Software for Numerical Linear Algebra

Prof. P. Arbenz
Dr. O. Chinellato
Prof. M. Gutknecht
Dr. M. Sala

Zürich, 2006

Contents

Chapter 1

Introduction

These are the lecture notes of a course on “Software for Numerical Linear Algebra” that was taught in the summer semester 2006 at the Computer Science Department of ETH Zurich. The course was taught by other lecturers (Proffs. Walter Gander, Martin Gutknecht and Beresford Parlett) in previous years. The course is targeted at computer science students who had attended classes in linear algebra and introductory computational science.

The course consisted of 14 two-hours lectures and the same amount of time for tutoring student’s assignments. These lecture notes contain more material than was actually taught. The sections that were not covered are indicated by an asterisk (★). They contain additional or complimentary material that we found interesting.

The topics covered in the lectures were

Dense linear algebra with LAPACK

In this part of the lectures emphasis was given on eigenvalue computations since the students had been exposed to dense system solving by Gaussian elimination in earlier courses. We decided to cover

- the QR algorithm including important ideas as Householder reflectors, QR factorization in particular of Hessenberg matrices, Givens rotations, shifting strategies for improving convergence, etc.
- Cuppen’s divide and conquer algorithm as a more recent idea for solving the symmetric tridiagonal eigenvalue problem. This algorithm is in principle quite easy to understand. On the other hand there are a number of interesting details (deflation, zero finder, spectral shifts) that have to be solved properly, before the algorithm becomes stable algorithm.
- These algorithms are efficiently implemented in LAPACK, a huge collection of Fortran subroutines for solving the basic tasks of dense linear algebra.

Sparse matrix computations

The part on sparse matrix computations was the largest and covered eight out of 14 lectures. Various aspects concerning the solution of linear systems and eigenvalue problems with sparse system matrices were addressed. The occurrence of sparse matrices was motivated by finite element discretizations of second order partial differential equations.

Topics covered from numerical linear algebra were

- Storage schemes for sparse matrices.
- Direct solution methods and matrix reorderings to reduce fill-in. Only little time was devoted to this issue. It was treated as an introduction to ideas used in preconditioning based on incomplete factorizations.

- Iterative solution methods based on Krylov spaces: conjugate gradient type methods (e.g., CG, PCG, GMRes).
- Preconditioning of linear systems to reduce the condition of the system to be solved and in this way lower the number of iteration steps until convergence. Several preconditioning techniques were covered: besides conventional splitting techniques, incomplete LU and Cholesky factorizations, domain decomposition, and multigrid preconditioners.
- ARPACK was chosen as an example for a widely used software package that is used for sparse eigenproblems. In contrast to system solving where numerous software packages are available, for solving eigenvalue problems there seems to be just this package in the public domain that provides stable and efficient solution paths for symmetric/Hermitian or nonsymmetric/non-Hermitian eigenvalue problems and the singular value decomposition.

Parallel computation with the Trilinos framework

Two lectures were devoted to issues in parallel computation. Besides a brief introduction on basic concepts (Flynn's taxonomy) and terms (speedup, efficiency, latency, bandwidth) the following topics were covered

- Basic MPI programming with point-to-point and collective communication.
- Techniques to enhance load balancing based on graph partitioning.
- Finally, an introduction to the Trilinos framework for solving linear (and nonlinear) algebra problems on distributed memory parallel computers. Trilinos provides a wealth of preconditioners for iterative solvers. Its C++ interface makes it easy for computer science students to solve reasonably sized problems in a short time.

Student's Assignments

The purpose of assignments is to enforce/encourage students to explore the algorithms they learned during the course. Students were required to complete twelve assignments that consisted mainly in applying and comparing the software and algorithms discussed on a few simple examples. Some assignments also required MATLAB programming. (At ETH, CS students can be assumed to be knowledgeable in MATLAB.) Besides having a convenient interface, MATLAB provides easy means to visualize solutions of finite element problems or the convergence behaviour of iterative solvers.

May 2006, PA, OC, MG, MS.

Part I

Dense Linear Algebra

Chapter 2

Basics

2.1 Notation

The fields of real and complex numbers are denoted by $\mathbb{R}$ and $\mathbb{C}$, respectively. Elements in $\mathbb{R}$ and $\mathbb{C}$, *scalars*, are denoted by lowercase letters, a, b, $c, \ldots$, and α, β, $\gamma, \ldots$

Vectors are denoted by boldface lowercase letters, $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}, \ldots$, and $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}, \ldots$ We denote the space of vectors of n *real* components by $\mathbb{R}^n$ and the space of vectors of n *complex* components by $\mathbb{C}^n$.

$$\mathbf{x} \in \mathbb{R}^n \Longleftrightarrow \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad x_i \in \mathbb{R}. \tag{2.1}$$

We often make statements that hold for real or complex vectors or matrices. Then we write, e.g., $\mathbf{x} \in \mathbb{F}^n$.

The **inner product** of two n-vectors in $\mathbb{C}$ is defined as

$$(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} x_i \bar{y}_i = \mathbf{y}^* \mathbf{x}, \tag{2.2}$$

that is, we require linearity in the first component and sesquilinearity(?) in the second.

$\mathbf{y}^* = (\bar{y}_1,\ \bar{y}_2, \ldots, \bar{y}_n)$ denotes conjugate transposition of complex vectors. To simplify notation we denote real transposition by an asterisk as well.

Two vectors $\mathbf{x}$ and $\mathbf{y}$ are called **orthogonal**, $\mathbf{x} \perp \mathbf{y}$, if $\mathbf{x}^*\mathbf{y} = 0$.

