_k y - \|r_0\|_2 e_1\|_2 \ . \tag{47}$$

Again for more details see [57]; Section 6.7. A comparison of both methods is given in [57]. In general the rate of convergence of SYMMLQ or MINRES for indefinite symmetric systems of equations is much worse than of CG for definite systems. Preconditioning techniques for these methods are specified in [64].

5.3 Iterative methods for general matrices

In this section we consider iterative methods to solve $Ax = b$ where the only requirement is that $A \in I\!R^{n\times n}$ is nonsingular. In the symmetric case we have seen that CG has the following two nice properties:

- optimality, the error is minimal measured in a certain norm,
- short recurrences, only the results of one foregoing step is necessary so work and memory do not increase for an increasing number of iterations.

It is shown in [27] that it is impossible to obtain a Krylov method based on $K^i(A; r_0)$, which has both properties for general matrices. So either the method has an optimality property but long recurrences, or no optimality and short recurrences. Recently some surveys on general iteration methods have been published: [13], [37] Section 10.4, [31], [66], [39], [8].

It appears that there are essentially three different ways to solve non-symmetric linear systems, while maintaining some kind of orthogonality between the residuals:

1. Solve the normal equations $A^TAx = A^Tb$ with Conjugate Gradients.

2. Construct a basis for the Krylov subspace by a 3-term bi-orthogonality relation.
3. Make all the residuals explicitly orthogonal in order to have an orthogonal basis for the Krylov subspace.

These classes form the subject of the following subsections. An introduction and comparison of these classes is given in [55].

5.3.1 CG applied to the normal equations

The first idea to apply CG to the normal equations

$$A^T A x = A^T b, \tag{48}$$

or

$$A A^T y = b \quad \text{with} \quad x = A^T y \tag{49}$$

is obvious. When A is nonsingular $A^T A$ is symmetric and positive definite. So all the properties and theoretical results for CG can be used. There are however some drawbacks first the rate of convergence now depends on $K_2(A^T A) = K_2(A)^2$. In many applications $K_2(A)^2$ is very large so the convergence of CG applied to (48) is very slow. Another difference is that CG applied to $Ax = b$ depends on the eigenvalues of A whereas CG applied to (48) depends on the eigenvalues of $A^T A$, which are equal to the singular values of A squared.

Per iteration a multiplication with A and A^T is necessary, so the amount of work is approximately two times as much as for the CG method. Furthermore, in several (FEM) applications Av is easily obtained but $A^T v$ not due to the unstructured grid and the data structure used. Finally not only the convergence depends on $K_2(A)^2$ but also the error due to rounding errors. To improve the numerical stability it is suggested in [9] to replace inner products like $p^T A^T A p$ by $(Ap)^T Ap$. Another improvement is the method LSQR proposed by [58]. This method is based on the application of the Lanczos method to the auxiliary system

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

This is a very reliable algorithm. It uses reliable stopping criteria and estimates of standard errors for x and the condition number of A.

5.3.2 BiCG type methods

In this type of methods we have short recurrences but no optimality property. We have seen that CG is based on the Lanczos algorithm. The Lanczos algorithm for non-symmetric matrices is called the bi-Lanczos algorithm. BiCG type methods are based on bi-Lanczos. In the Lanczos method we try to find a matrix Q such that $Q^T Q = I$ and

$$Q^T A Q = T \quad \text{tridiagonal} .$$

In the Bi-Lanczos algorithm we construct a similarity transformation X such that

$$X^{-1} A X = T \quad \text{tridiagonal} .$$

To obtain this matrix we construct a basis $r_0, ..., r_{i-1}$, which are the residuals, for $K^i(A; r_0)$ such that $r_j \perp K^j(A^T; s_0)$ and $s_0, ..., s_{i-1}$ form a basis for $K^i(A^T; s_0)$ such that $s_j \perp K^j(A; r_0)$, so the sequences $\{r_i\}$ and $\{s_i\}$ are bi-orthogonal. Using these properties and the definitions $R_k = [r_0 ... r_{k-1}]$, $S_k = [s_0 ... s_{k-1}]$ the following relation can be proved [75]:

$$AR_k = R_k T_k + \alpha_k r_k e_k^T \ , \tag{50}$$

and

$$S_k^T (Ax_k - b) = 0 \ .$$

Using (50), $r_0 = b$ and $x_k = R_k y$ we obtain

$$S_k^T R_k T_k y = s_0 r_0^T e_1. \tag{51}$$

Since $S_k^T R_k$ is a diagonal matrix with diagonal elements $r_j^T s_j$, we find, that if all these diagonal elements are nonzero,

$$T_k y = e_1 \ , \quad x_k = R_k y \ .$$

We see that this algorithm fails when a diagonal element of $S_k^T R_k$ becomes (nearly) zero, because these elements are used to normalize the vectors s_j (compare [37] §9.3.6). This is called a serious (near) break down. The way to get around this difficulty is the so-called look-ahead strategy. For details on look-ahead we refer to [61], and [32]. Another way to avoid break down is to restart as soon as a diagonal element gets small. This strategy is very simple, but one should realize that at a restart the Krylov subspace that has been built up so far, is thrown away, which destroys possibilities for faster (superlinear) convergence. (The description of the methods given below is based on those given in [75].)

BiCG

As has been shown for Conjugate Gradients, the LU decomposition of the tridiagonal system T_k can be updated from iteration to iteration and this leads to a recursive update of the solution vector. This avoids to save all intermediate r and s vectors. This variant of Bi-Lanczos is usually called Bi-Conjugate Gradients, or shortly Bi-CG [28]. Of course one can in general not be sure that an LU decomposition (without pivoting) of the tridiagonal matrix T_k exists, and if it does not exist then a serious break-down of the Bi-CG algorithm occurs. This break-down can be avoided in the Bi-Lanczos formulation of this iterative solution scheme. The algorithm is given as follows:

Bi-CG

```
$x_0$ is given; $r_0 = b - Ax_0$;
$\hat{r}_0$ is an arbitrary vector $(\hat{r}_0, r_0) \neq 0$
possible choice $\hat{r}_0 = r_0$ ;
$\rho_0 = 1$
$\hat{p}_0 = p_0 = 0$
for $i = 1, 2, ...$
        $\rho_i = (\hat{r}_{i-1}, r_{i-1})$ ; $\beta_i = (\rho_i / \rho_{i-1})$ ;
        $p_i = r_{i-1} + \beta_i p_{i-1}$ ;
        $\hat{p}_i = \hat{r}_{i-1} + \beta_i \hat{p}_{i-1}$ ;
        $v_i = Ap_i$
        $\alpha_i = \rho_i / (\hat{p}_i, v_i)$;
        $x_i = x_{i-1} + \alpha_i p_i$
        $r_i = r_{i-1} - \alpha_i v_i$
        $\hat{r}_i = \hat{r}_{i-1} - \alpha_i A^T \hat{p}_i$
    end for
```

Note that for symmetric matrices Bi-Lanczos generates the same solution as Lanczos, provided that $s_0 = r_0$, and under the same condition, Bi-CG delivers the same iterates as CG, for positive definite matrices. However, the Bi-orthogonal variants do so at the cost of two matrix vector operations per iteration step.

QMR

The QMR method [33] relates to Bi-CG in a similar way as MINRES relates to CG. For stability reasons the basis vectors r_j and s_j are normalized (as is usual in the underlying Bi-Lanczos algorithm).

If we group the residual vectors r_j, for $j = 0, ..., i-1$ in a matrix R_i, then we can write the recurrence relations as

$$AR_i = R_{i+1}\bar{T}_i \ ,$$

with

$$\bar{T}_i = \begin{array}{c} \longleftarrow \qquad i \qquad \longrightarrow \\ \left(\begin{array}{ccccc} \ddots & \ddots & & & \\ \ddots & \ddots & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \\ & & & \ddots & \end{array}\right) \end{array} \begin{array}{c} \uparrow \\ \\ i+1 \\ \\ \downarrow \end{array} \ .$$

Similar as for MINRES we would like to construct the x_i, with

$$x_i \in span\ \{r_0, Ar_0, ..., A^{i-1}r_0\} \ , \quad x_i = R_i\bar{y},$$

for which

$$\begin{aligned} \|Ax_i - b\|_2 &= \|AR_i\bar{y} - b\|_2 \\ &= \|R_{i+1}\bar{T}_i y - b\|_2 \\ &= \|R_{i+1}\{\bar{T}_i y - \|r_0\|_2 e_1\}\|_2 \end{aligned}$$

is minimal. However, in this case that would be quite an amount of work since the columns of R_{i+1} are not necessarily orthogonal. In [33] it is suggested to solve the minimum norm least squares problem

$$\min_{y \in \mathbb{R}^i} \|barT_i y - \|r_0\|_2 e_1\|_2 \ . \tag{52}$$

This leads to the simplest form of the QMR method. A more general form arises if the least squares problem (52) is replaced by a weighted least squares problem. No strategies are yet known for optimal weights, however. In [33] the QMR method is carried out on top of a look-ahead variant of the bi-orthogonal Lanczos method, which makes the method more robust.

Experiments suggest that QMR has a much smoother convergence behavior than Bi-CG, but it is not essentially faster than Bi-CG. For the algorithm we refer to [8] page 24.

CGS

For the bi-conjugate gradient residual vectors it is well-known that they can be written as $r_j = P_j(A)r_0$ and $\hat{r}_j = P_j(A^T)\hat{r}_0$, and because of the bi-orthogonality relation we have that

$$(r_j, \hat{r}_i) = (P_j(A)r_0, P_i(A^T)\hat{r}_0) = (P_i(A)P_j(A)r_0, \hat{r}_0) = 0 \ , \ \text{for } i < j.$$

The iteration parameters for bi-conjugate gradients are computed from innerproducts like above. Sonneveld observed in [70] that we can also construct the vectors $r_j = P_j^2(A)r_0$, using only the latter form of the innerproduct for recovering the bi-conjugate gradients parameters (which implicitly define the polynomial P_j). By doing so, it can be avoided that the vectors $\hat{r}_j$ have to be formed, nor is there any multiplication with the matrix A^T. The resulting CGS [70] method works in general very well for many non-symmetric linear problems. It converges often much faster than Bi-CG (about twice as fast in some cases). However, CGS usually shows a very irregular convergence behavior. This behavior can even lead to cancellation and a spoiled solution [74].

The following scheme carries out the CGS process for the solution of $Ax = b$, with a given preconditioner K:

```
Conjugate Gradient Squared method
    $x_0$ is an initial guess; $r_0 = b - Ax_0$;
    $\tilde{r}_0$ is an arbitrary vector, such that
    $(r_0, \tilde{r}_0) \neq 0$ ,
    e.g., $\tilde{r}_0 = r_0$ ; $\rho_0 = (r_0, \tilde{r}_0)$ ;
    $\beta_{-1} = \rho_0$ ; $p_{-1} = q_0 = 0$ ;
    for $i = 0, 1, 2, ...$ do
        $u_i = r_i + \beta_{i-1} q_i$ ;
        $p_i = u_i + \beta_{i-1}(q_i + \beta_{i-1} p_{i-1})$ ;
        $\hat{p} = K^{-1} p_i$ ;
        $\hat{v} = A\hat{p}$ ;
        $\alpha_i = \frac{\rho_i}{(\tilde{r}_0, \hat{v})}$ ;
        $q_{i+1} = u_i - \alpha_i \hat{v}$ ;
        $\hat{u} = K^{-1}(u_i + q_{i+1})$
        $x_{i+1} = x_i + \alpha_i \hat{u}$ ;
        if $x_{i+1}$ is accurate enough then quit;
        $r_{i+1} = r_i - \alpha_i A\hat{u}$ ;
        $\rho_{i+1} = (\tilde{r}_0, r_{i+1})$ ;
        if $\rho_{i+1} = 0$ then method fails to converge!;
        $\beta_i = \frac{\rho_{i+1}}{\rho_i}$ ;
    end for
```

In exact arithmetic, the α_j and β_j are the same constants as those generated by BiCG. Therefore, they can be used to compute the Petrov-Galerkin approximations for eigenvalues of A.

Bi-CGSTAB

Bi-CGSTAB [75] is based on the following observation. Instead of squaring the Bi-CG polynomial, we can construct other iteration methods, by which x_i are generated so that $r_i = \tilde{P}_i(A)P_i(A)r_0$ with other i^{th} degree polynomials $\tilde{P}_i$. An obvious possibility is to take for $\tilde{P}_j$ a polynomial of the form

$$Q_i(x) = (1 - \omega_1 x)(1 - \omega_2 x)...(1 - \omega_i x) \ ,$$

and to select suitable constants $\omega_j \in I\!R$. This expression leads to an almost trivial recurrence relation for the Q_i. In Bi-CGSTAB ω_j in the j^{th} iteration step is chosen as to minimize r_j, with respect to ω_j, for residuals that can be written as $r_j = Q_j(A)P_j(A)r_0$.
The preconditioned Bi-CGSTAB algorithm for solving the linear system $Ax = b$, with preconditioning K reads as follows:

```
Bi-CGSTAB method
    $x_0$ is an initial guess; $r_0 = b - Ax_0$;
    $\bar{r}_0$ is an arbitrary vector, such that $(\bar{r}_0, r_0) \neq 0$, e.g., $\bar{r}_0 = r_0$ ;
    $\rho_{-1} = \alpha_{-1} = \omega_{-1} = 1$ ;
    $v_{-1} = p_{-1} = 0$ ;
    for $i = 0, 1, 2, ...$ do
        $\rho_i = (\bar{r}_0, r_i)$ ; $\beta_{i-1} = (\rho_i/\rho_{i-1})(\alpha_{i-1}/\omega_{i-1})$ ;
        $p_i = r_i + \beta_{i-1}(p_{i-1} - \omega_{i-1}v_{i-1})$ ;
        $\hat{p} = K^{-1}p_i$ ;
        $v_i = A\hat{p}$ ;
        $\alpha_i = \rho_i/(\bar{r}_0, v_i)$ ;
        $s = r_i - \alpha_i v_i$ ;
        if $\|s\|$ small enough then
            $x_{i+1} = x_i + \alpha_i \hat{p}$ ; quit;
        $z = K^{-1}s$ ;
        $t = Az$ ;
        $\omega_i = (t, s)/(t, t)$ ;
        $x_{i+1} = x_i + \alpha_i \hat{p} + \omega_i z$ ;
        if $x_{i+1}$ is accurate enough then quit;
        $r_{i+1} = s - \omega_i t$ ;
    end for
```

The matrix K in this scheme represents the preconditioning matrix and the way of preconditioning [75]. The above scheme in fact carries out the Bi-CGSTAB procedure for the explicitly postconditioned linear system

$$AK^{-1}y = b \ ,$$

but the vectors y_i and the residual have been transformed back to the vectors x_i and r_i corresponding to the original system $Ax = b$. Compared to CGS two extra innerproducts need to be calculated.

In exact arithmetic, the α_j and β_j have the same values as those generated by Bi-CG and CGS. Hence, they can be used to extract eigenvalue approximations for the eigenvalues of A (see Bi-CG).

An advantage of these methods is that they use short recurrences. A disadvantage is that there is only a semi-optimality property. As a result of this, more matrix vector products are needed and no convergence properties have been proved. In experiments we see that the convergence behavior looks like CG for a large class of problems. However, the influence of rounding errors is much more important than for CG. Small changes in the algorithm can lead to instabilities. Finally it is always necessary to compare the norm of the updated residual to the exact residual $\|b - Ax_k\|_2$. If "near" break down had occurred these quantities may be different by several orders of magnitude. In such a case the method should be restarted.

5.3.3 GMRES-type methods

These methods are based on long recurrences, and have certain optimality properties. The long recurrences imply that the amount of work per iteration and required memory grow for increasing number of iterations. Consequently in practice one cannot afford to run the full algorithm, and it becomes necessary to use restarts or to truncate vector recursions. In this section we describe GMRES, GCR and a combination of both GMRESR.

GMRES
In this method, Arnoldi's method is used for computing an orthonormal basis $\{v_1, ..., v_k\}$ of the Krylov subspace $K^k(A; r_0)$. The modified Gram-Schmidt version of Arnoldi's method can be described as follows [67]:

```
1. Start: choose $x_0$ and compute $r_0 = b - Ax_0$ and $v_1 = r_0/\|r_0\|_2$,

2. Iterate: for $j = 1, ..., k$ do:
          $v_{j+1} = Av_j$
          for $i = 1, .., j$ do:
              $h_{ij} := v_{j+1}^T v_i$ , $v_{j+1} := v_{j+1} - h_{ij} v_i$ ,
          end for
          $h_{j+1,j} := \|v_{j+1}\|_2$ , $v_{j+1} := v_{j+1}/h_{j+1,j}$
      end for
   The entries of the upper $k+1 \times k$ Hessenberg matrix $\bar{H}_k$ are the scalars $h_{ij}$.
```

In GMRES (General Minimal RESidual method) the approximate solution $x_k = x_0 + z_k$ with $z_k \in K^k(A; r_0)$ is such that

$$\|r_k\|_2 = \|b - Ax_k\|_2 = \min_{z \in K^k(A; r_0)} \|r_0 - Az\|_2 \tag{53}$$

As a consequence of (53) it appears that r_k is orthogonal to $AK^k(A; r_0)$, so $r_k \perp K^k(A; Ar_0)$. If A is symmetric the GMRES method is equivalent to the MINRES method as described in [57]. Using the matrix $\bar{H}_k$ it follows that $AV_k = V_{k+1}\bar{H}_k$ where the $n \times k$ matrix V_k is defined by $V_k = [v_1, ..., v_k]$. With this equation it is shown in [67] that $x_k = x_0 + V_k y_k$ where y_k is the solution of the following least squares problem:

$$\|\beta e_1 - \bar{H}_k y_k\|_2 = \min_{y \in I\!R^k} \|\beta e_1 - \bar{H}_k y\|_2 \tag{54}$$

with $\beta = \|r_0\|_2$ and e_1 is the first unit vector in $I\!R^{k+1}$. GMRES is a stable method and no break down occurs, if $h_{j+1,j} = 0$ than $x_j = x$ so this is a "lucky" break down (see [67]; Section 3.4).

Due to the optimality (see inequality (53) convergence proofs are possible [67]. If the eigenvalues of A are real the same bounds on the norm of the residual can be proved as for the CG method. For a more general eigenvalue distribution we shall give one result in the following theorem. Let P_m be the space of all polynomials of degree less than m and let σ represent the spectrum of A.

Theorem 5.1 *Suppose that A is diagonalizable so that $A = XDX^{-1}$ and let*

$$\varepsilon^{(m)} = \min_{\substack{p \in P_m \\ p(0)=1}} \max_{\lambda_i \in \sigma} |p(\lambda_i)|$$

Then the residual norm of the m-th iterate satisfies:

$$\|r_m\|_2 \leq K(X)\varepsilon^{(m)} \|r_0\|_2 \tag{55}$$

where $K(X) = \|X\|_2 \|X^{-1}\|_2$. If furthermore all eigenvalues are enclosed in a circle centered at $C \in I\!R$ with $C > 0$ and having radius R with $C > R$, then

$$\varepsilon^{(m)} \leq \left(\frac{R}{C}\right)^m . \tag{56}$$

Proof: see [67]; p. 866.

For GMRES we see in many cases a super linear convergence behavior comparable to CG. Recently the same type of results are proved for GMRES [77]. As we have already noted in the beginning, work per iteration and memory requirements increase for an increasing number of iterations. In this algorithm the Arnoldi process requires k vectors in memory in the k-th iteration. Furthermore, $2k^2 \cdot n$ flops are needed for the total Gram Schmidt process. To restrict work and memory requirements one stops GMRES after m iterations, form the approximate solution and use this as a starting vector for a following application of GMRES. This is denoted by the GMRES(m) procedure. Not restarted GMRES is denoted by full GMRES. However restarting destroys many of the nice properties of full GMRES, for instance the optimality property is only valid inside a GMRES(m) step and the superlinear convergence behavior is lost. This is a severe drawback of the GMRES(m) method.

<u>GCR</u>
Slightly earlier than GMRES, [26] proposed the GCR method (Generalized Conjugate Residual method). The algorithm is given as follows:

```
GCR algorithm
choose $x_0$, compute $r_0 = b - Ax_0$

    for $i = 1, 2, ...$ do
       $s_i = r_{i-1}$ ,
       $v_i = As_i$ ,
       for $j = 1, ..., i-1$ do
          $\alpha = (v_i, v_j)$ ,
          $v_i := v_i - \alpha v_j$ ,  $s_i := s_i - \alpha s_j$ ,
       end for
       $v_i := v_i / \|v_i\|_2$ ,  $s_i := s_i / \|v_i\|_2$
```

```
        $x_i := x_{i-1} + (r_{i-1}, v_i)s_i$ ;
        $r_i := r_{i-1} - (r_{i-1}, v_i)v_i$ ;
    end for
```

The storage of s_i and v_i costs two times as much memory as for GMRES. The rate of convergence of GCR and GMRES are comparable. However there are examples where GCR breaks down. So comparing full GMRES and full GCR the first one is preferred in many applications.
When the required memory is not available GCR can be restarted. Furthermore, another strategy is possible which is known as truncation. An example of this is to replace the j-loop by

```
    for $j = i - m, ..., i - 1$ do
```

Now $2m$ vectors are needed in memory. Other truncation variants to discard search direction are possible. In general we see that truncated methods have a better convergence behavior especially if super linear convergence plays an important role. So if restarting or truncation is necessary truncated GCR is in general better than restarted GMRES. For convergence results and other properties we refer to [26].

GMRESR
Recently, methods are proposed to diminish the disadvantages of restarting and or truncation. One of these methods is GMRESR proposed in [78] and further investigated in [81]. This method consists of an outer- and an inner loop. In the inner loop we approximate the solution of a linear system with GMRES to find a good search direction. Thereafter in the outer loop the minimal residual approximation using these search directions is calculated by a GCR approach.

GMRESR algorithm
choose x_0 and m, compute $r_0 = b - Ax_0$

```
    for $i = 1, 2, ...$ do
        $s_i = P_{m,i-1}(A)r_{i-1}$ ,
        $v_i = As_i$ ,
        for $j = 1, ..., i - 1$ do
            $\alpha = (v_i, v_j)$ ,
            $v_i := v_i - \alpha v_j$ ,  $s_i := s_i - \alpha s_j$ ,
        end for
        $v_i := v_i / \|v_i\|_2$ ,  $s_i := s_i / \|v_i\|_2$
        $x_i := x_{i-1} + (r_{i-1}, v_i)s_i$ ;
        $r_i := r_{i-1} - (r_{i-1}, v_i)v_i$ ;
    end for
```

The notation $s_i = P_{m,i-1}(A)r_{i-1}$ denotes that one applies one iteration of GMRES(m) to the system $As = r_{i-1}$. The result of this operation is s_i. For $m = 0$ we get GCR, whereas for $m \to \infty$ one outer iteration is sufficient and we get GMRES. For the amount of work we refer to [78], where also optimal choices of m are given. In many problems the rate of convergence of GMRESR is comparable to full GMRES, whereas the amount of work and memory is much less. In the following picture we have tried to visualize the strong point of GMRESR in comparison with GMRES(m) and truncated GCR. A search directions is indicated by the symbol v_i. We see for GMRES(3) that after 3 iterations all information is thrown away. For GCR(3)

$v_1v_2v_3$ restart $\quad$ $v_1v_2v_3$ restart $\qquad$ GMRES(3)

$v_1v_2v_3\underbrace{v_4v_5v_6}_{\rightarrow}$ $\qquad$ ... $\qquad$ GCR truncated with 3 vectors

$$\begin{tikzcd}[row sep=small] \hat{v}_1\hat{v}_2\hat{v}_3 \arrow[d] & \hat{v}_1\hat{v}_2\hat{v}_3 \arrow[d] & \dots \\ \text{condense} & \text{condense} & \\ v_1 & v_2 & \end{tikzcd}$$

GMRESR with GMRES(3) as innerloop.

Figure 10: The use of search directions for restarted GMRES, truncated GCR and full GMRESR.

a window of the last 3 vectors moves from left to right. For GMRESR the information after 3 inner iterations is condensed in to one search direction so "no" information gets lost.
Also for GMRESR restarts and truncation is possible [81]. In the inner loop other iterative methods can be used. Several of these choices lead to a good iterative method. In theory we can call the same loop again, which motivates the name GMRES Recursive. A comparable approach is the FGMRES method give in [65]. Herein the outer loop consists of a slightly adapted GMRES algorithm. Since FGMRES and GMRESR are comparable in work and memory but FGMRES can not be truncated we prefer the GMRESR method.

5.3.4 Choice of an iterative method

For non-symmetric matrices it is very difficult to decide which iterative method should be used. All the methods treated here have their own type of problems for which they are winners. Furthermore, the choice depends on the computer used and the availability of memory. In general CGS and Bi-CGSTAB are easy to implement and reasonably fast for a large class of problems. If break down or bad convergence appear, GMRES like methods are better. Finally LSQR always converges but can take a large number of iterations.

In [81] we have tried to specify some easy to obtain parameters to facilitate a choice. First one should have an (crude) idea of the total number of iterations (mg) using full GMRES, secondly one should measure the ratio f which is defined as

$$f = \frac{\text{the CPU time used for one preconditioned matrix vector product}}{\text{the CPU time used for a vector update}}$$

Note that f depends on the used computer. Under certain assumptions given in [81] we obtain Figure 11. This figure gives only qualitative information. It illustrates the dependence of the choice on f and mg. If mg is large and f is small, Bi-CGSTAB is the best method. For large values of f and small values of mg the GMRES method is optimal and for intermediate values GMRESR is the best method. In [8] a flowchart is given with suggestions for the selection of a suitable iterative method.

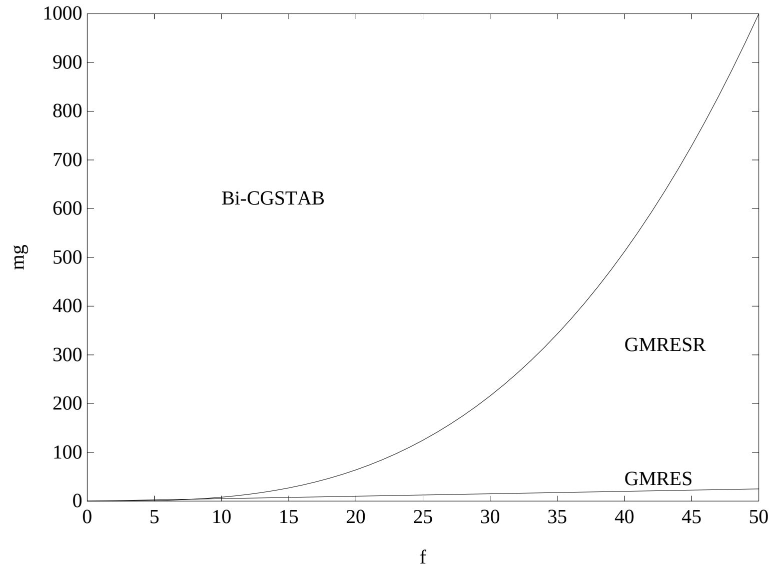


Figure 11: Regions of feasibility of Bi-CGSTAB, GMRES, and GMRESR.

5.3.5 Iterative methods for complex matrices

There are in practice, important applications that lead to linear systems where the coefficient matrix has complex valued entries. Examples are: complex Helmholtz equations, Schrödinger's equation, under water acoustics etc. If the resulting system is Hermitian the methods of Chapter 3 can be used. In these algorithms the inner product $x^T y$ should be replaced by the complex inner product $\bar{x}^T y$. For non Hermitian matrices iterative methods as given in Sections 5.3.1 to 5.3.3 can be used. Again they should be adapted to use the correct inner product.
In many applications the resulting complex linear systems have additional structure that can be exploited. For instance matrices of the following form arise:

$$A = e^{i\Theta}(T + \sigma I) \quad \text{where} \quad T = \bar{T}^T \ , \quad \Theta \in I\!R \ , \quad \sigma \in \mathbb{C}$$

Another special case that arises frequently in applications are complex symmetric matrices

$$A = A^T$$

For example, the complex Helmholtz equations leads to complex symmetric systems. For methods to solve these systems we refer to [32]; section 2.2, 2.3, 6.

5.4 Exercises

1. Show that the solution $\begin{pmatrix} y \\ x \end{pmatrix}$ of the augmented system

$$\begin{pmatrix} I & A \\ A^T & 0 \end{pmatrix} \begin{pmatrix} y \\ x \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix}$$

is such that x satisfies $A^TAx = A^Tb$.

2. Take the following matrix

$$\begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix}.$$

   (a) Suppose that GCR is applied to the system $Ax = b$. Show that GCR converges in 1 iteration if $x - x_0 = cr_0$, where $c \neq 0$ is a scalar and $r_0 = b - Ax_0$.

   (b) Apply GCR for the choices $b = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ and $x_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$.

   (c) Do the same for $x_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$.

3. In the GCR algorithm the vector r_i is obtained from vectorupdates. Show that the relation $r_i = b - Ax_i$ is valid.

4. Prove the following properties for the GMRES method:

   - $AV_k = V_{k+1}\bar{H}_k$,

   - $x_k = x_0 + V_ky_k$, where y_k is obtained from (54).

5. Figure 11 can give an indication which solution method should be used. Give an advice in the following situations:

   - Without preconditioning Bi-CGSTAB is the best method. What happens if preconditioning is added?
   - We use GMRESR for a stationary problem. Switching to an instationary problem, what are good methods?
   - We use GMRES. After optimizing the matrix vector product, which method is optimal?

6. **A practical exercise**
For the methods mentioned below we use as test matrices:

$$[a, f] = poisson(5, 5, 100, 0,' central')$$

and

$$[a, f] = poisson(5, 5, 100, 0,' upwind')$$

   (a) Adapt the matlab cg algorithm such that it solves the normal equations. Apply the algorithm to both matrices.

(b) Implement Bi-CGSTAB from the lecture notes. Take $K = I$ (identity matrix). Apply the algorithm to both matrices.

(c) Implement the GCR algorithm from the lecture notes. Apply the algorithm to both matrices.

(d) Compare the convergence behavior of the three methods.

6 Iterative methods for eigenvalue problems

6.1 Introduction

In many technical problems eigenvalues play an important role. For example eigenvalues give information of physical properties like eigenmodes, or eigenvalues are used to analyze and/or enhance mathematical methods for the solution of a physical problem.
As examples of the first kind we mention the following:

- eigenvalues are important to obtain eigenfrequences of a construction,
- characteristic properties of a fluid flow problem are defined using eigenvalues,
- if a bifurcation occurs eigenvalues and eigenvectors can be used to calculate a solution after the bifurcation point.

Examples of the second kind are:

- estimation of the 2-norm of a matrix (or its inverse),
- to predict and understand the convergence behavior of an iterative method,
- as a check of a discretization method. In general the matrices are so large that it is not easy to check their contents. However a small number of extreme eigenvalues can give sufficient information to decide wether the obtained discretization is correct or not,
- the choice of the time step for stable time integration methods.

In the remainder of this section we give some general information about eigenvalue problems.

The mathematical eigenvalue problem for a linear system of equations can be defined as follows: find $\lambda \in \mathbb{C}$ and $x \in \mathbb{C}^n$ such that $Ax = \lambda x$ and $x \neq 0$.
Some references for the theory on this type of problems are [37]; Chapter 7, 8, 9, [85], [17], [18] and [59]. Again the symmetric eigenvalue problem is much easier than the unsymmetric eigenvalue problem (compare the situation for linear systems). This observation not only holds from a computational point of view but also for the theory of the eigenvalue problem.
All methods to solve the eigenvalue problem are of an iterative nature. We distinguish between two different classes of methods. In the first class of methods the matrix A is transformed to a condensed form (computational costs $O(n^3)$) and the iteration process is applied to the condensed matrix (costs $O(n^2)$). As an example of these methods we mention the QR method. This class of methods is used in the public domain linear algebra software library LAPACK and is described in [37]; Chapter 7, 8. Drawback of these methods are that the matrix A should be given explicitly, and in general a large amount of memory is required. It is advised to use these methods for matrices with relatively small dimensions (say $n < 200$). In the second class the iteration is applied to the original matrix. A clear advantage is that the matrix A does not have to be available. The only requirement is that one is able to calculate matrix vector-products. It is advised to use this type of methods only if a small number of eigenvalues is wanted or the matrix A can not be easily formed.
In Section 6.2 we consider the classical power method. Krylov subspace methods are described in Section 6.3 for symmetric matrices and Section 6.4 for unsymmetric matrices.

6.2 The Power method

The Power method is the classical method to compute the largest few eigenvalues of a matrix. The method is motivated by the property that if we multiply a vector by a matrix, the contribution of the eigenvector corresponding to the largest eigenvalue (in absolute value sense) increased more than the contribution of the other eigenvectors. If the vector is multiplied a large number of times by the matrix, the contribution of this eigenvector will dominate, so the resulting iteration vector will approximate this eigenvector. So we arrive at the following algorithm.

```
The Power method
$q_0 \in \mathbb{C}^n$ is given
for $k = 1, 2, ...$
        $z_k = Aq_{k-1}$
        $q_k = z_k/\|z_k\|_2$
        $\lambda^{(k)} = q_{k-1}^T z_k$
endfor
```

It is easily be seen that if q_{k-1} is an eigenvector corresponding to λ_j then

$$\lambda^{(k)} = q_{k-1}^T A q_{k-1} = \lambda_j q_{k-1}^T q_{k-1} = \lambda_j \|q_{k-1}\|_2^2 = \lambda_j .$$

In order to derive the convergence behavior of the Power method we assume that the n eigenvalues are ordered such that $|\lambda_1| > |\lambda_2| \geq ... \geq |\lambda_n|$ and the eigenvectors by $x_1, ..., x_n$ so $Ax_i = \lambda_i x_i$. Each arbitrary start vector q_0 can be written as:

$$q_0 = a_1 x_1 + a_2 x_2 + ... + a_n x_n$$

and if $a_1 \neq 0$ if follows that

$$A^k q_0 = a_1 \lambda_1^k (x_1 + \sum_{j=2}^{n} \frac{a_j}{a_1} \left(\frac{\lambda_j}{\lambda_1} \right)^k x_j) \ . \tag{57}$$

Using this equality we conclude that

$$|\lambda_1 - \lambda^{(k)}| = O\left(\left| \frac{\lambda_2}{\lambda_1} \right|^k \right) \ , \quad \text{and also} \tag{58}$$

the angle between *span* $\{q_k\}$ and *span* $\{x_1\}$ is of order $|\frac{\lambda_2}{\lambda_1}|^k$.
These formula's (57) and (58) can be used to obtain the following observations. First it is important that $a_1 \neq 0$ so the starting vector should have a non-zero component in the x_1-vector. Due to rounding errors this is in general no problem because if q_0 has no component in the x_1 direction such a component is created during the computation. However, a large component in the start vector leads to a faster convergence. Secondly we see that the convergence depends on $|\frac{\lambda_2}{\lambda_1}|$. So applying the Power method to $A - cI$ the rate of convergence is equal to $|\frac{\lambda_2 - c}{\lambda_1 - c}|$. This property shows that the Power method is not shift invariant. Furthermore, it can be used to increase the convergence speed. Finally if c is chosen carefully we can compute other eigen values. For example: suppose $\lambda_i \in \mathbb{R}$ $i = 1, ..., n$ then the choice $c \cong \lambda_1$ leads to the fact that the in norm largest eigenvalue of $A - \lambda_1 I$ is equal to λ_n. So also the smallest

eigenvalue can be computed by the Power method. Thirdly we see that the Power method is a linearly converging method. This implies that the following stopping criterion can be used:

$$\text{estimate} \quad r \quad \text{from} \quad \tilde{r} = \frac{|\lambda^{(k+1)} - \lambda^{(k)}|}{|\lambda^{(k)} - \lambda^{(k-1)}|} \ , \tag{59}$$

and stop if $\frac{\tilde{r}}{1-\tilde{r}} \frac{|\lambda^{(k+1)}-\lambda^{(k)}|}{|\lambda^{(k+1)}|} \leq \varepsilon$.

This stopping criterion leads to $||\lambda_1| - \lambda^{(k+1)}| \leq \varepsilon$.

Note that there is a problem if $|\lambda_1| = |\lambda_2|$, which is the case for instance if $\lambda_1 = \bar{\lambda}_2$. A vector q_0 which has a nonzero component in x_1 and x_2 can be written as

$$q_0 = a_1 x_1 + a_2 x_2 + \sum_{j=3}^{n} a_j x_j \ .$$

The component in the direction of $x_3, ..., x_n$ will vanish in the Power method, but q_k will not tend to a limit. In [85], pp. 579-582 a method is given to obtain the eigenvalues λ_1 and λ_2 from the last three iterates. However when the imaginary part of λ_1 is small the obtained results have a poor accuracy.

The inverse Power method

We have seen that small eigenvalues can be computed by a correct shift of the matrix. However, in general the differences between small eigenvalues are much less then the differences between the large eigenvalues. So convergence to the smallest eigenvalue is very slow. A remedy for this is to apply the Power method to the inverse matrix A^{-1}. It is easily seen that the eigenvalues of A^{-1} are $\frac{1}{\lambda_i}$. So the smallest eigenvalue of A is the largest eigenvalue of A^{-1}. This leads to a much faster rate of convergence. As an example suppose

$$\lambda_1 = 1000 \ , \quad \lambda_{n-1} = 1.1 \ \text{ and } \ \lambda_n = 1 \ .$$

The rate of convergence of the Power method applied to

$$A - 1000I \ \text{ is equal to } \ \frac{|1.1 - 1000|}{1 - 1000|} = 0.99989$$

whereas application to

$$A^{-1} \ \text{ leads to } \ \frac{\frac{1}{1.1}}{\frac{1}{1}} = 0.909 \ .$$

In order to compute $z_k = A^{-1} q_{k-1}$ one solves the z_k from the linear system

$$A z_k = q_{k-1},$$

by Gaussian elimination, or an iterative solver. In general the inverse Power method costs less work than the Power method applied to the shifted matrix.