The inner product (2.2) induces a **norm** in $\mathbb{F}$,

$$\|\mathbf{x}\| = \sqrt{(\mathbf{x}, \mathbf{x})} = \left(\sum_{i=1}^{n} |x_i|^2 \right)^{1/2}. \tag{2.3}$$

This norm is often called Euklidian norm or 2-norm.

The group of m-by-n **matrices** with components in the field $\mathbb{F}$ is denoted by $\mathbb{F}^{m \times n}$,

$$A \in \mathbb{F}^{m \times n} \Longleftrightarrow A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix}, \quad a_{ij} \in \mathbb{F}. \tag{2.4}$$

The matrix $A^* \in \mathbb{F}^{n \times m}$,

$$A^* = \begin{pmatrix} \bar{a}_{11} & \bar{a}_{21} & \ldots & \bar{a}_{m1} \\ \bar{a}_{12} & \bar{a}_{22} & \ldots & \bar{a}_{m2} \\ \vdots & \vdots & & \vdots \\ \bar{a}_{1n} & \bar{a}_{2n} & \ldots & \bar{a}_{nm} \end{pmatrix} \tag{2.5}$$

is the (Hermitian) **transposed** of A. Notice, that with this notation n-vectors can be identified with n-by-1 matrices.

The following classes of square matrices are of particular importance

- $A \in \mathbb{F}^{n\times n}$ is called **Hermitian** if and only if $A^* = A$.
- A *real* Hermitian matrix is called **symmetric**.
- $U \in \mathbb{F}^{n\times n}$ is called **unitary** if and only if $U^{-1} = U^*$.
- *Real* unitary matrices are called **orthogonal**.

We define the norm of a matrix to be the norm induced by the vector norm (2.3),

$$\|A\| := \max_{\mathbf{x}\neq\mathbf{0}} \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|} = \max_{\|\mathbf{x}\|=1} \|A\mathbf{x}\|. \tag{2.6}$$

The condition number of a nonsingular matrix is defined as $\kappa(A) = \|A\|\|A^{-1}\|$. It is easy to show that if U is unitary then $\|U\mathbf{x}\| = \|\mathbf{x}\|$ for all $\mathbf{x}$. Thus the condition number of a unitary matrix is 1.

2.2 Statement of the problem

The **(special) eigenvalue problem** is the following

Given a square matrix $A \in \mathbb{F}^{n\times n}$.
Find scalars $\lambda \in \mathbb{C}$ and vectors $\mathbf{x} \in \mathbb{C}$, $\mathbf{x} \neq \mathbf{0}$, such that

$$A\mathbf{x} = \lambda\mathbf{x}, \tag{2.7}$$

i.e., such that

$$(A - \lambda I)\mathbf{x} = \mathbf{0} \tag{2.8}$$

has a nontrivial (nonzero) solution.

So, we are looking for numbers λ such that $A - \lambda I$ is *singular*.

Definition 2.1 Let the pair $(\lambda, \mathbf{x})$ be a solution of (2.7) or (2.8), respectively. Then

- λ is called an **eigenvalue** of A,
- $\mathbf{x}$ is called an **eigenvector** corresponding to λ
- $(\lambda, \mathbf{x})$ is called **eigenpair** of A.
- The set $\sigma(A)$ of *all* eigenvalues of A is called **spectrum** of A.
- The set of all eigenvectors corresponding to an eigenvalue λ together with the vector **0** form a linear subspace of $\mathbb{C}^n$ called the **eigenspace** of λ. As the eigenspace of λ is the null space of $\lambda I - A$ we denote it by $\mathcal{N}(\lambda I - A)$.
- The dimension of $\mathcal{N}(\lambda I - A)$ is called **geometric multiplicity** $g(\lambda)$ of λ.
- An eigenvalue λ is a zero of the **characteristic polynomial**

  $$\chi(\lambda) := \mathsf{det}\,(\lambda I - A) = \lambda^n + a_{n-1}\lambda^{n-1} + \cdots + a_0.$$

  The multiplicity of λ as a zero of χ is called the **algebraic multiplicity** $m(\lambda)$ of λ. We will later see that

  $$1 \leq g(\lambda) \leq m(\lambda) \leq n, \qquad \lambda \in \sigma(A), \quad A \in \mathbb{F}^{n\times n}.$$

Remark 2.1. *A nontrivial solution solution* $\mathbf{y}$ *of*

$$\mathbf{y}^* A = \lambda \mathbf{y}^* \tag{2.9}$$

is called ***left eigenvector*** *corresponding to* λ. *A left eigenvector of* A *is a right eigenvector of* A^*, *corresponding to the eigenvalue* $\bar{\lambda}$, $A^*\mathbf{y} = \bar{\lambda}\mathbf{y}$.

Problem 2.2 Let $\mathbf{x}$ be a (right) eigenvector of A corresponding to an eigenvalue λ and let $\mathbf{y}$ be a left eigenvector of A corresponding to a *different* eigenvalue $\mu \neq \lambda$. Show that $\mathbf{x}^*\mathbf{y} = 0$.