Orthogonal iteration

A straightforward generalization of the power method is "orthogonal iteration" which can be used to compute more than one eigenvalue. Let p be an integer less than n, and $Q_0 \in \mathbb{C}^{n \times p}$

an orthogonal matrix. Compute a sequence of matrices $\{Q_k\}$ where $Q_k \in \mathbb{C}^{n\times p}$ as follows:

```
for $k = 1, 2, ...$
    $Z_k = AQ_{k-1}$
    orthonormalize the columns of $Z_k$ such that
    $Q_k R_k = Z_k$ ,  $R_k \in \mathbb{R}^{k\times k}$ is an upper triangular matrix,
        and $\bar{Q}_k^T Q_k = I$.
endfor
```

This can be used to approximate the p largest eigenvalues. For more details we refer to [37]; Section 7.3.2.

6.3 A Krylov subspace method for symmetric matrices

Symmetry simplifies the real eigenvalue problem $Ax = \lambda x$ in two ways. It implies that all eigenvalues are real and that there is an orthogonal basis of eigenvectors. It can be shown that if A is a real $n \times n$ symmetric matrix then there exists a real orthogonal matrix Q such that

$$Q^T AQ = diag\ (\lambda_1, ..., \lambda_n)\ .$$

The iterative method considered in this section is known as the Lanczos method [50], [37]; Chapter 9. The relation between the Power method and the Lanczos method is comparable to the relation between basic iterative methods for linear systems and the CG method. We have seen that in the Power method one calculates $q_0, Aq_0, A^2q_0, ...$ and sees that the vector A^kq_0 tends to the eigenvector corresponding to the largest eigenvalue. In the Power method only one vector is used. To explain the properties of the Lanczos method we first define the Rayleigh quotient

$$r(x) = \frac{x^TAx}{x^Tx} \quad , \quad x \neq 0\ .$$

It is easily seen that $\min\limits_{x\in\mathbb{R}^n} r(x) = \lambda_n$ the smallest eigenvalue and $\max\limits_{x\in\mathbb{R}^n} r(x) = \lambda_1$ the largest eigenvalue.
In the Lanczos method the approximations after k iterations are $\theta_1^{(k)}$ of λ_1 and $\theta_k^{(k)}$ of λ_n. They satisfy the following (in)equalities

$$\theta_1^{(k)} = \max_{y\in K^k(A;q_0)} r(y) \leq \lambda_1$$

and

$$\theta_k^{(k)} = \min_{y\in K^k(A;q_0)} r(y) \geq \lambda_n\ .$$

These (in)equalities imply that $\theta_1^{(k)}$ is always closer to λ_1 than the approximation of the Power method. Furthermore, Lanczos gives an approximation of the smallest eigenvalue. The rate of convergence of $\theta_1^{(k)}$ to λ_1 and $\theta_k^{(k)}$ to λ_n is comparable. The Lanczos method involves partial tridiagonalizations of the matrix A. Information of A's extremal eigenvalues tends to emerge long before the tridiagonalization is complete. This makes the Lanczos algorithm particularly useful in situations where a few of A's largest or smallest eigenvalues are desired. Unfortunately, roundoff errors make the Lanczos method somewhat difficult to use in practice. The central problem is a loss of orthogonality among the Lanczos vectors that the iteration

produces. Some ideas are given to repair orthogonality. We start by the specification of the Lanczos algorithm:
Choose a starting vector q_1 where $\|q_1\|_2 = 1$
<u>Lanczos method</u>

```
$r_0 = q_1$ ;   $\beta_0 = 1$ ; $q_0 = 0$ ; $j = 0$           initialization

while   $\beta_j \neq 0$ do                                   iteration
        $q_{j+1} = r_j/\beta_j$                               normalization of $q$
        $j := j + 1$
        $\alpha_j = q_j^T A q_j$
        $r_j = (A - \alpha_j I)q_j - \beta_{j-1}q_{j-1}$      new direction
                                                            orthogonal to
        $\beta_j = \|r_j\|_2$                                 previous $q$.
end while
```

Thereafter we form the tridiagonal symmetric matrix T_j as follows

$$T_j = \begin{bmatrix} \alpha_1 & \beta_1 & & & 0 \\ \beta_1 & \alpha_2 & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & 0 & \ddots & \ddots & \beta_{j-1} \\ & & & \beta_{j-1} & \alpha_j \end{bmatrix} .$$

This matrix is called the Ritz matrix. The eigenvalues of T_j : $\theta_1^{(j)}, ..., \theta_j^{(j)}$ are called Ritz values and are approximations of the eigenvalues of A.
With respect to work we note that the Lanczos method costs one matrix vectorproduct per iteration and 5 vector operations. The memory requirements are 5 vectors in memory. Therafter the eigenvalues of T_j have to be calculated. Note that T_j is in general much smaller than A and has only three non zero elements per row. So this eigenvalue problem is always solved by a QR like method ([37]; Section 8.2) for instance by a call to a LAPACK subroutine.
The Lanczos vector q_j has several nice properties. In the following theorem it is proved that the vectors $q_1, ..., q_j$ form an orthonormal basis for $K^j(A; q_1)$.

Theorem 6.1 *Let $A \in I\!R^{n\times n}$ be symmetric and assume $q_1 \in I\!R^n$ satisfies $\|q_1\|_2 = 1$. Then the Lanczos algorithm runs until $j = m$ where m is the number of independent vectors in $K^n(A; q_1)$. Moreover for $j \in [1, m]$ we have*

$$AQ_j = Q_j T_j + r_j e_j^T \ , \qquad (60)$$

where $Q_j = [q_1, ..., q_j]$ has orthonormal columns that span $K^j(A; q_1)$.

Proof: see [37]; Section 9.1.3.

The Lanczos results can also be used to obtain an approximation of the eigenvectors of A. In order to do this all Lanczos vectors should be kept in memory. Suppose that $\theta_i^{(j)}$ is an eigenvalue of T_j and its corresponding eigenvector is denoted by s_i where $\|s_i\|_2 = 1$. The vector $y_i = Q_j s_i$ is called the Ritzvector and is an approximation of the eigenvector of A

belonging to the eigenvalue approximated by $\theta_i^{(j)}$. Heuristically this can be seen as follows: suppose $\|r_j e_j^T\|_2$ in (60) is small then

$$AQ_j \cong Q_j T_j$$

so

$$Ay_i = AQ_j s_i \cong Q_j T_j s_i = Q_j \theta_i^{(j)} s_i = \theta_i^{(j)} y_i \ .$$

It can be shown that $\|Ay_i - \theta_i^{(j)} y_i\|_2 = |\beta_j| \, |(s_i)_j|$ where $(s_i)_j$ denotes the final element of the vector s_i ([37]; Section 9.13). This equation can be used to obtain the following error bound:

$$\min_{\mu \in \lambda(A)} |\theta_i^{(j)} - \mu| \leq |\beta_j| \, |(s_i)_j| \quad i = 1, ..., j \ . \tag{61}$$

It is much cheaper to check this bound than forming y_i and compute $\|Ay_i - \theta_i^{(j)} y_i\|_2$. So (61) can be used as a stopping criterion.

In [37]; Section 9.1.4 some theoretical results are given on the convergence behavior of the extremal Ritzvalues. Suppose λ_1 is the largest eigenvalue of A than it is proved that the largest Ritzvalue θ_1 converges to λ_1. The speed of convergence depends on the so called gap-ratio

$$\rho_1 = \frac{(\lambda_1 - \lambda_2)}{(\lambda_2 - \lambda_n)} \ . \tag{62}$$

The value of ρ_1 measures the distance of λ_1 to the rest of the spectrum divided by the distance of λ_2 to λ_n. A large gap ratio leads to a fast convergence of θ_1 to λ_1.
It can be shown that the Lanczos algorithm is shift invariant. If it is applied to

$$\tilde{A} = A - cI \quad \text{the new matrix} \quad \tilde{T}_j \quad \text{is equal to} \quad \tilde{T}_j = T_j - cI \ .$$

So all the results are only shifted and the convergence speed remains the same. This is a clear difference with the Power method. This is in agreement with the fact that the gap ratio is shift invariant:

$$\tilde{\rho}_1 = \frac{\tilde{\lambda}_1 - \tilde{\lambda}_2}{\tilde{\lambda}_2 - \tilde{\lambda}_n} = \frac{\lambda_1 - c - (\lambda_2 - c)}{\lambda_2 - c - (\lambda_n - c)} = \frac{\lambda_1 - \lambda_2}{\lambda_2 - \lambda_n} = \rho_1 \ .$$

If the smallest eigenvalues of a matrix A are wanted it is a good idea to apply the Lanczos method to the inverse system

$$A^{-1} x = \mu x \ .$$

This can lead to a much better gap ratio. Note that $\mu = \frac{1}{\lambda}$. Suppose we have an example where $\lambda_1 = 1000$, $\lambda_{n-1} = 1.1$ and $\lambda_n = 1$. The gap ratio for the smallest eigenvalue is equal to

$$\rho_n = \frac{|\lambda_n - \lambda_{n-1}|}{|\lambda_{n-1} - \lambda_1|} = \frac{0.1}{1000} = 10^{-4}$$

so the iteration takes very long to obtain a good approximation. For the inverse problem we want to calculate the largest eigenvalue

$$\mu_1 = 1 \ , \ \mu_2 = \frac{1}{1.1} \ , \ ..., \ \mu_n = \frac{1}{1000}$$

the gap ratio is now equal to

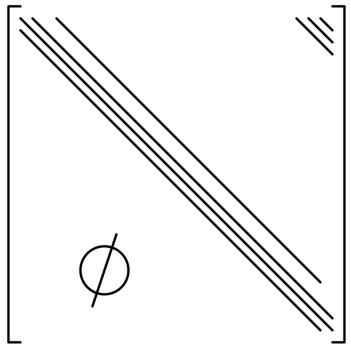
$$\tilde{\rho}_1 = \frac{|\mu_1 - \mu_2|}{|\mu_2 - \mu_n|} = \frac{1 - \frac{1}{1.1}}{\frac{1}{1.1} - \frac{1}{1000}} \simeq 0.1$$

which is much larger than for the original matrix. A drawback is again that one has to solve a linear system of equations in every iteration.

The convergence of Ritz values to interior eigenvalues is not so good. Moreover theoretical results for this convergence are not sharp. In general the same behavior as the for CG method applied to linear systems is observed. So if the Ritzvalue θ_1 is close to λ_1, the method behaves as if the eigenvalue λ_1 is absent. So once λ_1 has been approximated, θ_2 converges faster to λ_2.
With respect to rounding errors we note that equation (60) holds to working precision. However loss of orthogonality of the computed vectors q_j appears if one of the Ritz values converges to an eigenvalue. One remedy is to orthogonalize each newly computed Lanczos vector against its predecessors. This leads to the complete reorthogonalization Lanczos method. However, such an orthogonalization requires many vector operations. This makes the method unpractical if many iterations are necessary. To decrease the costs, a selective orthogonalization procedure is proposed [37]; Section 9.2.4. In this algorithm the new Lanczos vector is not orthogonalized against all its predecessors, but only against the much smaller set of converged Ritz vectors. For details we refer to [60], [69] and [46].

6.4 Krylov subspace methods for unsymmetric matrices

Arnoldi
A generalization of the Lanczos method to unsymmetric matrices is the Arnoldi method [2]. In this method the matrix A is transformed to an upper Hessenberg matrix by an orthogonal transformation. An upper Hessenberg matrix has the following nonzero pattern:



The relation between Lanczos and Arnoldi is comparable to the relation between CG and GMRES for linear systems. The Arnoldi method has the same nice properties with respect to convergence as the Lanczos method. A drawback is that for Lanczos only 5 vector operations are necessary during computation, whereas for Arnoldi the number of vector operations is proportional to the number of iterations.
The Arnoldi algorithm is given by: choose a starting vector q_1 where $\|q_1\|_2 = 1$.
Arnoldi method

```
$r = q_1$ ;      $\beta = 1$ ;  $j = 0$    initialization

while        $\beta \neq 0$              iteration
    $q_{j+1} = r/\beta$ ;               normalization
    $j = j + 1$ ; $r = Aq_j$ ;
    for $i = 1, ..., j$                 modified Gram
        $h_{ij} = q_i^T r$              Schmidt orthogonalization
        $r = r - h_{ij} q_i$
    end for
    $\beta = \|r\|_2$
    if $j < n$
        $h_{j+1,j} = \beta$
    end if
end while
```

After the iteration is stopped one can form the Hessenberg matrix H_j as follows

$$H_j = \begin{bmatrix} h_{11} & \dots & \dots & h_{1j} \\ h_{21} & \ddots & & \vdots \\ & \ddots & \ddots & \vdots \\ O & & h_{jj-1} & h_{jj} \end{bmatrix} .$$

H_j is the Ritzmatrix, θ_i the Ritzvalue. In the same way as for Lanczos we have if $H_j s_i = \theta_i s_i$ then $Q_j s_i$ is an approximation of the corresponding eigenvector.

If the matrix A is symmetric the matrix H_j becomes tridiagonal so we get the same results as using the Lanczos method. Many of the properties of the Lanczos method can be generalized to the Arnoldi method. In general it is a stable method with respect to rounding errors. There is no break down possible, only the case $\beta = 0$ can occur, but then an invariant subspace is obtained and all eigenvalues can be calculated (Assuming that q_0 has nonzero components in all eigenvector directions). A drawback of this method is the fact that due to the modified Gram Schmidt orthogonalization the amount of work increases quadratically. Restarting the Arnoldi method prevents this, however in such a case the good convergence properties are lost. The rate of convergence can be much different from Lanczos, because complex eigenvalues can occur. If all the eigenvalues are real than the convergence behavior of Arnoldi is comparable to Lanczos. In the general case of complex eigenvalues we see that the Ritz values converging to the extreme eigenvalues are converging much faster than the interior ones.

Bi-Lanczos
To get rid of the Gram Schmidt process another generalization is proposed: Bi-Lanczos. It is possible to reduce A to tridiagonal form using a general similarity transformation. However, this leads to an unstable procedure ([85]; pp. 388-405). In the Bi-Lanczos procedure two vector sequences are produced x_j and y_j, which have the property that they are bi-orthogonal: if $X_j = [x_1, ..., x_j]$, $Y_j = [y_1, ..., y_j]$ we have $X^T Y = I$. The Bi-Lanczos method runs as follows: choose starting vectors x_1, y_1 such that $x_1^T y_1 = 1$.

Bi-Lanczos
$j = 0$, $\beta_0 = 1$, $x_0 = 0$, $r_0 = x_1$, $y_0 = 0$, $p_0 = y_1$

```
while $\beta_j \neq 0 \bigwedge r_j^T p_j \neq 0$
    $\gamma_j = r_j^T p_j / \beta_j$
    $x_{j+1} = r_j/\beta_j$ ; $y_{j+1} = p_j/\gamma_j$
    $j := j+1$
    $\alpha_j = y_j^T A x_j$ ; $r_j = (A - \alpha_j I)x_j - \gamma_{j-1} x_{j-1}$ ,
    $\beta_j = \|r_j\|_2$ ; $p_j = (A - \alpha_j I)^T y_j - \beta_{j-1} y_{j-1}$ ,
end while
```

The tridiagonal Ritz matrix T_j is formed by:

$$T_j = \begin{bmatrix} \alpha_1 & \gamma_1 & & & \\ \beta_1 & \alpha_2 & \gamma_2 & O & \\ & \ddots & \ddots & \ddots & \\ & O & \ddots & \ddots & \gamma_{j-1} \\ & & & \beta_{j-1} & \alpha_j \end{bmatrix} .$$

The amount of work per iteration is equal to two matrix vector products one with A and the other with A^T. Furthermore, 12 vector operations are needed. To circumvent stability problems the look ahead Lanczos procedures are developed per iteration [32]. However, many open questions remain as there are: how to implement complete or selective orthogonalization, which stop criterion can be used, multiple eigenvalues etc.

6.5 The generalized eigenvalue problem

In practical finite element eigenvalue problems one also wants to solve generalized eigenvalue problems. In such a problem one has to solve the following problem: for $A, B \in \mathbb{R}^{n \times n}$ given, compute $\lambda \in \mathbb{C}$ and $x \in \mathbb{C}^n$ where $x \neq 0$ such that

$$Ax = \lambda Bx . \tag{63}$$

In finite element problems A may be the stiffness matrix and B the mass matrix. For theoretical properties and QR like methods we refer to [37]; Section 7.7. If A and B are symmetric and B also positive definite we refer to [37]; Section 8.7.2. If B^{-1} exists (63) can be transformed to

$$B^{-1}Ax = \lambda x \tag{64}$$

so iteration methods can be applied to (64).
There are also iterative methods which are suited to be applied to (63) directly. For these methods we refer to: [36] and [73].

6.6 Exercises

1. The Power method can be used to approximate the largest eigenvalue λ_1. In this exercise two methods are given to estimate the eigenvalue λ_2 if λ_1 and eigenvector x_1 are known.
   (a) Take $q_0 = (A - \lambda_1 I)q$, where q is an arbitrary vector. Show that the Power method applied to this starting vector leads to an approximation of λ_2 (Annihilation Technique).
   (b) Take A is symmetric and show that if the Power method is applied to the matrix
   $$B = A - \frac{\lambda_1}{x_1^T x_1} x_1 x_1^T$$
   one gets an approximation of λ_2. What is the amount of work per iteration using B (Hotelling Deflation).

2. Suppose that $A \in I\!R^{n\times n}$ is skew-symmetric.
   (a) Derive a Lanczos-like algorithm for computing a skew-symmetric tridiagonal matrix T_m such that
   $$AQ_m = Q_m T_m,$$
   where $m = dim\{K(A; q_1, n)\}$ and $Q_m^T Q_m = I_m$.
   (b) Show that if m is equal to the dimension of the smallest invariant subspace for A that contains q_1.

3. Suppose $A \in I\!R^{n\times n}$ is symmetric and that we wish to compute its largest eigenvalue. Let η be an approximate eigenvector and set
   $$\alpha = \frac{\eta^T A \eta}{\eta^T \eta}, \quad z = A\eta - \alpha\eta.$$
   (a) Show that there is an eigenvalue of A in the interval $[\alpha - \delta, \alpha + \delta]$, where $\delta = \|z\|_2 / \|\eta\|_2$.
   (b) Consider $\bar{\eta} = a\eta + bz$ and show how to determine a and b such that $\bar{a} = \bar{\eta}^T A \bar{\eta} / \bar{\eta}^T \bar{\eta}$ is maximal.
   (c) Compare this with the first two iterations of Lanczos.

4. **A practical exercise**
   A bending beam with a force P on top (see Figure 12) can be described by the following equation:
   $$EI\frac{d^2w}{dx^2} = Pw, \; w(0) = w(L) = 0.$$
   The solution of this equation is $w(0) = 0$. For certain values of P, there is also a non-trivial solution. The smallest value of such a P is: $P = \frac{EI\pi^2}{L^2}$.

   We can also approximate the smallest value of P by the smallest eigenvalue of
   $$A = \frac{EI}{h^2}\begin{pmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & \cdots \\ 0 & -1 & 2 & -1 & 0 & \cdots \\ & \ddots & \ddots & \ddots & \ddots & \ddots \end{pmatrix},$$

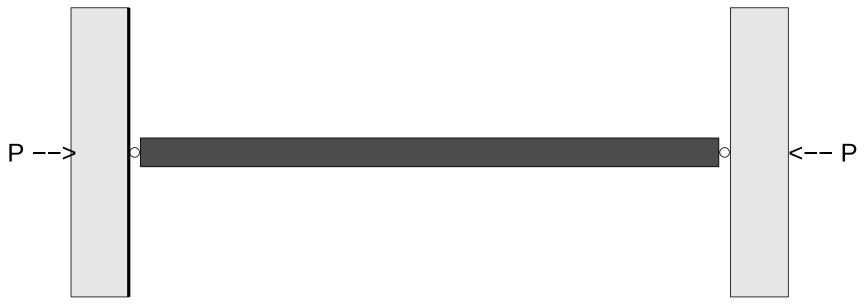


Figure 12: Bending beam configuration

where $A \in I\!R^{n\times n}$ and $h = \frac{L}{n+1}$. Take $n = 100$, $EI = 10$ and $L = 2$.

(a) Compute an approximation of P by doing 50 iterations of the inverse Power method applied to A.

(b) Do 10 iterations with the Lanczos method and form T_{10}.

(c) Compute the eigenvalues of T_j using the 'eig' command of Matlab.

(d) Compare the convergence of the inverse Power method and the Lanczos method.

7 Vector computers

The solution of large 3D PDE problems requires a considerable amount of memory and computing time. At present the fastest computers are provided with a large and fast internal memory (core). Two types of supercomputers are distinguished, the so called vector computers and parallel computers. Both types of machines have their own characteristic properties. It is only possible to obtain fast algorithms if these properties are taken into account. In the following sections we shall give a short introduction to vector machines. The following chapter contains an introduction to parallel computers. For more details we refer to the following surveys: [56], [37]; Sections 1.4, 6.1, 6.2, [21] and [20]. It is much easier to implement an algorithm on a supercomputer by using FORTRAN95 instead of FORTRAN77. For a good description of the language we refer to [54]. Finally we note that it makes only sense to optimize an algorithm for a supercomputer if the original code is optimized with respect to the number of floating point operations. For instance if ICCG(0) is used a diagonal scaling, and Eisenstat implementation should have been used.

7.1 Introduction

We shall give a short description of the special features of a vector computer. Thereafter we shall show which parts of ICCG are easily vectorizable. For the parts which are difficult to vectorize we present some ideas to obtain better vectorizable code.

Pipe-lining and time model
In a computer an internal clock influences all the operations. In every clock cycle one operation is performed: for instance fetching a number from memory, part of an addition of two real numbers, etc. The length of one clock cycle influences the speed of the computer. In order to explain the special nature of a vector computer we analyze the addition of two reals. In classical functional units it takes some clock cycles to add two real number. To understand this we refer to Figure 13 where we see that 5 different sub-operations are used. This means that in a classical computer every 5 clock cycles one results is obtained. In a vector functional

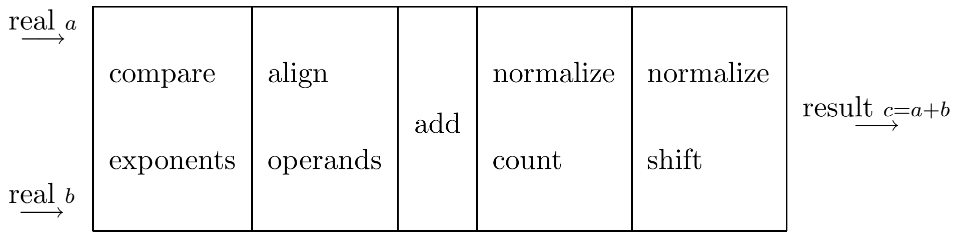
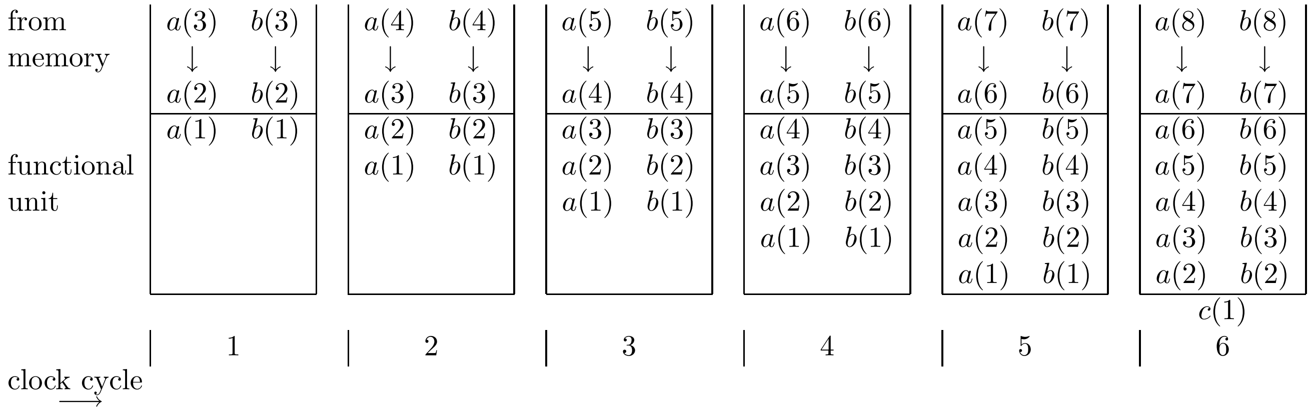


Figure 13: The add operation divided into 5 sub-operations

unit each sub-operation is executed on a piece of hardware that operates concurrently with the other stages of the pipeline. Each sub-operation is started at the beginning of a clock cycle and completed at the end of a clock cycle. Note that the concept of pipe-lining is similar to that of an assembly line process in an industrial plant. Furthermore, there is no gain if one pair of reals is added. This technique makes only sense if a large amount of reals in a row are added. As an example we mention the following operation:

```
do $i = 1, n$
    $c(i) = a(i) + b(i)$
enddo
```

To follow this process we give in Figure 14 the first clock cycles of this operation. It is easily seen that the first 5 clock cycles no result appears, but in the 6th clock cycle we get the first



from memory
functional unit
clock cycle
a(3) b(3) a(2) b(2) a(1) b(1)
a(8) b(8) a(7) b(7) a(6) b(6) a(5) b(5) a(4) b(4) a(3) b(3) a(2) b(2)
c(1)
1
2
3
4
5
6

Figure 14: The first clock cycles for adding two vectors

result, in the 7th the second result etc. So for a classical functional unit the addition costs $5n$ clock cycles, whereas for the vector functional unit this costs $5 + n$ clock cycles. This is much better for n large enough. Suppose one clock cycle costs μ seconds. The time to compute the addition is equal to

$$t(n) = t_{start} + n\mu \; .$$

In our example the start-up time $t_{start} = 5\mu$. For other operations the start-up time may be different.

Mflop rates

To compare different machines the speed is measured in Mflop. One Mflop means that 10^6 flops (floating point operations $+, *, -$) are performed in one second. In the given example the speed $R(n)$ is given by

$$R(n) = \frac{\text{number of flops } * 10^{-6}}{\text{required time}} = \frac{n}{t_{start} + n\mu} \cdot 10^{-6} \quad \text{Mflop.} \tag{65}$$

Using (65) two characteristic quantities are defined [45] :
the asymptotic rate of performance.

$$R_\infty = \lim_{n\to\infty} R(n) = \frac{10^{-6}}{\mu} \quad \text{Mflop} \; , \tag{66}$$

and $n_{\frac{1}{2}}$, the smallest n for which half of the peak performance is achieved so:

$$R(n_{\frac{1}{2}}) = \frac{10^{-6}}{2\mu} \; . \tag{67}$$

For the operation given this leads to

$$\frac{n_{\frac{1}{2}}}{t_{start} + n_{\frac{1}{2}}\mu} = \frac{1}{2\mu}$$

so

$$n_{\frac{1}{2}} = \frac{t_{start}}{\mu}. \tag{68}$$

Machines which have large $n_{\frac{1}{2}}$ values do not perform well on short vectors.

Vector registers

Until now we have not said how to obtain all these reals from memory and how to store the results. This is very important since memory is in general slower than the functional units. The first special feature is an intermediate very-high-speed memory the so-called vector register. The reals are first fetched from the "slow" main memory and put into the fast vector registers. The results c are stored in a vector register and then put in main memory. In general several vector registers are used also to store intermediate results. In this way it is possible to link certain operations together, for instance the loop

```
for $i = 1, ..., n$ do
    $x(i) = x(i) + a * y(i)$
end for
```

can be done in the way sketched in Figure 15. So after some start-up time every clock cycle 2

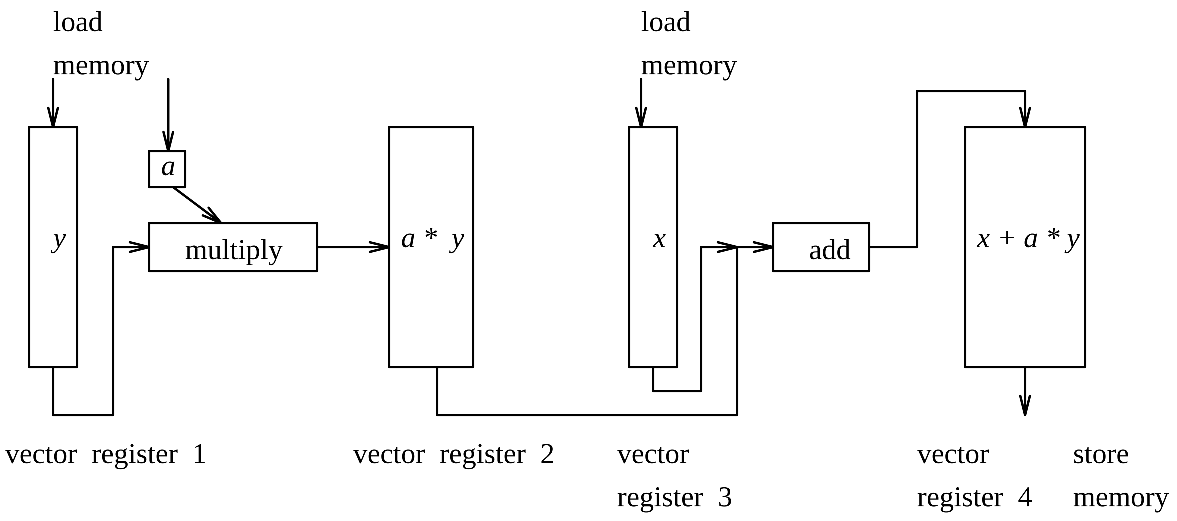


Figure 15: The flow of operands by a vector update

flops (+ and $*$) are done. Since vector registers are costly they can contain only a relatively small number of reals (e.g. 64 or 128 double precision numbers). This means that vectors too large to fit into the vector registers are broken into pieces that can be accommodated by the registers. After the first segment (or strip) is complete, the next one is started. This is done automatically by the compiler and is known as strip-mining. Because each break up costs overhead, strip-mining incurs a start-up overhead for each piece.

Memory banks

It seems that the problem is only shifted since the fast vector registers have to be filled from the "slow" main memory. To solve this problem the main memory is divided in to several different parts, which are called memory banks (memory interleaving). Suppose we have a computer where the load operation of one real costs 8 clock cycles (this is called the bank cycle time). The minimum number of memory banks for this machine is 8 in order to keep the functional units busy. The storing pattern of the vector a, used in $c(i) = a(i) + b(i)$ is shown in Figure 16. If the vector is loaded the load of $a(1)$ is started in the first clock cycle, the load of $a(2)$ in the second clock cycle until $a(8)$. The real number $a(9)$ is again in the first memory bank, however since there are 8 clock cycles passed the operation to load $a(9)$ can be started in the 9th clock cycle. So again after some start up time every clock cycle one real is loaded from main memory to a vector register.

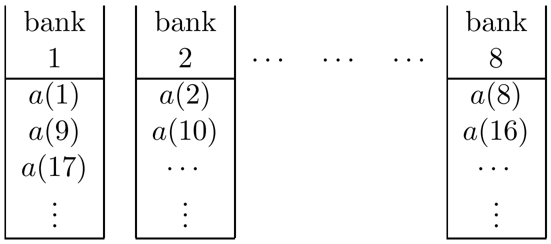


Figure 16: Storage pattern of a in the banks of the main memory

Memory bank conflicts

If the following loop must be carried out

```
for $i = 1, ..., n$ do
    $c(2 * i) = a(2 * i) + b(2 * i)$
end for
```

we see that $a(10)$ is needed after 4 cycles from the second memory bank. However this bank is busy loading $a(2)$ and the load of $a(10)$ has to wait 4 cycles. This is called a memory bank conflict. This can degrade the performance of the computer used considerably. In many computers the number of memory banks is larger than the bank cycle time (two to four times as large). Memory bank conflicts can also occur if one uses a two-dimensional array. Suppose a, b and c are two-dimensional arrays where the first dimension is equal to 2 and the second one equal to n. The loop

```
for $i = 1, ..., n$ do
    $c(1, i) = a(1, i) + b(1, i)$
end for
```

leads again to memory bank conflicts. This can be explained by the fact that two-dimensional arrays are always stored (FORTRAN) column-wise (see Figure 17). This problem is easily repaired by changing the order of the arrays: so $a(1 : 2\ ,\ 1 : n)$ is replaced by $a(1 : n\ ,\ 1 : 2)$

```
for $i = 1, ..., n$ do
    $c(i, 1) = a(i, 1) + b(i, 1)$
end for
```

and no memory bank conflicts occur. It is a good idea to have a loop in the first dimension. In the literature the quantity stride is introduced to describe the above given problems. The stride of a stored floating point vector is the distance (in memory locations) between the vector components. For a unit stride the vector components are stored contiguously.

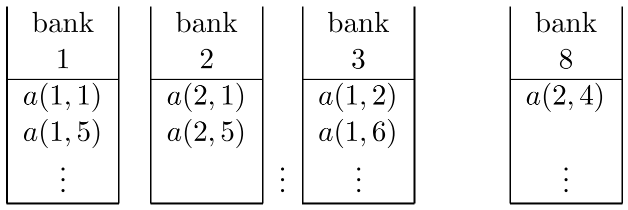


Figure 17: Storage pattern of a two-dimensional array $a(1:2\ ,\ 1:n)$

Cache
In some computers there is an additional level in the memory hierarchy: the cache. The cache is a part of the memory between the main memory and the vector registers (see Figure 18). The cache is much larger than a vector register, but much smaller than main memory.

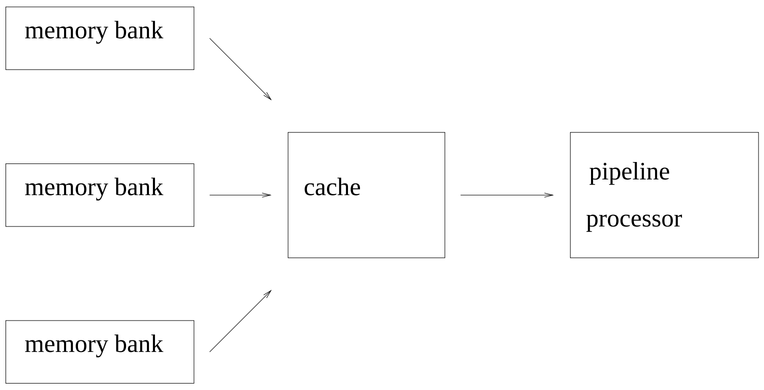


Figure 18: A pipelined processor with memory interleaving and cache

Data transfer between cache and vector registers is fast but transfer between cache and main memory is slow. In many iterative algorithms this deteriorates the performance considerably. In some algorithms (direct methods) part of the problem can be stored in the cache, then several flops ($\gg 10$) are done per point and then a new part is loaded. These ideas are used in BLAS3 routines and LAPACK solvers, which lead to favorable Mflop rates for these computers. The cache is not only used in supercomputers, but also in present day workstations and PC's.

Expensive operations
There are several operations which are more expensive than the standard operations $+, -$, and $*$. The first one is the division operation. On many vector-computers it takes at least two clock cycles to compute a division of a vector element by a given constant a. So if possible, it is better to avoid divide operations. One possibility for the given example is to calculate $\beta = 1/\alpha$ and multiply all vector elements by β. Note that both the division by α and the multiplication by β leads to fully vectorizable code, whereas the first operation costs two times as much as the second operation. However, the multiplication by β can more sensitive to rounding errors than the division by α.

Other operations which are hard to vectorize are recurrences. The following algorithm is used to solve a bi-diagonal lower triangular system of equations $Ax = b$, where A has elements a_{ij}:

```
for $i = 1, ..., n$ do
    $x(i) = (b(i) - a_{i,i-1} * x(i-1))/a_{ii}$
end for
```

Note that $x(i)$ can only be calculated if $x(i-1)$ is already known. So it is impossible to do this in a vectorized way. This implies that this loop runs in scalar speed instead of vector speed.

Amdahl's law
In a large problem there are parts which runs in scalar speed and parts which run in vector speed. Suppose that the algorithm considered costs n flops. A part pn of the flops where $p \in [0,1]$ is done in vector speed v (Mflops), and the remaining part $(1-p)n$ is done in scalar speed s (Mflops). The total time to perform the computations is:

$$t = \frac{pn}{v} + \frac{(1-p)n}{s} \tag{69}$$

and the speed R is given by

$$R = \frac{n}{t} = \frac{1}{\frac{p}{v} + \frac{(1-p)}{s}} \ . \tag{70}$$

This is known as Amdahl's law [1]. It easily follows from (70) that

$$R \leq \frac{s}{1-p} \ . \tag{71}$$

So if the fraction p which runs in vector speed is small, for instance $p = \frac{1}{2}$, it has no sense to use a computer with a slow scalar speed and a fast vector speed. Suppose $v = 130$ Mflops and $s = 4$ Mflops [21]; Section 4.1.1 (these are realistic numbers for the Cray 1) then for $p = \frac{1}{2}$, $R \leq 2 \cdot s = 8$ Mflops which is very disappointing in comparison with the very fast vector speed.

Compiler directives
Sometimes compilers cannot vectorize a loop which is vectorizable. As an example consider the copy of the array x to a shifted array. The following loop can achieve this:

```
ishift = 1
for $i = 1, ..., n$-ishift do
    $x(i) = x(i + \text{ishift})$ .
end for
```

Many compilers see this as a recurrence and it runs in scalar speed. However, it is only a copy and $x(i)$ can be copied without waiting (provided the sequence is kept). Many compilers enable the user to force vectorization of such a loop by compiler directives. A drawback is that they are machine-dependent so code tuned to a certain vector machine has to be adapted if it is run on another machine.

7.2 ICCG and vector computers

In this section we shall consider the various parts of the preconditioned CG algorithm with respect to its vectorization properties. The main parts of ICCG consist of vector updates ($y = x + a * y$), inner products (and norms), matrix vector multiplications and the solution of lower and upper triangular systems of equations. In the remainder of this section we discuss these operations.

Vector update

This operation is the easiest to vectorize. Since an add and multiply operation are used they can be linked together. However, it is useful to look into more detail to this operation, because it illustrates a difference between types of vector computers. Some vector computers like Cray 1, and Convex can only load or store one vector from main memory to a vector register. For the vector update $y = y + a * x$ this means that first the vector y is loaded, then vector x is loaded and concurrently it is multiplied by a and summed with y. Thereafter y is stored in main memory. The total time is equal to $3n/v$ and $2n$ flops are done so the speed is equal to $\frac{2}{3}v$ Mflops.

On other machines like the Cray Y-MP it is possible to load two vectors and store one vector concurrently. For such a computer the total time is n/v and the speed is $\frac{2nv}{n} = 2v$ Mflops. This is much better than the speed of the foregoing machines.