Remark 2.2. *Let* A *be an* ***upper triangular*** *matrix,*

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ & a_{22} & \dots & a_{2n} \\ & & \ddots & \vdots \\ & & & a_{nn} \end{pmatrix}, \qquad a_{ik} = 0 \textit{ for } i > k. \tag{2.10}$$

Then we have

$$\mathsf{det}\,(\lambda I - A) = \prod_{i=1}^{n} (\lambda - a_{ii})$$

Problem 2.3 Let $\lambda = a_{ii}$, $1 \leq i \leq n$, be an eigenvalue of A in (2.10). Can you give a corresponding eigenvector? Can you explain a situation where $g(\lambda) < m(\lambda)$?

The **(generalized) eigenvalue problem** is the following

Given two square matrices $A, B \in \mathbb{F}^{n \times n}$.
Find scalars $\lambda \in \mathbb{C}$ and vectors $\mathbf{x} \in \mathbb{C}$, $\mathbf{x} \neq \mathbf{0}$, such that

$$A\mathbf{x} = \lambda B\mathbf{x}, \tag{2.11}$$

or, eqivalently, such that

$$(A - \lambda B)\mathbf{x} = \mathbf{0} \tag{2.12}$$

has a nontrivial solution.

Definition 2.4 Let the pair $(\lambda, \mathbf{x})$ be a solution of (2.11) or (2.12), respectively. Then

- λ is called an **eigenvalue** of A **relative to** B,
- $\mathbf{x}$ is called an **eigenvector** of A **relative to** B corresponding to λ.
- $(\lambda, \mathbf{x})$ is called an **eigenpair** of A **relative to** B,
- The set $\sigma(A;\ B)$ of *all* eigenvalues of (2.11) is called the **spectrum** of A **relative to** B.
- Let B be nonsingular. Then
$$A\mathbf{x} = \lambda B\mathbf{x} \Longleftrightarrow B^{-1}A\mathbf{x} = \lambda\mathbf{x} \tag{2.13}$$
- Let both A and B be Hermitian, $A = A^*$ and $B = B^*$. Let further be B positive definite and $B = LL^*$ be its Cholesky factorization. Then
$$A\mathbf{x} = \lambda B\mathbf{x} \Longleftrightarrow L^{-1}AL^{-*}\mathbf{y} = \lambda\mathbf{y}, \quad \mathbf{y} = L^*\mathbf{x}. \tag{2.14}$$
- Let A be invertible. Then $A\mathbf{x} = \mathbf{0}$ implies $\mathbf{x} = \mathbf{0}$. That is, $0 \notin \sigma(A;\ B)$. Therefore,
$$A\mathbf{x} = \lambda B\mathbf{x} \Longleftrightarrow \mu\mathbf{x} = A^{-1}B\mathbf{x}, \quad \mu = \frac{1}{\lambda} \tag{2.15}$$

- *Difficult situation*: both A and B are singular.

  1. Let, e.g.,
  $$A = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$
  Then,
  $$\begin{aligned} A\mathbf{e}_2 &= \mathbf{0} = 0 \cdot B\mathbf{e}_2 = 0 \cdot \mathbf{e}_2 \\ A\mathbf{e}_1 &= \mathbf{e}_1 = \lambda B\mathbf{e}_1 = \lambda\mathbf{0} \end{aligned}$$
  So 0 is an eigenvalue of A relative to B.
  As in (2.15) we may change roles of A and B. Then
  $$B\mathbf{e}_1 = \mathbf{0} = \mu A\mathbf{e}_1 = 0\mathbf{e}_1.$$
  So, $\mu = 0$ is an eigenvalue of B relative to A. We therefore say, informally, that $\lambda = \infty$ is an eigenvalue of A relative to B. So, $\sigma(A;\ B) = \{0, \infty\}$.
  2. Let
  $$A = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} = A.$$
  Then Then,
  $$\begin{aligned} A\mathbf{e}_1 &= 1 \cdot B\mathbf{e}_1, \\ A\mathbf{e}_2 &= \mathbf{0} = \lambda B\mathbf{e}_0 = \lambda\mathbf{0}, \text{ for } \textbf{all } \lambda \in \mathbb{C}. \end{aligned}$$
  Therefore, in this case, $\sigma(A;\ B) = \mathbb{C}$.

2.3 Similarity transformations

Definition 2.5 A matrix $A \in \mathbb{F}^{n\times n}$ is **similar** to a matrix $C \in \mathbb{F}^{n\times n}$, $A \sim C$, if and only if there is a nonsingular matrix S such that
$$S^{-1}AS = C. \tag{2.16}$$
The mapping $A \longrightarrow S^{-1}AS$ is called a **similarity transformation**.

Theorem 2.6. *Similar matrices have equal eigenvalues with equal multiplicities. If $(\lambda, \mathbf{x})$ is an eigenpair of A and $C = S^{-1}AS$ then $(\lambda, S^{-1}\mathbf{x})$ is an eigenpair of C.*

Proof. • $A\mathbf{x} = \lambda\mathbf{x}$ and $C = S^{-1}AS$ imply that
$$CS^{-1}\mathbf{x} = S^{-1}ASS^{-1}\mathbf{x} = S^{-1}\lambda\mathbf{x}.$$
Hence, A and C have equal eigenvalues and their geometric multiplicity is not changed by the similarity transformation.

•
$$\begin{aligned} \mathsf{det}\,(\lambda I - C) &= \mathsf{det}\,(\lambda S^{-1}S - S^{-1}AS) \\ &= \mathsf{det}\,(S^{-1}(\lambda I - A)S) = \mathsf{det}\,(S^{-1})\mathsf{det}\,(\lambda I - A)\mathsf{det}\,(S) = \mathsf{det}\,(\lambda I - A) \end{aligned}$$

■

Definition 2.7 Two matrices A and B are called **unitarily similar** if S in (2.16) is unitary. If the matrices are real the term orthogonally similar is used.

Unitary similarity transformations are very important in numerical computation. Let U be unitary. Then $\|U\| = \|U^{-1}\| = 1$, so U's condition number $\kappa(U) = 1$. So, if $C = U^{-1}AU$ then $C = U^*AU$ and $\|C\| = \|A\|$. In particular, if A is disturbed by δA then

$$U^*(A + \delta A)U = C + \delta C, \qquad \|\delta C\| = \|\delta A\|.$$

That is, errors (perturbations) in A are not amplified by a unitary similarity transformation. This is in contrast to arbitrary similarity transformations. However, small eigenvalues may suffer from large errors.

Another reason for the importance of unitary similarity transformations is the preservation of symmetry: If A is symmetric then $U^{-1}AU = U^*AU$ is symmetric as well.

For generalized eigenvalue problems, similarity transformations are not so crucial since we can operate with different matrices from both sides. If S and T are nonsingular

$$A\mathbf{x} = \lambda B\mathbf{x} \quad \Longleftrightarrow \quad TAS^{-1}S\mathbf{x} = \lambda TBS^{-1}S\mathbf{x},$$

Thus, $\sigma(A;\ B) = \sigma(TAS^{-1}, TBS^{-1})$. Let us consider a special case: let $B = LU$ be the LU-factorization of B. Then we set $S = U$ and $T = L^{-1}$ and obtain $TBU^{-1} = L^{-1}LUU^{-1} = I$. Thus, $\sigma(A;\ B) = \sigma(L^{-1}AU^{-1}, I) = \sigma(L^{-1}AU^{-1})$.

2.4 Schur decomposition

Theorem 2.8. ***(Schur decomposition)*** *If $A \in \mathbb{C}^{n\times n}$ then there is a unitary $U \in \mathbb{C}^{n\times n}$ such that*

$$U^*AU = T \tag{2.17}$$

is upper triangular. The diagonal elements of T are the eigenvalues of A.

Proof. The proof is by induction. For $n = 1$, the theorem is obviously true.

Assume that the theorem holds for matrices of order $\leq n - 1$. Let $(\lambda, \mathbf{x})$,
$vert\mathbf{x}$
$vert = 1$, be an eigenpair of A, $A\mathbf{x} = \lambda\mathbf{x}$. We construct a unitary matrix U_1 with first column $\mathbf{x}$ (e.g. the Householder reflector U_1 with $U_1\mathbf{x} = \mathbf{e}_1$). $U_1 = [\mathbf{x}, \overline{U}]$. Then

$$U_1^*AU_1 = \begin{bmatrix} \mathbf{x}^*A\mathbf{x} & \mathbf{x}^*A\overline{U} \\ \overline{U}^*A\mathbf{x} & \overline{U}^*A\overline{U} \end{bmatrix} = \begin{bmatrix} \lambda & \times\cdots\times \\ \mathbf{0} & \hat{A} \end{bmatrix}$$

as $A\mathbf{x} = \lambda\mathbf{x}$ and $\overline{U}^*\mathbf{x} = \mathbf{0}$ by construction of U_1. By assumption, there exists a unitary matrix $\hat{U} \in \mathbb{C}^{(n-1)\times(n-1)}$ such that $\hat{U}^*\hat{A}\hat{U} = \hat{T}$ is upper triangular. Setting $U := U_1(1 \oplus \hat{U})$, we obtain (2.17) ∎

Notice , that this proof is not constructive as we assume the knowledge of an eigenpair $(\lambda, \mathbf{x})$. So, we cannot employ it to actually compute the Schur form. The QR algorithm is used for this purpose. We will discuss this basic algorithm in chapter 3.

Let $U^*AU = T$ be the Schur decomposition of A with $U = [\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n]$. The Schur decomposition can be written as $AU = UT$. The k-th column of this equation is

$$A\mathbf{u}_k = \lambda\mathbf{u}_k + \sum_{i=1}^{k-1} t_{ik}\mathbf{u}_i, \qquad \lambda_k = t_{kk}. \tag{2.18}$$

This implies that

$$A\mathbf{u}_k \in \text{span}\{\mathbf{u}_1, \ldots, \mathbf{u}_k\}, \quad \forall k. \tag{2.19}$$

Thus, the first k **Schur vectors** $\mathbf{u}_1, \ldots, \mathbf{u}_k$ form an **invariant subspace**[1] for A. From (2.18) it is clear that the *first* Schur vector is an eigenvector of A. The further columns of U, however, are in general *no* eigenvectors of A. Notice, that the Schur decomposition is not unique. In the proof we have chosen *any* eigenvalue λ. This indicates that the eigenvalues can be arranged in any order in the diagonal of T. This also indicates that the order with which the eigenvalues appear on T's diagonal can be manipulated.

[1] A subspace $\mathcal{V} \subset \mathbb{F}^n$ is called invariant for A if $A\mathcal{V} \subset \mathcal{V}$.

Problem 2.9 Let

$$A = \begin{bmatrix} \lambda_1 & \alpha \\ 0 & \lambda_2 \end{bmatrix}.$$

Find an orthogonal 2×2 matrix Q such that

$$Q^* A Q = \begin{bmatrix} \lambda_2 & \beta \\ 0 & \lambda_1 \end{bmatrix}.$$

Hint: the first column of Q must be the (normalized) eigenvector of A with eigenvalue λ_2.