Inner product

The inner product can be computed by the following loop

```
$dot = 0$
for $i = 1, ..., n$ do
    $dot = dot + x(i) * y(i)$
end for
```

This looks like a recurrence since the next *dot* can only be calculated if the current *dot* is known. It is possible to avoid this. To do this one reserves one vector-register, which is denoted by *helpdot*. The inner product can now be calculated as follows (where we suppose that n is a multiple of 64):

```
for $i = 1, ..., 64$ do
    $helpdot(i) = 0$
end for
for $j = 1, ..., n/64$ do
    for $i = 1, ..., 64$ do
        $helpdot(i) = helpdot(i) + x((j-1) * 64 + i) * y((j-1) * 64 + i)$
    end for
end for
$dot = 0$
for $i = 1, ..., 64$ do
    $dot = dot + helpdot(i)$
end for
```

Note that a large part of the calculation can be done in vector speed. It is not necessary to do this explicitly if the inner product is calculated in a for the compiler clear way or by a call to a BLAS routine. In this case the compiler replaces the original code by optimized code. In general the start up time for an inner product is larger than for a vector update. When only one load or store is possible the asymptotic rate of performance is v Mflops which is 30% higher than the vector update. The reason for this is that the result of an inner product is a number so no storing of a result vector is necessary.

Matrix vector products
If the matrix is structured the matrix vector product has good vectorization properties. Suppose the matrix has 5 non-zero diagonals, then the product can be computed by:

```
for $i = 1, ..., n$ do
    $y(i) = 0$
end for
for $j = 1, ..., 5$ do
    for $i = 1, ..., n$ do
        $y(i) = a(i,j) * x(i + ishift(j)) + y(i)$                (72)
    end for
end for
```

or

```
for $i = 1, ..., n$ do
    $y(i) = a(i,1) * x(i + ishift(1)) + a(i,2) * x(i + ishift(2))$                (73)
        $+a(i,3) * x(i + ishift(3)) + a(i,4) * x(i + ishift(4))$
        $+a(i,5) * x(i + ishift(5))$

end for
```

Especially for machines with one load/store operation loop (73) is much faster than loop (72) because all intermediate results in (73) are kept in vector registers, whereas in (72) all intermediate results are loaded and stored from main memory.

For general unstructured matrices the matrix vector product is harder to vectorize. First the number of nonzero elements per row may be different for different rows. Secondly the shift also depends on the row number. Suppose the number of non zero elements per row is fixed and equal to 5, the non zero elements are stored in a two dimensional matrix $a(1:n,\ 1:5)$ (row i is stored in $a(i,\ 1:5)$) and the column number of each element is stored in $col\ (1:n,\ 1:5)$. Then the matrix vector product can be calculated by

```
for $i = 1, ..., n$ do
    $y(i) = 0$
end for
for $j = 1, ..., 5$ do
    for $i = 1, ..., n$ do
        $y(i) = y(i) + a(i,j) * x(col(i,j))$
    end for
end for
```

This means that $x(col(i,j))$ is not a contiguous vector. This is called indirect addressing and can reduce the speed considerably (memory bank conflicts can occur etc.). At least the asymptotic rate of performance is reduced by a factor 2 with respect to direct addressing. (compare [21]: Section 4.4).

Preconditioning
With preconditioning one has to solve an upper or a lower triangular matrix. This leads in general to reccurences which are not easily vectorizable. For the model problem the lower

triangular matrix L is given by (compare (29))

$$L = \begin{bmatrix} d_1 & & & & \\ b_1 & d_2 & & & \\ & \ddots & \ddots & & \\ c_1 & & b_m & d_{m+1} & \\ & \ddots & & \ddots & \ddots \end{bmatrix} \tag{74}$$

where m is the number of grid points in the x-direction. Solving the linear system

$$Ly = x$$

can be done by the following loop (75)

```
for $i = 1, ..., n$ do
   $y(i) = (x(i) - b_{i-1} * y(i-1) - c_{i-m} * y(i-m))/d_i$
end for
```

This contains a recurrence and is not vectorizable. There are several ideas to change this loop to faster versions, we describe two of them: partial vectorization and diagonal ordering.

Partial vectorization
For more details and other ideas we refer to [3] and [74]. To explain partial vectorization we consider the grid used for this problem. It can be easily seen that in order to calculate $y(i)$ the value of $y(i-1)$ and $y(i-m)$ should be known. For the partial vectorization we rearrange the order of computation. We first calculate all $y(i)$ on line 1, then on line 2 etc. This implies

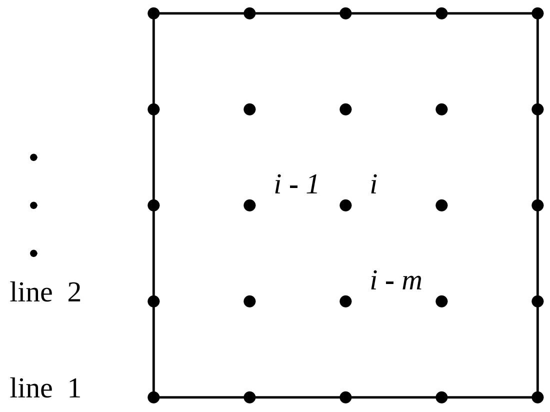


Figure 19: Partial vectorization

that if $y(i)$ on line 2 is calculated the values of $y(i-m)$ are on the foregoing line and hence known. This means that along line 2 this part can be done in vector speed. This leads to the following loops:

```
         for $j = 1, ..., n/m$ do                         number of lines
c$dir force vector                                        compiler directive
            for $k = 1, ..., m$ do
               $i = (j-1) * m + k$                        vector part
               $y(i) = x(i) - c_{i-m} * y(i-m)$
```

```
        end for
        for $k = 1, ..., m$ do
            $i = (j - 1) * m + k$                                    scalar part
            $y(i) = (y(i) - b_{i-1} * y(i - 1))/d_i$
        end for
    end for
```

A compiler directive is necessary to force vectorization of the first loop, since it looks to the compiler as a recurrence. Drawback of this approach is that only a small part of the algorithm is vectorized, and the vector-length is equal to m, which is in general much less than n.

Diagonal ordering
In the diagonal ordering the numbering is rearranged in the following way (see Figure 20). Now we first calculate $y(i)$ on diagonal 1 then on diagonal 2 etc. If $y(i)$ is calculated at

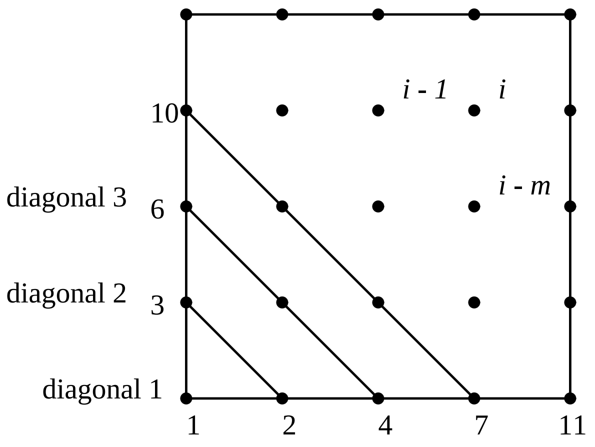


Figure 20: Diagonal reordering

diagonal j all the values $y(i - 1)$, and $y(i - m)$ are on diagonal $j - 1$ so they are already known. This means that all the computations can be done in vector speed. A drawback is again the small vector length especially for the first and final diagonals. In general for this problem the diagonal ordering is faster than the partial vectorization (on Convex C3840), but harder to program. For other stencils partial vectorization can be better than diagonal ordering.

7.3 Exercises

1. Explain why loop (73) can be faster than loop (72).

2. Make a comparison (theoretical) of the required time to perform loop (75) with the time to perform partial vectorization and diagonal ordering.

3. Given a scalar speed s and a vector speed v.

   (a) Compute the time to solve $Ly = x$ with loop (75).

   (b) Suppose that $L = I - B$, where $\|B\| < 1$. Furthermore, a Neumann series of three terms $I + B + B^2$ is a good approximation of L^{-1}. Compute the required time using the Neumann series.

   (c) Compare both approaches.

4. (a) Try to compute the Mflop rate of your computer using the loop

```
for $i = 1, \ldots, n$ do
    $c(i) = c(i) + a * b(i)$
end for
```

   (b) Do the same for the loop

```
for $i = 1, \ldots, n$ do
    $c(i) = c(i) + a * b(i * k)$
end for
```

   for various values of k ($k = 2, 8, 64, \ldots$).

   (c) Do you see any cache effects?

8 Parallel computers

A good introduction to parallel computing is given in [49]. Topics in parallel numerical algebra are discussed in [20]. Until now the algorithms considered are sequential which means that a new task (subroutine) is started if the foregoing task is completed. On parallel machine tasks are done concurrently. This enables one to speed up the rate of performance considerably. Furthermore, in many applications memory requirements are the bottleneck. Using parallel machines it may be easier to obtain enough memory depending on the type of architecture. However, there are drawbacks. Only recently fast, general purpose parallel computers are available. To program them such that the performance is in the vicinity of the peak performance is a tedious task. It leads to totally new methods or old methods which have been discarded on sequential machines, but have good parallelization properties. In the future some of the work will be done by the compiler, however this takes several years and it is impossible to make a good parallel program from every sequential program.

8.1 Introduction

In this section we give an introduction on parallel computing.

Flynn's categories
A much referenced classification of computer architectures was given in [29]. In this paper, computers are divided into four categories. An instruction stream consists of subroutines, statements etc., whereas a data stream consists of numbers, for instance the contents of a matrix etc.

SISD - Single Instruction stream, Single Data stream;
SIMD - Single Instruction stream, Multiple Data stream
MISD - Multiple Instruction stream, Single Data stream
MIMD - Multiple Instruction stream, Multiple Data stream

Classical sequential computers are examples of the SISD category. An array of processors with a central controller is an SIMD computer. MISD machines are seldomly built, some special-purpose machines execute different instructions on the same data. Finally, the most general form are MIMD computers. In such a computer different instructions can be executed on different parts of data. See [49] p.17, 18 for a comparison of SIMD and MIMD computers. Some machines consists of a mix of processors belonging to different categories.

Memory organization
Shared memory: each processor has access to a common shared memory. Such a machine is easy to use because all data is available on every processor. However if two processes read and write the same part of data, the order of these processes is important. A drawback of shared-memory parallel computers is that a fast interconnection network is needed in order to obtain good performance. To enhance the performance some shared-memory computers use local and global memory. In many applications this leads to a considerable decrease in the amount of memory references, which use the interconnection network. If the time to access any memory word is identical the computer is called a uniform memory access (UMA) computer. On a non-uniform memory access (NUMA) computer the time to access remote memory is longer than the time to access local memory. When the processors use a cache

there is the problem that a shared variable is only adapted in one of the caches, which leads to incorrect results. This implies that software should be available to solve this problem (cache coherence).

Distributed memory: a network of processors each with its own local memory. The processors need to communicate by sending and receiving messages to access memory on another processor. In this memory organization, two models of computations are possible: the synchronous "systolic" model and the a-synchronous message-passing model. In the systolic model the processors pace their computations and communications according to the tick of a global clock. In a message-passing computer, the processors coordinate their activities on the basis of received messages. There is no global clock. A message-passing computer looks like a NUMA computer. The difference between them is that a remote memory access in a message-passing computer is done by explicit messages, whereas this is done by the hardware/software on a NUMA computer. Mixes of memory organizations are also proposed, for instance in the concept of a virtual-shared memory system [51]: the distributed memory system (hardware) looks like a shared memory system (done by software) to the user.

Static interconnection network
In order to communicate between processors they have to be interconnected. The easiest way (from user point) would be that every processor is connected to every other processor. However, this is not possible for a high number of processors since the number of connections increases quadratically. So in most parallel computers every processor (node) is only connected to a limited number of other processors. A drawback is that if there is no direct link between processors then two or more messages are necessary to get the data at the right place. There are many interconnection schemes, for instance the hypercube (is optimal in some sense), the tree (divide and conquer procedures) and the torus. The torus seems a good scheme to be used for *pde* problems. Below we sketch the 1D torus (or ring) and the 2D torus. Every rectangle is a processor and every line is a connection. Looking at the block structure

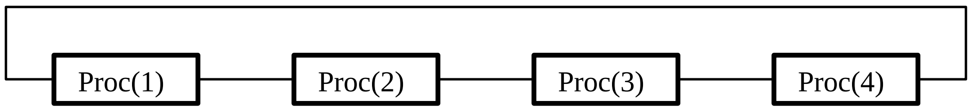


Figure 21: 1D torus: all neighbors (in 1D) and the end nodes are connected

of the model problem (see Section 8.3) one sees that the 2D torus seems a good choice for a block structured program because the only data transfer is between neighboring nodes. For a three dimensional *pde* problem a 3D torus seems to be the best. For more details and criteria to evaluate static interconnection networks we refer to [49] Section 2.4.

Communication
In this paragraph we only consider a distributed memory system. As we have already remarked, information from one processor which is needed on another processor leads to communication. Depending on the machine this may be done by the hardware when the program refers to non-local data, or it may require explicit sending and/or receiving of messages on the part of the programmer. A very simple model for the time it takes to move n data items from one location to another is $\alpha + \beta n$, with $\beta, \alpha > 0$. The quantity α is the start-up time of the operation; another term for this is latency. The incremental time per data item moved

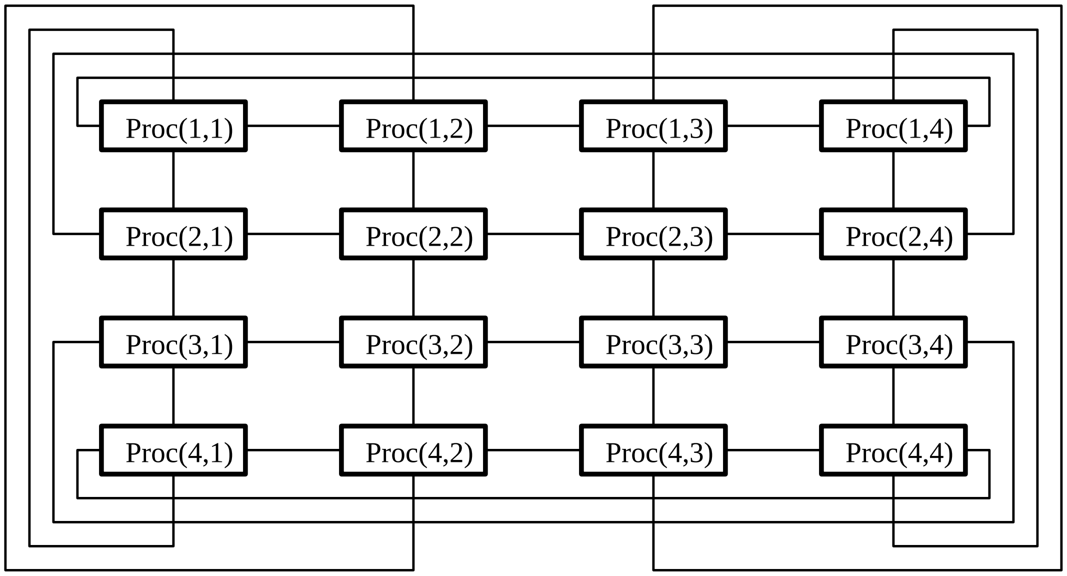


Figure 22: 2D torus: all neighbors (in 2D) and the end nodes in every line and column are connected

is β; its reciprocal is called bandwidth (see [49] Section 2.7). In general $\beta \ll \alpha$, it takes a relatively long time to start-up a message, after which data items arrive at a higher rate (this is comparable with the start-up time of a vector operation). Sometimes communication and computation can overlap. This saves wall-clock time. Wall-clock time is the real time one has to wait between the start and the termination of a program.

A special kind of communication is synchronization. This appears if two or more processors have to wait until a certain stage is reached. This costs time since one or more processors are idle part of the time.

Granularity
An important aspect of a parallel algorithm is its granularity. This qualitative term refers to the amount of computation that takes place between synchronization points (messages). We think that for *pde* problems coarse grained parallelism (block structure, domain decomposition) is easier to use than fine grained parallelism. With coarse grained parallelism it is possible that the amount of communication is an order of magnitude less then the amount of computation. This is important because communication takes in general more time than the same amount of computation. Which algorithm is preferred is also motivated by the computer used. A coarse grain algorithm is suitable for a cluster of workstations where communication is much slower than computation, but it can also be used on a Cray T3E where communication and computation performance are close together. However using a fine grain algorithm on a cluster of workstations leads to a disappointing performance.

Load balancing
Note that processes on different processors consist of a different amount of work. So some processors may be ready while others are busy with computations. We call a parallel computation load balanced if each processor has roughly the same amount of work to perform. This is a difficult problem.

Speed up and scalability
Finally one wants to quantify the speed up from the sequential program to the parallel program ([37] Section 6.1.1 and [49] Chapter 4). A fair comparison is the fastest sequential program on a sequential machine (possible 1 node of a parallel machine) to the fastest parallel program (in general different from the sequential one) on p parallel processors. The theoretical value, a speedup with a factor p, is very unlikely in practice. The title of [7] Bailey's paper: "Twelve ways to fool the masses when giving performance results on parallel computers", shows that the measurement of the speed up factor is a tricky business.

Algorithms should be scalable, which means that they remain efficient as they run on larger problems and larger machines (see [49] Section 4.4). As problems (machines) grow, it is desirable to avoid algorithm redesign.

Two models to study the accelerating effect of parallel processing are known: Amdahl's law and Gustafson's law.

Amdahl's law: Assume that a process consists of basic operations all carried out with the same computational speed and that a fraction f of these operations can be carried out in parallel. The parallel part costs $\frac{ft_1}{p}$ time and the serial part $(1-f)t_1$. The speedup on p processors is equal to

$$S_p = \frac{t_1}{t_p} = \frac{t_1}{\frac{t_1}{p}(f + (1-f)p)} = \frac{p}{f + (1-f)p} < \frac{1}{1-f}. \tag{76}$$

Such a process is not scalable because $S_p < \frac{1}{1-f}$ so if p is large (with respect to $\frac{1}{1-f}$) no gain is obtained to include more processors.

Gustafson's law: In Amdahl's law one assumes that the portion of the code that can be parallelized remains constant as the problem size increases. As Gustafson [41] pointed out, this is often not appropriate. In Gustafson's model it is supposed that a fraction g of the total wall-clock time t_p is done in parallel, whereas $1-g$ is the fraction of the time used for serial work. The time for a single processor to do the same job would be $(1-g+gp)t_p$. The speedup is given by

$$S = \frac{(1-g+gp)t_p}{t_p} = 1 - g + gp \tag{77}$$

and we see that for p large $S = gp$. Such a job is scalable. It can be seen that the amount of work in parallel mode increases if the number of processors p increases. This means that f used in Amdahl's law depends on p and $f \to 1$ if $p \to \infty$.

Below we give some classes of parallel machines and discuss the advantages and disadvantages from our point of view. We shall give some computers as examples, it is not the intend to be complete.