2.5 The real Schur decomposition

Real matrices can have complex eigenvalues. If complex eigenvalues exist, then they occur in *complex conjugate pairs*! That is, if λ is an eigenvalue of the real matrix A, then also $\bar{\lambda}$ is an eigenvalue of A. The following theorem indicated that complex computation can be avoided.

Theorem 2.10. ***(Real Schur decomposition)*** *If $A \in \mathbb{R}^{n\times n}$ then there is an orthogonal $Q \in \mathbb{R}^{n\times n}$ such that*

$$Q^* A Q = \begin{bmatrix} R_{11} & R_{12} & \cdots & R_{1m} \\ & R_{22} & \cdots & R_{2m} \\ & & \ddots & \vdots \\ & & & R_{mm} \end{bmatrix} \tag{2.20}$$

is upper quasi-triangular. The diagonal blocks R_{ii} are either 1×1 or 2×2 matrices. A 1×1 block corresponds to a real eigenvalue, a 2×2 block corresponds to pair of complex conjugate eigenvalues.

Remark 2.3. *The matrix*

$$\begin{bmatrix} \alpha & \beta \\ -\beta & \alpha \end{bmatrix}, \quad \alpha, \beta \in \mathbb{R},$$

has the eigenvalues $\alpha + i\beta$ and $\alpha - i\beta$.

Proof. Let $\lambda = \alpha + i\beta$, $\beta \neq 0$, be an eigenvalue of A with eigenvector $\mathbf{x} = \mathbf{u} + i\mathbf{v}$. Then $\bar{\lambda} = \alpha - i\beta$ is an eigenvalue corresponding to $\bar{\mathbf{x}} = \mathbf{u} - i\mathbf{v}$. To see this we first observe that

$$\begin{aligned} A\mathbf{x} &= A(\mathbf{u} + i\mathbf{v}) = A\mathbf{u} + iA\mathbf{v}, \\ \lambda\mathbf{x} &= (\alpha + i\beta)(\mathbf{u} + i\mathbf{v}) = (\alpha\mathbf{u} - \beta\mathbf{v}) + i(\beta\mathbf{u} - \alpha\mathbf{v}). \end{aligned}$$

Thus,

$$\begin{aligned} A\bar{\mathbf{x}} &= A(\mathbf{u} - i\mathbf{v}) = A\mathbf{u} - iA\mathbf{v}, \\ &= (\alpha\mathbf{u} - \beta\mathbf{v}) - i(\beta\mathbf{u} + \alpha\mathbf{v}) \\ &= (\alpha - i\beta)\mathbf{u} - i(\alpha - i\beta)\mathbf{v} = (\alpha - i\beta)(\mathbf{u} - i\mathbf{v}) = \bar{\lambda}\bar{\mathbf{x}}. \end{aligned}$$

Now, the actual proof starts. Let k be the number of komplex konjugate pairs. We prove the theorem by induction on k.

First we consider the case $k = 0$. In this case A has real eigenvalues and eigenvectors. It is clear that we can repeat the proof of the Schur decomposition of Theorem 2.8 in real arithmetic to get the decomposition (2.17) with $U \in \mathbb{R}^{n\times n}$ and $T \in \mathbb{R}^{n\times n}$. So, there are n diagonal blocks R_{jj} in (2.20) all of which are 1×1.

Let us now assume the the theorem is true for all matrices with fewer than k complex conjugate pairs. Then, with $\lambda = \alpha + i\beta$, $\beta \neq 0$ and $\mathbf{x} = \mathbf{u} + i\mathbf{v}$, as previously, we have

$$A[\mathbf{u}, \mathbf{v}] = [\mathbf{u}, \mathbf{v}] \begin{bmatrix} \alpha & \beta \\ -\beta & \alpha \end{bmatrix}.$$

Let $\{\mathbf{x}_1, \mathbf{x}_2\}$ be an orthonormal basis of span$([\mathbf{u}, \mathbf{v}])$. Then, since $\mathbf{u}$ and $\mathbf{v}$ are linearly independent, there is a nonsingular 2×2 real square matrix C with

$$[\mathbf{x}_1, \mathbf{x}_2] = [\mathbf{u}, \mathbf{v}]C.$$

Now,

$$\begin{aligned} A[\mathbf{x}_1, \mathbf{x}_2] &= A[\mathbf{u}, \mathbf{v}]C = A[\mathbf{u}, \mathbf{v}] \begin{bmatrix} \alpha & \beta \\ -\beta & \alpha \end{bmatrix} C \\ &= [\mathbf{x}_1, \mathbf{x}_2]C^{-1} \begin{bmatrix} \alpha & \beta \\ -\beta & \alpha \end{bmatrix} C =: [\mathbf{x}_1, \mathbf{x}_2]S. \end{aligned}$$

S and $\begin{bmatrix} \alpha & \beta \\ -\beta & \alpha \end{bmatrix}$ are similar and therefore have equal eigenvalues. Now we construct an orthogonal matrix $[\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots, \mathbf{x}_n] =: [\mathbf{x}_1, \mathbf{x}_2, W]$. Then

$$[[\mathbf{x}_1, \mathbf{x}_2], W]^T A[[\mathbf{x}_1, \mathbf{x}_2], W] = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ W^T \end{bmatrix} [[\mathbf{x}_1, \mathbf{x}_2]S, AW] = \begin{bmatrix} S & [\mathbf{x}_1, \mathbf{x}_2]^T AW \\ O & W^T AW \end{bmatrix}.$$

The matrix $W^T AW$ has less than k complex-conjugate eigenvalue pairs. Therfore, by the induction assumption, there is an orthogonal $Q_2 \in \mathbb{R}^{(n-2)\times(n-2)}$ such that the matrix

$$Q_2^T (W^T AW) Q_2$$

is quasi-triangular. Thus, the orthogonal matrix

$$Q = [\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots, \mathbf{x}_n] \begin{pmatrix} I_2 & O \\ O & Q_2 \end{pmatrix}$$

transforms A similarly to quasi-triangular form. ■

2.6 Hermitian matrices

Definition 2.11 A matrix $A \in \mathbb{F}^{n\times n}$ is *Hermitian* if

$$A = A^*. \tag{2.21}$$

The Schur decomposition for Hermitian matrices is particularly simple. We first note that A being Hermitian implies that the upper triangular Λ in the Schur decomposition $A = U\Lambda U^*$ is Hermitian and thus diagonal. In fact, because

$$\overline{\Lambda} = \Lambda^* = (U^*AU)^* = U^*A^*U = U^*AU = \Lambda,$$

each diagonal element λ_i of Λ satisfies $\overline{\lambda}_i = \lambda_i$. So, Λ has to be *real*. In summary have the following result.

Theorem 2.12. ***(Spectral theorem for Hermitian matrices)*** *Let A be Hermitian. Then there is a unitary matrix U and a real diagonal matrix Λ such that*

$$A = U\Lambda U^* = \sum_{i=1}^{n} \lambda_i \mathbf{u}_i \mathbf{u}_i^*. \tag{2.22}$$

The columns $\mathbf{u}_1, \ldots, \mathbf{u}_n$ of U are eigenvectors corresponding to the eigenvalues $\lambda_1, \ldots, \lambda_n$. They form an orthonormal basis for $\mathbb{F}^n$.

The decomposition (2.22) is called a *spectral decomposition* of A.

As the eigenvalues are real we can sort them with respect to their magnitude. We can, e.g., arrange them in ascending order such that $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$.

If $\lambda_i = \lambda_j$, then any nonzero linear combination of $\mathbf{u}_i$ and $\mathbf{u}_j$ is an eigenvector corresponding to λ_i,

$$A(\mathbf{u}_i\alpha + \mathbf{u}_j\beta) = \mathbf{u}_i\lambda_i\alpha + \mathbf{u}_j\lambda_j\beta = (\mathbf{u}_i\alpha + \mathbf{u}_j\beta)\lambda_i.$$

However, eigenvectors corresponding to *different* eigenvalues are orthogonal. Let $A\mathbf{u} = \mathbf{u}\lambda$ and $A\mathbf{v} = \mathbf{v}\mu$, $\lambda \neq \mu$. Then

$$\lambda\mathbf{u}^*\mathbf{v} = (\mathbf{u}^*A)\mathbf{v} = \mathbf{u}^*(A\mathbf{v}) = \mathbf{u}^*\mathbf{v}\mu,$$

and thus

$$(\lambda - \mu)\mathbf{u}^*\mathbf{v} = 0,$$

from which we deduce $\mathbf{u}^*\mathbf{v} = 0$ as $\lambda \neq \mu$.

In summary, the eigenvectors corresponding to a particular eigenvalue λ form a subspace, the *eigenspace* $\{\mathbf{x} \in \mathbb{F}^n, A\mathbf{x} = \lambda\mathbf{x}\} = \mathcal{N}(A - \lambda I)$. They are perpendicular to the eigenvectors corresponding to all the other eigenvalues. Therefore, the spectral decomposition (2.22) is unique up to ± signs if all the eigenvalues of A are distinct. In case of multiple eigenvalues, we are free to choose any orthonormal basis for the corresponding eigenspace.

Remark 2.4. *The notion of Hermitian or symmetric has a wider background. Let $\langle \mathbf{x}, \mathbf{y} \rangle$ be an inner product on $\mathbb{F}^n$. Then a matrix A is symmetric with respect to this inner product if $\langle A\mathbf{x}, \mathbf{y} \rangle = \langle \mathbf{x}, A\mathbf{y} \rangle$ for all vectors $\mathbf{x}$ and $\mathbf{y}$. For the ordinary Euklidian inner product $(\mathbf{x}, \mathbf{y}) = \mathbf{x}^*\mathbf{y}$ we arrive at the element-wise definition 2.6 if we set $\mathbf{x}$ and $\mathbf{y}$ equal to coordinate vectors.*

It is important to note that all the properties of Hermitian matrices that we will derive subsequently hold similarly for matrices symmetric with respect to a certain inner product.

Example 2.13 We consider the one-dimensional Sturm-Liouville eigenvalue problem

$$-u''(x) = \lambda u(x), \quad 0 < x < \pi, \quad u(0) = u(\pi) = 0, \tag{2.23}$$

that models the vibration of a homogenous string of length π that is *fixed* at both ends. The eigenvalues and eigenvectors or eigenfunctions of (2.23) are

$$\lambda_k = k^2, \quad u_k(x) = \sin kx, \qquad k \in \mathbb{N}.$$

Let $u_i^{(n)}$ denote the approximation of an (eigen)function u at the grid point x_i,

$$u_i \approx u(x_i), \quad x_i = ih, \quad 0 \leq i \leq n+1, \quad h = \frac{\pi}{n+1}.$$

We approximate the second derivative of u at the *interior* grid points by

$$\frac{1}{h^2}(-u_{i-1} + 2u_i - u_{i+1}) = \lambda u_i, \qquad 1 \leq i \leq n. \tag{2.24}$$