Vector-computers with multiple CPU's
Most of the present-day supercomputers have more than one central processor unit (CPU). They are shared memory machines and the number of CPU's is limited (less than 16). An advantage is that the CPU's are very fast and easy to program, however they are only useful (as a parallel machine) if they are used in dedicated mode. This means that only one user is active on the machine. This is only possible in very special cases. Otherwise only one CPU

with timesharing is used per process. Examples of these machines are: Cray Y-MP and NEC SX.

Massive Parallel Processing (slow processors)
In these machines a high number (up to 65,536) of simple (slow) processors are used. A typical example of such a machine is the Connection Machine (CM5) built by Thinking Machines Corporation. These machines can give very fast results for very well parallelized code. However, in general they are hard to program to give results in optimized code. As an example we refer to [20]; p. 130 where it is stated that the matrix multiplication subroutine in the CM-2 Scientific Subroutine Library took approximately 10 person-years of effort. Furthermore, in general the code is not (or not easily) portable to other MPP machines. We think that for 3D PDE problems the type of machines presented in the foregoing paragraph (Cray, NEC) are faster than MPP machines (CM-2). As an illustration of this we refer to [80].

Massive Parallel Processing (fast processors)
In this type of machines a moderate number of computers (workstations) are connected. These machines have a distributed memory and the node computers are very fast (much faster than that of the CM5). Until now the number of processors is in the range (1-1000). Some examples are[2]: Fujitsu VPP, Hitachi SR, HP Exemplar, IBM SP2, and SGI/Cray T3E. Besides these machines, there are software packages which can be used to make a cluster of workstations that can be used as a parallel machine. These packages are discussed in the next section.

Beowulf cluster
A Beowulf cluster is not a particular product. It is a concept for clustering varying number of small, relatively inexpensive computers running the Linux operating system and using MPI or PVM for message passing. The goal of Beowulf clustering is to create a parallel processing supercomputer environment at a price well below that of conventional supercomputers.

A Beowulf cluster is ideal for tackling very complex problems that can be split up and run simultaneously in separate computers. And that's a key point: not every problem can be approached in parallel so not every problem will benefit from the Beowulf approach. In the taxonomy of parallel computers, Beowulf clusters fall somewhere between MPP (Massively Parallel Processors, like the nCube, CM5, Convex SPP, Cray T3D, Cray T3E, etc.) and NOWs (Networks of Workstations).

Beowulf clusters benefit from developments in both these classes of architecture. MPPs are typically larger and have a lower latency interconnect network than an Beowulf cluster. Programmers are still required to worry about locality, load balancing, granularity, and communication overheads in order to obtain the best performance. Even on shared memory machines, many programmers develop their programs in a message passing style. Programs that do not require fine-grain computation and communication can usually be ported and run effectively on Beowulf clusters.

A Beowulf class cluster computer is distinguished from a Network of Workstations by several subtle but significant characteristics. First, the nodes in the cluster are dedicated to the cluster. This helps ease load balancing problems, because the performance of individual nodes

[2]an up to date list of high-performance computers is given at: http://www.top500.org/

are not subject to external factors. Also, since the interconnection network is isolated from the external network, the network load is determined only by the application being run on the cluster. This eases the problems associated with unpredictable latency in NOWs. All the nodes in the cluster are within the administrative jurisdiction of the cluster. For examples, the interconnection network for the cluster is not visible from the outside world so the only authentication needed between processors is for system integrity. On a NOW, one must be concerned about network security. Finally, operating system parameters can be tuned to improve performance. For example, a workstation should be tuned to provide the best interactive feel (instantaneous responses, short buffers, etc), but in cluster the nodes can be tuned to provide better throughput for coarser-grain jobs because they are not interacting directly with users.

8.2 Software for parallel computing

In this section we give a concise description of software which can be used to implement parallel algorithms. We start with a description of software used on message passing machines (MPI, PVM). Thereafter a simpler message passing model is considered: BSP. Finally a short description of High Performance Fortran is given.

8.2.1 Message Passing Interface (MPI)

MPI[3] is an emerging and widely accepted standard for developing parallel programs with messages [40]. It can be used on distributed memory computers and networks of workstations. It can also be used on shared memory machines. MPI can be used in C and Fortran programs. Since MPI is a standard source code is portable. MPI contains a rich set of routines, yet most programs can run using a handful of the routines. Primarily routines are used to:

- start processes
- send messages
- receive messages
- synchronize

For communication the user inserts communication (MPI) calls into the program explicitly. Point-to-point (processor to processor) or global (one to all, all to one) communication is possible. MPI uses the SPMD (single program, multiple data) programming model. That implies that there is only one program, which runs on all processors. However it is possible that different processors execute different parts of the program. MPI consist of public domain software and is available on most MPP machines. It can also be used on a cluster of workstations. So spare time on workstations can be used to run a process in parallel without extra costs. Furthermore, one can get easily acquainted with parallel computing.

Another (older) package with approximately the same functionality is PVM (Parallel Virtual Machine).

[3]http://www.mcs.anl.gov/mpi/index.html

8.2.2 OpenMP

What is OpenMP[4]? OpenMP is a set of compiler directives and callable runtime library routines that extend Fortran (and separately, C and C++) to express shared memory parallelism [16]. The base language is left unspecified and vendors may implement OpenMP in any Fortran compiler. Naturally, Fortan 90 and Fortran 95 require the OpenMP implementation to include additional semantics over Fortran 77 in order to support pointers and allocatables.

The language extensions fall into one of three categories: control structures, data environment, and synchronization. The standard also includes a callable runtime library with accompanying environment variables.

A Simple Example
Below a simple code example is given for computing π using OpenMP. This example is meant only to illustrate how a simple loop may be parallelized in a shared memory programming model.

```
      program compute_pi
      integer n, i
      double precision w, r, sum, pi, f, a
c   function to integrate
      f(a) = 4d0 / (1d0 + a*a)
c   number of integration intervals
      n = 100
c   calculate the interval size
      w = 1d0/n
      sum = 0d0
!$OPM PARALLEL DO PRIVATE(x), SHARED(w)
!$OPM& REDUCTION(+: sum)
      do i = 1, n
         x = w * (i - 0.5d0)
         sum = sum + f(x)
      enddo
      pi = w * sum
      end
```

Program execution begins as a single process. This initial process executes serially and we can set up our problem in a standard sequential manner, reading and writing stdout as necessary. When we first encounter a PARALLEL construct, in this case a PARALLEL DO, a team of one or more processes is formed, and the data environment for each team member is created. The data environment here consists of one PRIVATE variable, x, one REDUCTION variable, sum, and one SHARED variable, w. All references to x and sum inside the parallel region address are private, non-shared, copies. The REDUCTION attribute takes an operator, such that at the end of the parallel region the private copies are reduced to the master copy using the specified operator. All references to w in the parallel region address the single master copy. The loop index variable, i, is PRIVATE by default. The compiler takes care of assigning the appropriate iterations to the individual team members, so in parallelizing this loop you

[4]http://www.openmp.org/

don't even need to know how many processors you will run it on.

Within the parallel region there may be additional control and synchronization constructs, but there are none in this simple example. The parallel region here terminates with the END DO which has an implied barrier. On exit of the parallel region, the initial process resumes execution using its updated data environment. In this case the only change to the master's data environment is in the reduced value of sum.

This model of execution is referred to as the fork/join model. Throughout the course of a program, the initial process may fork and join a number of times. The fork/join execution model makes it easy to get loop level parallelism out of a sequential program. Unlike in message passing, where the program must be completely decomposed for parallel execution, in a shared memory model it is possible to parallelize just at the loop level without decomposing the data structures. Given a working sequential program, it becomes fairly straightforward to parallelize individual loops in an incremental fashion and thereby immediately realize the performance advantages of a multiprocessor system.

8.2.3 The Bulk Synchronous Parallel (BSP) model

The essence of the BSP[5] approach to parallel programming is the notion of the superstep, in which communication and synchronization are completely decoupled. A BSP program is simply one which proceeds in phases, with the necessary global communications taking place between the phases. This approach to parallel programming is applicable to all kinds of parallel architecture: distributed memory architectures, shared memory multiprocessors, and networks of workstations. It provides a consistent, and very general, framework within which to develop portable software for scalable computing.

A step is defined as a basic operation on locally held data values. A BSP computation consists of a sequence of parallel supersteps, where each superstep is a sequence of steps carried out on local data, followed by a barrier synchronization at which point any non-local data accesses take effect. Requests for non-local data, or to update non-local data locations, can be made during a superstep but are not guaranteed to have completed until the synchronization at superstep end. Such requests are non-blocking; they do not hold up computation.

BSP Programming
The programmer's view of the computer is that it has a large, globally accessible, memory. The division between data held locally and data held on remote processors is consistent with modern non-uniform memory architecture (NUMA) systems which often exhibit a two level memory access time characteristic. This includes virtual shared memory machines with all types of coherent or non-coherent cache structures. This type of architecture is expected to dominate because it is perceived to be simpler to program.

To achieve scalability it will be necessary to organize the calculation in such a way as to obviate the bad effects of large latencies in the communications network. In some cases it will be necessary to select different algorithms for different communications networks.

By separating the computation on local data from the business of transferring shared data,

[5]http://www.bsp-worldwide.org

which will be handled by lower level software, we can ensure that the same computational code will be able to run on different hardware architectures from networked workstations to genuinely shared memory systems.

The superstep structure of BSP programs lends itself to optimization of the data transfers. All transfers in a superstep between a given pair of processors can be consolidated to form larger messages that can be sent with lower (latency) overheads and so as to avoid network contention. The lower level communications software should also exploit the most efficient communication mechanisms available on the actual hardware. Since this software is application-independent, the cost of achieving the efficiency can be spread over many applications and is an acceptable one.

Cost Modeling the calculation
Another advantage of the simple structure of BSP programs is that the modeling of their performance is much easier than for message passing systems, for example. In place of the random pair-wise synchronization that characterizes message passing, the superstep structure in BSP programs makes it relatively easy to derive cost models (i.e. formulae that give estimates for the total number of steps needed to carry out a parallel calculation, including allowance for the communications involved).

Availability
BSP is available on a number of MPP's. The use of *shmemm* routines on the Cray T3E machine can be seen as a special version of the BSP model.

8.2.4 High Performance Fortran (HPF)

High Performance Fortran (HPF[6]) is the result of efforts towards the standardization of a Fortran language suitable for the latest generation of high performance machines. It is a set of constructs and extensions to Fortran90 and allows the user to express parallelism in a relatively simple manner. The main aims of such a standard are to promote the wider use of parallelism by hiding the details of the underlying architecture from the programmer and to provide a code which is easily portable and non-machine specific.

Data Parallel Programming
The idea behind the data parallel programming paradigm is the support of whole array operations executed in parallel. Typically a single program controls the distribution of, and operations on the data on all processors. The languages used to program this vary from standard Fortran or C, with language extensions to deal with the parallelism, to specialized data parallel languages based on one machine. In most cases the actual distribution of data and communication between processors is done by the compiler, with guidance from the programmer.

Data Parallel Building Blocks
We describe the data parallel paradigm via the use of High Performance Fortran. Data parallel programming is concerned with defining collective operations on arrays or sets of array elements, with these arrays distributed over a number of processors. If an algorithm can be

[6]most of this section is copied from the Edinburgh Parallel Computing Centre (EPCC) HPF course: http://www.epcc.ed.ac.uk/computing/training/

expressed in terms of such operations then it is likely that a data parallel implementation will be efficient. Grid based algorithms are one good example. It is helpful to categorise the set of operations that form the basis for the implementation of data parallel algorithms. These are: control over data layout, whole array operations, array sections, conditional operations, reduction operations, shift operations, scan operations, and generalized communications. We will now consider each of these in more detail.

Controlling Data Layout
In many cases it is crucial that the user has control over the placement of data on the processors. The goals are to minimize communication between processors, keep all processors busy and to carry out operations in parallel to obtain highest performance. Data layout is normally controlled by language constructs or directives. Figure 23 shows two possible data layouts for a two dimensional array. The shaded portion indicates array elements and how the data could be distributed across processors P1, P2, P3, P4.

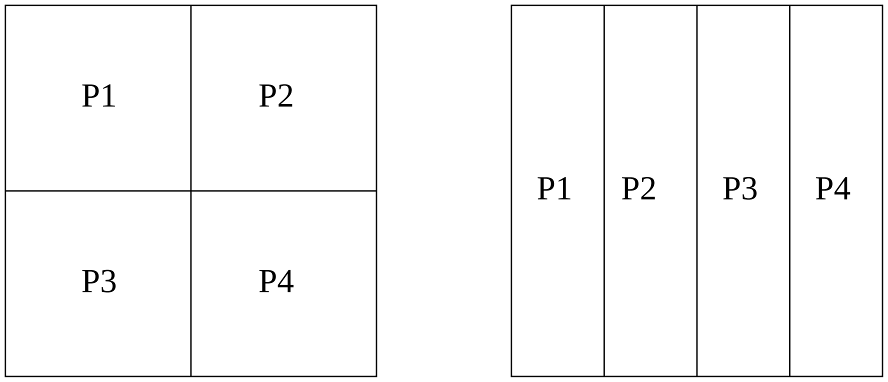


Figure 23: Possible data layouts on 4 processors

Whole Array Operations
Whole array operations would take as arguments whole arrays and apply the operation to every individual element of that array. The operation, such as sum, multiply and divide, is applied to each element of the array, possibly in parallel. Figure 24 shows the multiplication of two arrays. All the elements can be multiplied in parallel given a sufficient number of processors. An implementation should support array expressions and also extend the standard

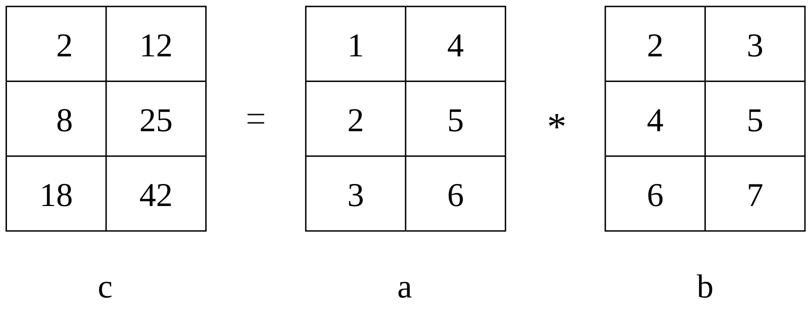


Figure 24: Whole array operation: multiplication of two arrays

mathematical functions (sin, cos etc.) to operate on array-valued arguments, applying the operation to every array element and returning an array of results. The intent of supplying such whole array operations is to enable the production of clearer code and to reduce the likelihood of mistakes.

Array Sections
There should be a method to access sections of an array. This would allow the programmer to specify regular sections of array on which to act. Figure 25 shows selection of a column of an array and selection of the interior elements of an array. This would be useful for, say

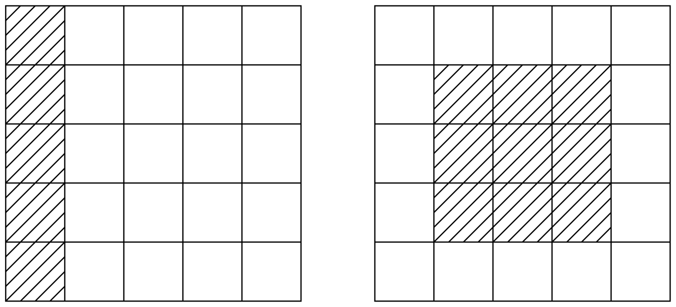


Figure 25: Array sections

multiplying matrices where whole rows and columns are multiplied, or the update of some grid based problem, where only the central array elements are of interest as opposed to the ”halo” around the edge.

Conditional Operations
Operations can be made to act on a subset of array elements, chosen subject to some conditional based (logical) mask or expression. This would allow the programmer to specify irregular sections of the array on which to act. For example, Figure 26 shows the selection of elements of a 7x7 element array, a shaded square indicating that the mask is set in this position. This could be used to adapt the grid problem and perform the update on the even

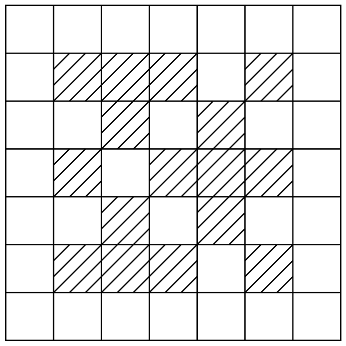


Figure 26: Conditional operations on arrays

and odd sites separately, or to apply an operation only to array elements which satisfy some condition, such as being not equal to zero.

Reduction Operations on Arrays
A reduction operation produces one result from the combination of many elements of an array.

Examples of possible reduction operations are:

- sum of elements in array,
- minimum or maximum values in the array,
- logical AND, OR, EOR,
- a count of the number of true elements in a logical array.

These operations are useful in control constructs where the logical flow of the program depends on some global property of an array. An example would be a converging iterative procedure applied to all elements of an array or array section. The iteration process could stop when all processors had achieved some tolerance, in other words when all values in some logical mask expression were true.

8.3 Iterative methods and parallel computers

In this section a number of parallel iterative methods are considered. We start with a coarse grain parallel method: domain decomposition with accurate solution of the subdomain problems. Thereafter some remarks are given on inaccurate solution of the subdomain problems (medium grain parallel). The section is concluded by a description of a number of fine grain parallel methods. For further reading we refer to [74], [49], [21], [20] and [8]; Section 4.4.

8.3.1 Domain decomposition (accurate subdomain solution)

In this subsection a domain decomposition algorithm is used to solve a system of equations in parallel. For more details we refer to [11].

The method
Consider the linear system

$$Ax = b. \tag{78}$$

We decompose A into blocks such that each block corresponds to all unknowns in a single subdomain. For two subdomains one obtains

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \tag{79}$$

where A_{11} and A_{22} represent the subdomain discretization matrices and A_{12} and A_{21} represent the coupling between subdomains. A domain decomposition iteration for (78) has the following form:

$$x^{i+1} = (I - M^{-1}A)x^i + M^{-1}b, \tag{80}$$

where M denotes a block Gauss-Seidel, or a block Gauss-Jacobi matrix:

$$M = \begin{pmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{pmatrix} \text{(Gauss-Seidel)}, M = \begin{pmatrix} A_{11} & 0 \\ 0 & A_{22} \end{pmatrix} \text{(Gauss-Jacobi)}.$$

Block Gauss-Seidel and block Gauss-Jacobi iterations are algebraic generalizations of the Schwarz domain decomposition algorithm. Similar to Schwarz domain decomposition, in each iteration, subdomains are solved using values from the neighboring block. For instance formula (80) for domain 1 becomes

$$x_1^{i+1} = x_1^i + A_{11}^{-1}(b_1 - A_{11}x_1^i - A_{12}x_2^i),$$

where x_2^i are the values from the neighboring block. The iterate update $\delta x_1^{i+1} = x_1^{i+1} - x_1^i$ is computed from

$$A_{11}\delta x_1^{i+1} = b_1 - A_{11}x_1^i - A_{12}x_2^i$$

by a direct method or an iterative solution method where the method is stopped after an accurate solution has been obtained.

When the subdomain problems are solved accurately the system

$$M^{-1}Ax = M^{-1}b \tag{81}$$

can be reduced to a system only involving unknowns near the interfaces. Suppose that the unknowns in the vicinity of the interfaces are denoted by x_r and the remaining ones by x_{nr}. When the components are ordered as $x = \begin{pmatrix} x_{nr} \\ x_r \end{pmatrix}$, system (81) has the form

$$M^{-1}Ax = \begin{pmatrix} I & R \\ 0 & D \end{pmatrix} \begin{pmatrix} x_{nr} \\ x_r \end{pmatrix} = \begin{pmatrix} g_{nr} \\ g_r \end{pmatrix},$$

which can be reduced to $Dx_r = g_r$ and $x_{nr} = g_{nr} - Rx_r$. System $Dx_r = g_r$ is much smaller than (81) and is known as the interface equations. The interface equations are solved by a Krylov subspace method (for instance GMRES). This implies that matrix vector products $y_r = Dx_r$ have to be calculated. To obtain this product a solution of the subdomain problems is needed (see [11], Section 3).

Parallel implementation
We consider a parallel implementation of the GMRES accelerated domain decomposition method. This means that the block Jacobi (additive Schwarz) algorithm is used. The parallel implementation is based on message passing using MPI or PVM. In the parallel computer (or cluster of workstations) each node is assigned certain subdomains to solve. The host program controls the acceleration of the domain decomposition algorithm using GMRES. Because of the reduction of (81) to a system concerning only the interface unknowns it is not necessary to parallelize GMRES. So the GMRES acceleration procedure is only executed on the host. Computation of the matrix vector product is performed in parallel.

Each node receives initial interface data (parts of the vector x_r^i), solves the assigned subdomain problems and sends the results (parts of the vector x_r^{i+1}) back to the host. Note that this programming model is not completely SPMD, because the host program has additional functions like acceleration and writing output.

The amount of computation to solve a subdomain problem accurately is much larger than the amount of communication. Therefore this is a coarse grain parallel method. For results on a cluster of workstations we refer to [11].

8.3.2 Domain decomposition (inaccurate subdomain solution)

In the previous subsection the subdomain problems are solved accurately. Since these problems have a similar nonzero structure as the original matrix, and since they may still be quite large, it reasonable to solve them using a second Krylov subspace iteration. A question which arises naturally, addresses the tolerance to which these inner iterations should converge. It seems senseless, for example, to solve the subdomain problems with a much smaller tolerance than is desired for the global solution. The influence of the accuracy of the subdomain solution on the convergence has been investigated in [12]. A large gain in CPU time has been observed on a sequential computer, when the subdomain accuracy is relatively low.

Note that if the subdomains are solved using a Krylov subspace method such as GMRES, then the approximate solution is a function of the right-hand side, which is the residual of the outer iteration. Furthermore, if the subdomain problems are solved to a tolerance, the number of inner iterations may vary from one subdomain to another, and in each outer iteration. The effective preconditioner is therefore a nonlinear operator and varies in each outer iteration. A variable preconditioner presents a problem for the GMRES method: namely the recurrence relation no longer holds. To allow the use of a variable preconditioner the GCR method is used as acceleration method.

Note that the GCR method is applied to the global matrix. So the GCR method should also be parallelized. For the details of this we refer to the following subsection. Depending on the required accuracy the method can be seen as a coarse grain (high accuracy), medium grain (medium accuracy) or fine grain (low accuracy) parallel method.

8.3.3 Parallel iterative methods

Many iterative methods (basic iterative methods, or preconditioned Krylov subspace methods) consist of 4 building blocks: vector updates, inner products, matrix vector products, and the solution of lower or upper triangular systems. First we give a description of the data distribution for our model problem. Thereafter the parallel implementation of the building blocks for the preconditioned GMRES method is considered. Parallel implementations of Krylov subspace methods are given in [68] and [49]. The large number of inner products used in the GMRES method can be a drawback on distributed memory computers, because an inner product needs global communication. In general the preconditioner is the most difficult part to parallelize.

Data distribution
On distributed memory machines it is important to keep information in local storage as much as possible. For this reason we assign storage space and update tasks as follows. The domain is subdivided into a regular grid of rectangular blocks. Each processor is responsible for all updates of variables associated with its block. An extra row of auxiliary points is added to the boundaries of a block to provide storage space for variables used in matrix-vector products and preconditioner construction.

Vector update
The vector update ($y = y + \alpha x$) is easy to parallelize. Suppose the vectors x and y are distributed. Once α is send to all processors, each processor can update its own segment independently.

Inner product
The work to compute an inner product is distributed as follows. First the inner product of the vector elements that reside on a processor is calculated for each processor. Thereafter these partial inner products are summed to form the full inner product. To obtain the full inner product global communication is required.

Communication time is the sum of start-up time (latency) and send time. On many parallel computers the send time is an order of magnitude less than the latency. For this reason it is attractive to combine communications as much as possible. Using the Classical Gram-Schmidt (CGS) method in the GMRES algorithm all inner products can be computed independently. So the communication steps can be clustered, which saves much start-up time. A drawback of CGS is that the resulting vectors may be not orthogonal due to rounding errors. Therefore, the Modified Gram-Schmidt (MGS) method is preferred, which is stable with respect to rounding errors. However, the inner products are calculated sequentially when MGS is used in the original GMRES method. So clustering of the inner product communications is impossible. Since for the Cray T3D, the latency is relatively small, we use in our T3D specific code [83] the Modified Gram-Schmidt method for stability reasons. On a computer with a relative large latency, it is better to use an adapted GMRES method ([19], [6]) where a parallel (clustered) variant of the MGS method can be used.

Table 1 contains the Megaflop rates for the inner product on the Cray T3D. In theory the maximum Megaflop rate of 1 processor is 150 Mflop/s. The observed flop rates are much lower: 33.5 for the inner product (Table 1). It appears that memory access is the most time consuming part. Note that on 1 processor the Megaflop rate increases for an increasing vector length (pipelining). For the inner product this leads to an expected rate of 4288 Mflop/s on

	measured		
p	1	16	128
#elem per proc			
32	10	34	194
1024	28	331	2346
8192	33	496	3875
65536	33	529	4212

Table 1: Inner product performance in Megaflops per second on the Cray T3D

128 processors. For long vectors the observed rate (see Table 1) is very close to this value.

Matrix vector product
To calculate a matrix vector product we divide the matrix in blocks. As an example we consider the matrix given in Figure 27.
The parts A_{11}, A_{12} are on processor 1, A_{21}, A_{22}, A_{23} on processor 2 etc. For the vector x which is divided on the processors we need the following communication: x_1 to processor 2, x_2 to processor 1 and 3 etc, note that a 1D torus is well suited for this approach. Then $y_1 = A_{11}x_1 + A_{12}x_2$ is calculated on processor 1 and $y_2 = A_{21}x_1 + A_{22}x_2 + A_{23}x_3$ is calculated on processor 2 etc. Other schemes are possible but this illustrates the ideas to parallelize a matrix vector product.

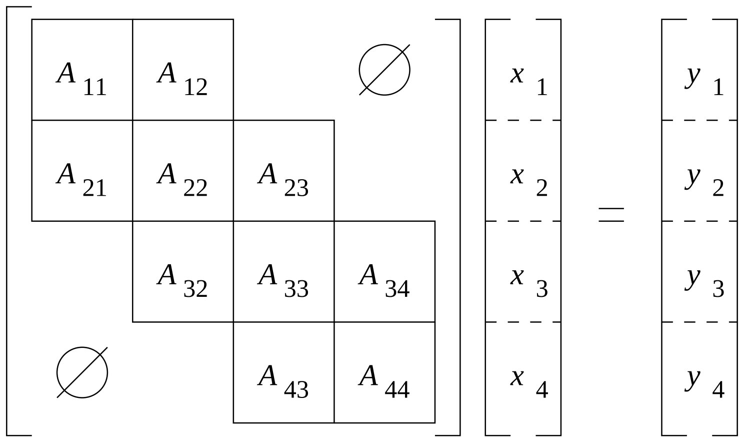


Figure 27: The calculation of $A * x$

Preconditioning
We restrict ourselves to a ILU preconditioning. This is in general the most difficult part to parallelize. Again lower and upper triangular systems have to be solved. A lower triangular system is sketched in Figure 28. Again recurrences prohibit parallelization of the computa-

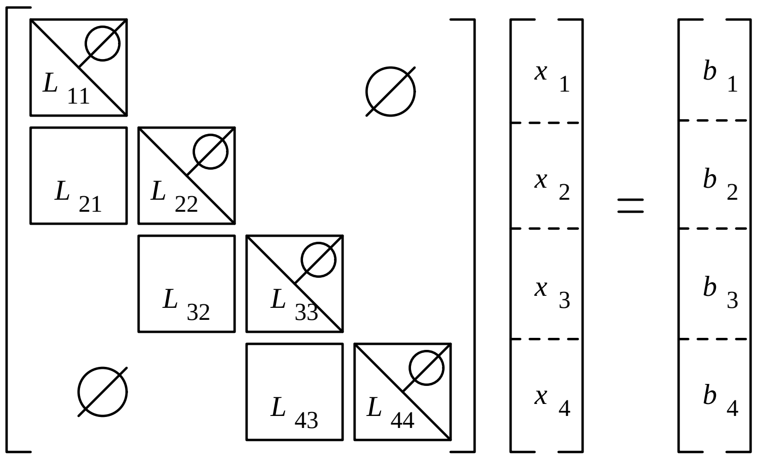


Figure 28: The calculating of $L^{-1}b$

tion of $L^{-1}b$. One can only start the computation of x_2 if x_1 is known. In many references a slightly adapted ILU decomposition is considered, which has better parallel properties. To parallelize the ILU preconditioner the couplings between the blocks are deleted. For most of these preconditioners the rate of convergence deteriorates when the number of blocks (processors) increases. When overlapping blocks are used the convergence behavior depends only slightly on the number of blocks. However, overlapping costs extra work. Another possibility for structured grid is to use the original ILU preconditioner and try to parallelize this. Both approaches are considered in more detail.

Parallel adapted ILU
In order to delete the couplings between the blocks one takes $L_{21} = 0, L_{32} = 0$ and $L_{43} = 0$ (looks like block Jacobi) [68]. This leads to very good parallel code. However, the number

of iterations may increase because the parallel preconditioner is worse than the original preconditioner. In order to reduce this effect, it is suggested in [62] to construct incomplete decompositions on slightly overlapping domains. This requires communication similar to that of matrix vector products. In [62] good speedup is observed for this idea.

Parallel original ILU

Parallelization of the construction of L, U, and the solution of the triangular systems is comparable. So we only consider the parallel implementation of the solution of $Lx = b$. The algorithm is explained for a matrix originating from a 5-point stencil. This approach is denoted as staircase parallelization of the ILU preconditioner [83]. A related approach is to calculate the unknowns on a diagonal of the grid, which is used for vectorization.

A rectangular computational domain is first decomposed into p strips parallel to the x_2-axis. We assume that the number of strips is equal to the number of processors. The number of grid points in x_d-direction is denoted by n_d. For ease of notation we assume that n_1 can be divided by p and set $n_x = n_1/p$ and $n_y = n_2$. The index i refers to the index in x_1-direction and j to the index in x_2-direction. The k^{th} strip is described by the following set $S_k = \{(i,j)|i \in [(k-1)\cdot n_x + 1, k \cdot n_x],\ j \in [1, n_y]\}$.

The vector of unknowns is denoted by $x(i,j)$. For a 5-point stencil it appears that in the solution of $Lx = b$, unknown $x(i,j)$ only depends on $x(i-1,j)$ and $x(i,j-1)$. The parallel algorithm now runs as follows: first all elements $x(i,1)$ for $(i,1) \in S_1$ are calculated on processor 1. Thereafter communication takes place between processor 1 and 2. Now $x(i,2)$ for $(i,2) \in S_1$ and $x(i,1)$ for $(i,1) \in S_2$ can be calculated in parallel etc. After some start-up time all processors are busy (Figure 29).

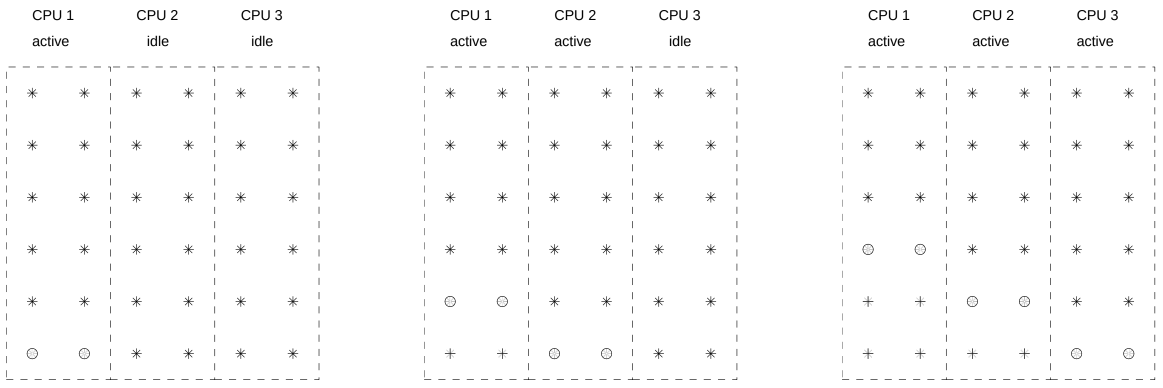


Figure 29: The first stages of the staircase parallel solution of the lower triangular system $Lx = b$. The symbols denote the following: $*$ nodes to be calculated, o nodes being calculated, and $+$ nodes that have been calculated.

Table 2 is copied from [83]. Note that when the grid size is large enough the total time per unknown is independent of the number of processors, which means that the method is scalable. In figures 30 and 31 we present the percentage of the total time for the various parts of GMRES. Figure 30 contains the results for $p = 8$ and an increasing grid size. It appears that the preconditioner vector product is the most time consuming part, it takes 65 % of the time for a small grid size and 45 % for a large grid size. In Figure 31 the results are shown for

p	1	2	4	8	16	32	64	128	256
32×8	259	645	1204	2510					
64×16	185	402	597	965	1969				
128×32	179	340	395	518	830	1694			
256×64	163	319	331	380	487	833	1521		
512×128		306	311	338	373	478	740	1443	
1024×256				317	335	375	469	731	1444
2048×512						354	374	492	722

Table 2: Measured total time per unknown in μ seconds for the solution of a Poisson equation on the Cray T3D (p = number of processors)

the Gustafsson model, the grid size increases linearly with the number of processors. There is only a small increase in the percentage used for the preconditioner vector product. This model suggests, as expected, that the preconditioner can be a bottle-neck especially if the number of grid cells in x_1-direction per processor is small.

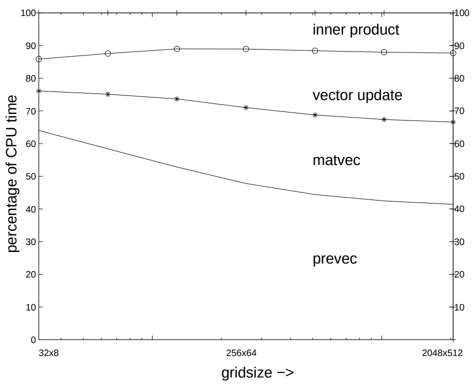


Figure 30: The percentage of time used by the various parts (8 processors)

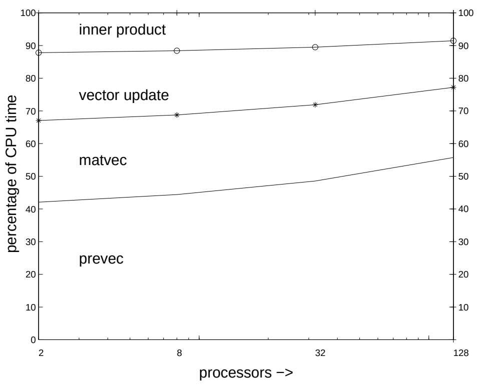


Figure 31: The percentage of time used by the various parts (grid size $256 \cdot \sqrt{p/2} \times 64 \cdot \sqrt{p/2}$)

8.3.4 The block Jacobi preconditioner

We consider an elliptic partial differential equation discretized using a cell-centered finite difference method on a computational domain Ω. Let the domain be the union of M nonoverlapping subdomains Ω_m, $m = 1, \ldots, M$. Discretization results in a sparse linear system $Ax = b$, with x, $b \in \mathrm{R}^N$. When the unknowns in a subdomain are grouped together one gets the block system:

$$\begin{bmatrix} A_{11} & \ldots & A_{1M} \\ \vdots & \ddots & \vdots \\ A_{M1} & \ldots & A_{MM} \end{bmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_M \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_M \end{pmatrix}. \tag{82}$$

In this system, the diagonal blocks A_{mm} express coupling among the unknowns defined on Ω_m, whereas the off-diagonal blocks A_{mn}, $m \neq n$ represent coupling across subdomain boundaries.

The only nonzero off-diagonal blocks are those corresponding to neighboring subdomains.

In order to solve system (82) with a Krylov subspace method we use the block Jacobi preconditioner:

$$K = \begin{bmatrix} A_{11} & & \\ & \ddots & \\ & & A_{MM} \end{bmatrix}.$$

When this preconditioner is used, systems of the form $Kv = r$ have to be solved. Since there is no overlap the diagonal blocks $A_{mm}v_m = r_m$, $m = 1, \ldots, M$ can be solved in parallel. In our method a blockwise application of the RILU preconditioner is used.

The convergence behavior of the block preconditioned GCR method

As a test example, we consider a Poisson problem, discretized with the finite volume method on a square domain. We do not exploit the symmetry of the Poisson matrix in these experiments. The domain is composed of a $\sqrt{p} \times \sqrt{p}$ array of subdomains, each with an $n \times n$ grid. With $h = \Delta x = \Delta y = 1.0/(n\sqrt{p})$ the discretization is

$$4u_{i,j} - u_{i+1,j} - u_{i-1,j} - u_{i,j-1} - u_{i,j+1} = h^2 f_{i,j}.$$

The right hand side function is $f_{i,j} = f(ih, jh)$, where $f(x, y) = -32(x(1-x) + y(1-y))$. Homogeneous Dirichlet boundary conditions $u = 0$ are defined on $\partial\Omega$, implemented by adding a row of ghost cells around the domain, and enforcing the condition, for example, $u_{0,j} = -u_{1,j}$ on boundaries. This ghost cell scheme allows natural implementation of the block preconditioner as well.

For the tests of this section, GCR is restarted after 30 iterations, and modified Gram-Schmidt was used as the orthogonalization method for all computations. The solution was computed to a fixed tolerance of 10^{-6}. We compare results for a fixed problem size on the 300×300 grid using 4, 9, 16 and 25 blocks. In Table 3 the iteration counts are given. Note that the number of iterations increases when the number of blocks grows. This implies that the parallel efficiency decreases when one uses more processors. In the next sections we present two

	$p = 4$	$p = 9$	$p = 16$	$p = 25$
RILU	341	291	439	437

Table 3: Number of iterations for various number of blocks

different approaches to diminish this drawback.