Collecting these equation and taking into account the boundary conditions, $u_0 = 0$ and $u_{n+1} = 0$, we get a (matrix) eigenvalue problem

$$T_n\mathbf{x} = \lambda\mathbf{x} \tag{2.25}$$

where

$$T_n := \frac{(n+1)^2}{\pi^2} \begin{bmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{bmatrix} \in \mathbb{R}^{n \times n}.$$

The matrix eigenvalue problem (2.25) can be solved explicitly [Zur64, p.229]. Eigenvalues and eigenvectors are given by

$$\begin{aligned}\lambda_k^{(n)} &= \frac{(n+1)^2}{\pi^2}(2-2\cos\phi_k) = \frac{4(n+1)^2}{\pi^2}\sin^2\frac{k\pi}{2(n+1)},\\ \mathbf{u}_k^{(n)} &= \left(\frac{2}{n+1}\right)^{1/2}[\sin\phi_k, \sin 2\phi_k, \ldots, \sin n\phi_k]^T, \qquad \phi_k = \frac{k\pi}{n+1}.\end{aligned} \tag{2.26}$$

Clearly, $\lambda_k^{(n)}$ converges to λ_k as $n \to \infty$. (Note that $\sin\xi \to \xi$ as $\xi \to 0$.) When we identify $\mathbf{u}_k^{(n)}$ with the piecewise linear function that takes on the values given by $\mathbf{u}_k^{(n)}$ at the grid points x_i then this function evidently converges to $\sin\, kx$.

Let $p(\lambda)$ be a polynomial of degree d, $p(\lambda) = \alpha_0 + \alpha_1\lambda + \alpha_2\lambda^2 + \cdots + \alpha_d\lambda^d$. As $A^j = (U\Lambda U^*)^j = U\Lambda^j U^*$ we can define a *matrix polynomial* as

$$p(A) = \sum_{j=0}^{d}\alpha_j A^j = \sum_{j=0}^{d}\alpha_j U\Lambda^j U^* = U\left(\sum_{j=0}^{d}\alpha_j\Lambda^j\right)U^*. \tag{2.27}$$

This equation shows that $p(A)$ has the same eigenvectors as the original matrix A. The eigenvalues are modified though, λ_k becomes $p(\lambda_k)$. We can imagine how more complicated functions of A can be computed if only the function is defined on the set of eigenvalues of A.

Definition 2.14 The quotient

$$\rho(\mathbf{x}) := \frac{\mathbf{x}^*A\mathbf{x}}{\mathbf{x}^*\mathbf{x}}, \qquad \mathbf{x} \neq \mathbf{0},$$

is called the *Rayleigh quotient* of A at $\mathbf{x}$.

Notice, that $\rho(\mathbf{x}\alpha) = \rho(\mathbf{x})$, $\alpha \neq 0$. So, the properties of the Rayleigh quotient can be investigated by just looking at its values on the unit sphere. Using the spectral decomposition $A = U\Lambda U$, we get

$$\mathbf{x}^*A\mathbf{x} = \sum_{i,j=1}^{n}\overline{(\mathbf{u}_i^*\mathbf{x})}(\mathbf{u}_j^*\mathbf{x})\mathbf{u}_i A\mathbf{u}_j = \sum_{i=1}^{n}\lambda_i|\mathbf{u}_i^*\mathbf{x}|^2.$$

Similarly, $\mathbf{x}^*\mathbf{x} = \sum_{i=1}^{n}|\mathbf{u}_i^*\mathbf{x}|^2$. With $\lambda_1 \le \lambda_2 \le \cdots \le \lambda_n$, we have

$$\lambda_1\sum_{i=1}^{n}|\mathbf{u}_i^*\mathbf{x}|^2 \le \sum_{i=1}^{n}\lambda_i|\mathbf{u}_i^*\mathbf{x}|^2 \le \lambda_n\sum_{i=1}^{n}|\mathbf{u}_i^*\mathbf{x}|^2.$$

So,

$$\lambda_1 \le \rho(\mathbf{x}) \le \lambda_n, \qquad \text{for all } \mathbf{x} \neq \mathbf{0}.$$

As

$$\rho(\mathbf{u}_k) = \lambda_k,$$

the extremal values λ_1 and λ_n are actually attained for $\mathbf{x} = \mathbf{u}_1$ and $\mathbf{x} = \mathbf{u}_n$, respectively. Thus we have proved the following theorem.

Theorem 2.15. *Let A be Hermitian. Then the Rayleigh quotient satisfies*

$$\lambda_1 = \min\rho(\mathbf{x}), \qquad \lambda_n = \max\rho(\mathbf{x}). \tag{2.28}$$

As the Rayleigh quotient is a continuous function it attaines *all* values in the closed interval $[\lambda_1, \lambda_n]$.

The next theorem generalizes the above theorem to interior eigenvalues. The following theorems is adhered to Poincaré, Fischer and Pólya.