Overlapping of the subdomains

It is well known that the convergence of an overlapping block preconditioner is nearly independent of the subdomain grid size when the physical overlap region is constant. To describe the overlapping block preconditioner we define the subdomains $\Omega_m^* \subset \Omega$. The domain Ω_m^* consists of Ω_m and n_{over} neighboring grid points (see Figure 32). The matrix corresponding to this subdomain is denoted by A_{mm}^*. Application of the preconditioner goes as follows: given r compute v using the steps

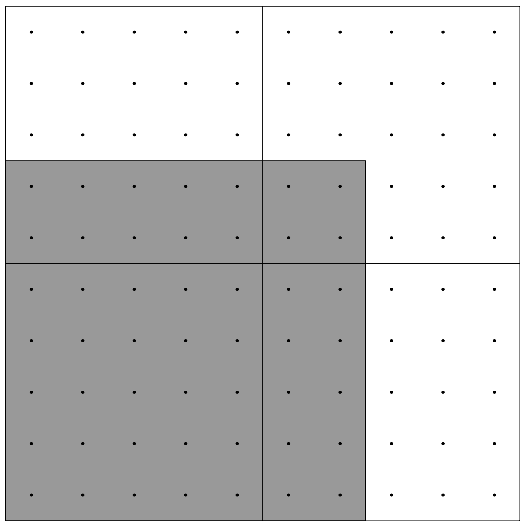


Figure 32: The shaded region is subdomain Ω_1^* for $n_{over} = 2$

1. r_m^* is the restriction of r to Ω_m^*,
2. solve $A_{mm}^* v_m^* = r_m^*$ in parallel,
3. form v_m, which is the restriction of v_m^* to Ω_m.

A related method is presented by Cai, Farhat and Sarkis [14]. A drawback of overlapping subdomains is that the amount of work increases proportional to n_{over}. Furthermore, it is not so easy to implement this approach on top of an existing software package.

Deflation

We present the Deflation acceleration only for the symmetric case. In our implementation we use the Deflated ICCG method as defined in [82]. To define the Deflated ICCG method we need a set of projection vectors $v_1, ..., v_M$ that form an independent set. The projection on the space A-perpendicular to span$\{v_1, ..., v_M\}$ is defined as

$$P = I - AVE^{-1}V^T \text{ with } E = (AV)^T V \text{ and } V = [v_1 ... v_M] \ .$$

The solution vector x can be split into two parts $x = (I - P^T)x + P^T x$. The first part can be calculated as follows $(I - P^T)x = VE^{-1}V^T Ax = VE^{-1}V^T b$. For the second part we project the solution x_j obtained from DICCG to $P^T x_j$. DICCG consists of applying CG to $L^{-T}L^{-1}PAx = L^{-T}L^{-1}Pb$.

The Deflated ICCG algorithm reads (see Reference [82]):

```
DICCG
$j = 0,\ \hat{r}_0 = Pr_0,\ p_1 = z_1 = L^{-T}L^{-1}\hat{r}_0;$
while $\|\hat{r}_j\|_2 >$ accuracy do
    $j = j + 1;\ \alpha_j = \frac{(\hat{r}_{j-1}, z_{j-1})}{(p_j, PAp_j)};$
    $x_j = x_{j-1} + \alpha_j p_j;$
    $\hat{r}_j = \hat{r}_{j-1} - \alpha_j P^T A p_j$;
    $z_j = L^{-T} L^{-1} \hat{r}_j$; $\beta_j = \frac{(\hat{r}_j, z_j)}{(\hat{r}_{j-1}, z_{j-1})}$;
    $p_{j+1} = z_j + \beta_j p_j$;
end while
```

For the coarse grid projection vectors we choose the vectors v_m as follows:

$$v_m(i) = 1, \;\; i \in \Omega_m, \;\; \text{and } v_m(i) = 0, \;\; i \notin \Omega_m. \tag{83}$$

We are able to give a sharp upperbound for the effective condition number of the deflated matrix, used with and without classical preconditioning [30]. This bound provides direction in choosing a proper decomposition into subdomains and a proper choice of classical preconditioner. If grid refinement is done keeping the subdomain resolutions fixed, the condition number can be shown to be independent of the number of subdomains.

In parallel, we first compute and store $((AV)^T V)^{-1}$ in factored form on each processor. Then to compute PAp we first perform the matrix-vector multiplication $w = Ap$, requiring nearest neighbor communications. Then we compute the local contribution to the restriction $\tilde{w} = V^T w$ and distribute this to all processors. With this done, we can solve $\tilde{e} = ((AV)^T V)^{-1} \tilde{w}$ and compute $(AV)^T \tilde{e}$ locally. The total communications involved in the matrix-vector multiplication and deflation are a nearest neighbor communication of the length of the interface and a global gather-broadcast of dimension M.

Block preconditioner and overlap

We consider a Poisson problem on a square domain with Dirichlet boundary conditions and a constant right-hand-side function. The problem is discretized by cell-centered finite differences. We consider overlap of 0, 1 and 2 grid points and use A_{mm}^{-1} in the block preconditioner. Table 4 gives the number of iterations necessary to reduce the initial residual by a factor 10^6 using a decomposition into 3×3 blocks with subgrid dimensions given in the table. Note that the number of iterations is constant along the diagonals. This agrees with domain decomposition theory that the number of iterations is independent of the subdomain grid size when

	overlap		
grid size	0	1	2
5×5	10	8	7
10×10	14	9	8
20×20	19	13	10
40×40	26	18	14

Table 4: Iterations for various grid sizes

the physical overlap remains the same.

In the second experiment we take a 5×5 grid per subdomain. The results for various number of blocks are given in Table 5. Note that without overlap the number of iterations increases considerably, whereas the increase is much smaller when 2 grid points are overlapped. The large overlap (2 grid points on a 5×5 grid) that has been used in this test, is not affordable for real problems.

Deflation

	overlap			overlap+deflation		
decomposition	0	1	2	0	1	2
2×2	6	5	4	6	4	4
3×3	10	8	7	11	6	6
4×4	15	9	7	14	9	6
5×5	18	12	9	16	10	8
6×6	23	13	10	17	11	9
7×7	25	16	12	17	12	10
8×8	29	17	12	18	13	10
9×9	33	19	14	18	14	11

Table 5: Iterations for various block decompositions with and without Deflation (subdomain grid size 5×5)

We do the same experiments using the Deflation (see Table 5). Initially we see some increase of the number of iterations, however, for more than 16 blocks the increase levels off. This phenomenon is independent of the amount of overlap. The same conclusion holds when block RILU is used instead of fully solving the subdomain problems.

Timing results of Deflation

Finally we present some timing results on the Cray T3E for a problem on a 480×480 grid. The results are given in Table 6. In this experiment we use GCR with the block RILU preconditioner combined with Deflation. Note that the number of iterations decreases when the number of blocks increases. This leads to an efficiency larger than 1. The decrease in iterations is partly due to the improved approximation of the RILU preconditioner for smaller subdomains. On the other hand when the number of blocks increases, more small eigenvalues are projected to zero which also accelerates the convergence (see [30]). We expect that there is some optimal value for the number of subdomains, because at the extreme limit there is only one point per subdomain and the coarse grid problem is identical to the original problem so there is no speedup at all.

p	iterations	time	speedup	efficiency
1	485	710	-	-
4	322	120	5	1.2
9	352	59	12	1.3
16	379	36	20	1.2
25	317	20	36	1.4
36	410	18	39	1.1
64	318	8	89	1.4

Table 6: Speedup of the iterative method using a 480×480 grid

References

[1] G. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. *Proceeding of the AFIPS computing conference*, 30:483–485, 1967.

[2] W.E. Arnoldi. The principe of minimized iterations in the solution of the matrix eigenvalue problem. *Quart. Appl. Math.*, 9:17–29, 1951.

[3] C.C. Ashcraft and R.G. Grimes. On vectorizing incomplete factorization and SSOR preconditioners. *SIAM J. Sci. Stat. Comput.*, 9:122–151, 1988.

[4] O. Axelsson. *Iterative Solution Methods.* Cambridge University Press, Cambridge, UK, 1994.

[5] O. Axelsson and G. Lindskog. On the eigenvalue distribution of a class of preconditioning methods. *Numer. Math.*, 48:479–498, 1986.

[6] Z. Bai, D. Hu, and L. Reichel. A Newton basis GMRES implementation. *IMA J. Num. Anal.*, 14:563–581, 1994.

[7] D.H. Bailey. Twelve ways to fool the masses when giving performance results on parallel computers. *Supercomputer*, 45:4–7, 1991.

[8] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, 1994.

[9] A. Björck and T. Elfving. Accelerated projection methods for computing pseudo-inverse solution of systems of linear equations. *BIT*, 19:145–163, 1979.

[10] E.K. Blum. *Numerical Analysis and Computation, Theory and Practice.* Addison-Wesley, Reading, 1972.

[11] E. Brakkee, A. Segal, and C.G.M. Kassels. A parallel domain decomposition algorithm for the incompressible Navier-Stokes equations. *Simulation Practice and Theory*, 3:185–205, 1995.

[12] E. Brakkee, C. Vuik, and P. Wesseling. Domain decomposition for the incompressible Navier-Stokes equations: solving subdomain problems accurately and inaccurately. *Int. J. Num. Meth. in Fluids*, 26:1217–1237, 1998.

[13] A.M. Bruaset. *A Survey of Preconditioned Iterative Methods.* Pitman research notes in mathematics series 328. Longman Scientific and Technical, Harlow, 1995.

[14] X.-C. Cai and M. Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM J. Sci. Comput.*, 21:792–797, 1999.

[15] B.A. Carré. The determination of the optimum accelerating factor for successive over-relaxation. *Computer Journal*, 4:73–78, 1961.

[16] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers, San Francisco, 2001.

[17] F. Chatelin. *Spectral Approximation of Linear Operations*. Academic Press, New York, 1983.

[18] F. Chatelin. *Eigenvalues of Matrices*. Wiley, Chichester, 1993.

[19] E. de Sturler and H.A. van der Vorst. Reducing the effect of global communication in GMRES(m) and CG on parallel distributed memory computers. *Appl. Num. Math.*, 18:441–459, 1995.

[20] J.W. Demmel, M.T. Heath, and H.A. van der Vorst. Parallel numerical linear algebra. In A. Iserles, editor, *Acta Numerica*, pages 111–197. Cambridge University Press, Cambridge, UK, 1993.

[21] J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, 1991.

[22] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Oxford, 1986.

[23] I.S. Duff and J.K. Reid. Some design features of a sparse matrix code. *ACM Trans. Math. Software*, 5:18–35, 1979.

[24] M. Eiermann, W. Niethammer, and R.S. Varga. A study of semiiterative methods for nonsymmetric systems of linear equations. *Numer. Math.*, 47:505–533, 1985.

[25] S.C. Eisenstat. Efficient implementation of a class of preconditioned conjugate gradient methods. *SIAM J. Sci. Stat. Comput.*, 2:1–4, 1981.

[26] S.C. Eisenstat, H.C. Elman, and M.H. Schultz. Variable iterative methods for nonsymmetric systems of linear equations. *SIAM J. Num. Anal.*, 20:345–357, 1983.

[27] V. Faber and T. Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM J. Num. Anal.*, 21:356–362, 1984.

[28] R. Fletcher. Factorizing symmetric indefinite matrices. *Lin. Alg. and its Appl.*, 14:257–277, 1976.

[29] M.J. Flynn. Very high speed computing systems. *Proc. IEEE*, 54:1901–1909, 1966.

[30] J. Frank and C. Vuik. On the construction of deflation-based preconditioners. MAS-R 0009, CWI, Amsterdam, 2000.

[31] R.W. Freund, G.H. Golub, and N.M. Nachtigal. Iterative solution of linear systems. In A. Iserles, editor, *Acta Numerica*, pages 57–100. Cambridge University Press, Cambridge, UK, 1992.

[32] R.W. Freund, M.H. Gutknecht, and N.M. Nachtigal. An implimentation of the look-ahead Lanczos algorithm for non-Hermitian matrices. *SIAM J. Sci. Comp.*, 14:137–156, 1993.

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

[34] A. George and J.W.H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Engelwood Cliffs, New Jersey, (USA), 1981.

[35] T. Ginsburg. The conjugate gradient method. In J.H. Wilkinson and C. Reinsch, editors, *Handbook for Automatic Computation, 2, Linear Algebra*, pages 57–69, Berlin, 1971. Springer.

[36] G.H. Golub, R. Underwood, and J.H. Wilkinson. The Lanczos algorithm for the symmetric $Ax = \lambda Bx$ problem. Report STAN-CS-72-270, Department of Computer Science, Stanford University, Stanford, California, 1972.

[37] G.H. Golub and C.F. van Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, 1996. Third edition.

[38] G.H. Golub and R.S. Varga. Chebychev semi-iterative methods, successive over-relaxation iterative methods and second order Richardson iterative methods. Part I and II. *Numer. Math.*, 3:147–156, 157–168, 1961.

[39] A. Greenbaum. *Iterative Methods for Solving Linear Systems.* Frontiers in applied mathematics 17. SIAM, Philadelphia, 1997.

[40] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI, portable programming with the Message-Passing Interface.* Scientific and Engineering Computation Series. The MIT Press, Cambridge, 1994.

[41] J.L. Gustafson. Reevaluating Amdahl's law. *Comm. ACM*, 31:532–533, 1988.

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

[43] L.A. Hageman and D.M. Young. *Applied Iterative Methods.* Academic Press, New York, 1981.

[44] M.R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *J. Res. Nat. Bur. Stand.*, 49:409–436, 1952.

[45] R.W. Hockney and C.R. Jesshope. *Parallel Computers 2: architecture, programming and algorithms.* Adam Hilger, Bristol, 1988.

[46] T.J.R. Hughes. *The Finite Element Method, Linear Static and Dynamic Finite Element Analysis.* Prentice Hall Inc., Englewood Cliffs, New Yersey, 1987.

[47] E.F. Kaasschieter. Preconditioned conjugate gradients for solving singular systems. *J. of Comp. Appl. Math.*, 24:265–275, 1988.

[48] I.P. King. An automatic reordening scheme for simultaneous equations derived from network systems. *Int. J. Num. Meth. in Eng.*, 2:523–533, 1970.

[49] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing; design and analysis of algorithms.* Benjamin/Cummings, Redwood City, 1994.

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

[51] K. Li and R. Schaefer. A hypercube shared virtual memory system. *Proceedings of 1989 International conference on parallel processing*, I:125–132, 1989.

[52] T.A. Manteuffel. The Tchebychev iteration for nonsymmetric linear systems. *Numer. Math.*, 28:307–327, 1977.

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

[54] M. Metcalf and J. Reid. *Fortran 90/95 explained*. Oxford University Press, Oxford, 1996.

[55] N.M. Nachtigal, S.C. Reddy, and L.N. Trefethen. How fast are non symmetric matrix iterations. *SIAM J. Matrix Anal. Appl.*, 13:778–795, 1992.

[56] J.M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems*. Plenum Press, New York, 1988.

[57] C.C. Paige and M.A. Saunders. Solution of sparse indefinite system of linear equations. *SIAM J. Num. Anal.*, 12:617–629, 1975.

[58] C.C. Paige and M.A. Saunders. LSQR: an algorithm for sparse linear equations and sparse least square problem. *ACM Trans. Math. Softw.*, 8:44–71, 1982.

[59] B.N. Parlett. *The Symmetric Eigenvalue Problem*. Prentice-Hall, Englewood Cliffs, 1980.

[60] B.N. Parlett and D.S. Scott. The Lanczos algorithm with selective orthogonalization. *Math. Comp.*, 33:217–238, 1979.

[61] B.N. Parlett, D.R. Taylor, and Z.A. Liu. A look-ahead Lanczos algorithm for unsymmetric matrices. *Math. Comp.*, 44:105–124, 1985.

[62] G. Radicati di Brozolo and Y. Robert. Vector and parallel CG-like algorithms for sparse non-symmetric systems. Technical Report 681-M, IMAG/TIM3, Grenoble, 1987.

[63] J.K. Reid. The use of conjugate for systems of linear equations posessing property A. *SIAM J. Num. Anal.*, 9:325–332, 1972.

[64] Y. Saad. Preconditioning techniques for non symmetric and indefinite linear system. *J. Comp. Appl. Math.*, 24:89–105, 1988.

[65] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Stat. Comput.*, 14:461–469, 1993.

[66] Y. Saad. *Iterative methods for sparse linear systems, Second Edition*. SIAM, Philadelphia, 2003.

[67] Y. Saad and M.H. Schultz. GMRES: a generalized minimal residual algorithm for solving non-symmetric linear systems. *SIAM J. Sci. Stat. Comp.*, 7:856–869, 1986.

[68] M.K. Seager. Parallelizing conjugate gradient for the Cray X_MP. *Parallel Computing*, 3:35–47, 1986.

[69] H.D. Simon. The Lanczos algorithm with partial reorthogonalization. *Math. Comp.*, 42:115–142, 1984.

[70] P. Sonneveld. CGS: a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 10:36–52, 1989.

[71] A. van der Sluis. Conditioning, equilibration, and pivoting in linear algebraic systems. *Numer. Math.*, 15:74–86, 1970.

[72] A. van der Sluis and H.A. van der Vorst. The rate of convergence of conjugate gradients. *Numer. Math.*, 48:543–560, 1986.

[73] H.A. van der Vorst. A vectorizable variant of some ICCG methods. *SIAM J. Sci. Stat. Comp.*, 3:350–356, 1982.

[74] H.A. van der Vorst. High performance preconditioning. *SIAM J. Sci. Stat. Comp.*, 10:1174–1185, 1989.

[75] H.A. van der Vorst. Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for solution of non-symmetric linear systems. *SIAM J. Sci. Stat. Comp.*, 13:631–644, 1992.

[76] H.A. van der Vorst. *Iterative Krylov Methods for Large Linear Systems.* Cambridge Monographs on Applied and Computational Mathematics, 13. Cambridge University Press, Cambridge, 2003.

[77] H.A. van der Vorst and C. Vuik. The superlinear convergence behaviour of GMRES. *J. Comput. Appl. Math.*, 48:327–341, 1993.

[78] H.A. van der Vorst and C. Vuik. GMRESR: a family of nested GMRES methods. *Num. Lin. Alg. Appl.*, 1:369–386, 1994.

[79] R.S. Varga. *Matrix Iterative Analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1962.

[80] Vector:. MPP lags in Los Alamos tests, Vector are Victors. *High Performance Computing Review*, 1:2:10–17, 1993.

[81] C. Vuik. Solution of the discretized incompressible Navier-Stokes equations with the GMRES method. *Int. J. Num. Meth. in Fluids*, 16:507–523, 1993.

[82] C. Vuik, A. Segal, and J.A. Meijerink. An efficient preconditioned CG method for the solution of a class of layered problems with extreme contrasts in the coefficients. *J. Comp. Phys.*, 152:385–403, 1999.

[83] C. Vuik, R.R.P. van Nooyen, and P. Wesseling. Parallelism in ILU-preconditioned GMRES. *Paral. Comp.*, 24:1927–1946, 1998.

[84] E.L. Wachspress. *Iterative Solution of Elliptic Systems.* Prentice-Hall, Englewood Cliffs, 1966.

[85] J.H. Wilkinson. *The Algebraic Eigenvalue Problem.* Clarendon Press, Oxford, England, 1965.

[86] J.H. Wilkinson. A priori error analysis of algebraic processes. In *Proc. International Congress Math.*, pages 629–639, Moscow, 1968. Izdat. MIR.

[87] D.M. Young. *Iterative Solution of Large Linear Systems.* Academic Press, New York, 1971.

[88] Z. Zlatev. *Computational Methods for General Sparse Matices, Mathematics and Applications 65.* Kluwer Academic Publishers, Dordrecht, 1991.