Theorem 2.16. ***(Minimum-maximum principle)*** *Let A be Hermitian. Then*

$$\lambda_p = \min_{X\in\mathbb{F}^{n\times p},\mathrm{rank}(X)=p} \max_{\mathbf{x}\neq\mathbf{0}} \rho(X\mathbf{x}) \tag{2.29}$$

Proof. Let $U_{p-1} = [\mathbf{u}_1, \ldots, \mathbf{u}_{p-1}]$. For every X with full rank we can choose $\mathbf{x} \neq \mathbf{0}$ such that $U_{p-1}^* X\mathbf{x} = \mathbf{0}$. Then $\mathbf{0} \neq \mathbf{z} := X\mathbf{x} = \sum_{i=p}^n z_i \mathbf{u}_i$. As in the proof of the previous theorem we obtain the inequality

$$\rho(\mathbf{z}) \geq \lambda_p.$$

To prove that equality holds in (2.29) we choose $X = [\mathbf{u}_1, \ldots, \mathbf{u}_p]$. Then

$$U_{p-1}^* X\mathbf{x} = \begin{bmatrix} 1 & & & 0 \\ & \ddots & & \vdots \\ & & 1 & 0 \end{bmatrix} \mathbf{x} = \mathbf{0}$$

implies that $\mathbf{x} = \mathbf{e}_p$, i.e., that $\mathbf{z} = X\mathbf{x} = \mathbf{u}_p$. So, $\rho(\mathbf{z}) = \lambda_p$. ■

The *trace* of a matrix $A \in \mathbb{F}^{n\times n}$ is defined to be the sum of the diagonal elements of a matrix. Similar matrices have equal traces. So, by the spectral theorem,

$$\mathrm{trace}(A) = \sum_{i=1}^n a_{ii} = \sum_{i=1}^n \lambda_i. \tag{2.30}$$

The following theorem is proved in a similar way as the minimum-maximum theorem.

Theorem 2.17. ***(Trace theorem)***

$$\lambda_1 + \lambda_2 + \cdots + \lambda_p = \min_{X\in\mathbb{F}^{n\times p}, X^*X=I_p} \textit{trace}(X^*AX) \tag{2.31}$$

2.7 Cholesky factorization

Definition 2.18 A Hermitian matrix is called **positive definite** (**positive semi-definite**) if all its eigenvalues are positive (nonnegative).

We have the following

Theorem 2.19. ***(Cholesky factorization)*** *Let $A \in \mathbb{F}^{n\times n}$ be positive definite. Then there is a* lower triangular *matrix L such that*

$$A = LL^*. \tag{2.32}$$

L is unique if we choose its diagonal elements to be positive.

Proof. We prove the Theorem by giving an algorithm that computes the desired factorization.

Since A is positive definite, we have $a_{11} = \mathbf{e}_1^* A \mathbf{e}_1 > 0$. Therefore we can form the matrix

$$L_1 = \begin{bmatrix} l_{11}^{(1)} & & & \\ l_{21}^{(1)} & 1 & & \\ \vdots & & \ddots & \\ l_{n1}^{(1)} & & & 1 \end{bmatrix} = \begin{bmatrix} \sqrt{a_{11}} & & & \\ \frac{a_{21}}{\sqrt{a_{1,1}}} & 1 & & \\ \vdots & & \ddots & \\ \frac{a_{n1}}{\sqrt{a_{1,1}}} & & & 1 \end{bmatrix}.$$

We now form the matrix

$$A_1 = L_1^{-1} A L_1^{-1^*} = \begin{bmatrix} 1 & 0 & \ldots & 0 \\ 0 & a_{22} - \frac{a_{21}a_{12}}{a_{11}} & \ldots & a_{2n} - \frac{a_{21}a_{1n}}{a_{11}} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2} - \frac{a_{n1}a_{12}}{a_{11}} & \ldots & a_{nn} - \frac{a_{n1}a_{1n}}{a_{11}} \end{bmatrix}.$$

This is the first step of the algorithm. Since (evidently) A_1 is again positive definite, we can proceed in a similar fashion factoring $A_1(2:n, 2:n)$, etc.

Collecting $L_1, L_2, \ldots$, we get

$$I = L_n^{-1} \cdots L_2^{-1} L_1^{-1} A (L_1^*)^{-1} (L_2^*)^{-1} \cdots (L_n^*)^{-1}$$

or

$$(L_1 L_2 \cdots L_n)(L_n^* \cdots L_2^* L_1^*) = A.$$

which is the desired result. It is easy to see that $L_1 L_2 \cdots L_n$ is a triangular matrix and that

$$L_1 L_2 \cdots L_n = \begin{bmatrix} l_{11}^{(1)} & & & & \\ l_{21}^{(1)} & l_{22}^{(2)} & & & \\ l_{31}^{(1)} & l_{32}^{(2)} & l_{33}^{(3)} & & \\ \vdots & \vdots & \vdots & \ddots & \\ l_{n1}^{(1)} & l_{n2}^{(2)} & l_{n3}^{(3)} & \ldots & l_{nn}^{(n)} \end{bmatrix}$$

■

Remark 2.5. *When working with symmetric matrices, one often stores only half of the matrix, e.g. the lower triangle consiting of all elements including and below the diagonal. The L-factor of the Cholesky factorization can overwrite this information to save memory.*

Definition 2.20 The **inertia** of a Hermitian matrix is the triple (ν, ζ, π) where ν, ζ, π is the number of negative, zero, and positive eigenvalues.

Theorem 2.21. ***(Sylvester's law of inertia)*** *If $A \in \mathbb{R}^{n\times n}$ is symmetric and $X \in \mathbb{R}^{n\times n}$ is nonsingular, the A and $X^T AX$ have the smae inertia.*

Proof. The proof is given in [GvL89]. ■

Remark 2.6. *Matrices A and $B = X^T AX$, X nonsingular, are called congruent. Thus Sylvester's law of inertia can be stated in the form:* The inertia is invariant under congruence transformations.

2.8 The singular value decomposition (SVD)*

Theorem 2.22. ***(Singular value decomposition)*** *If $A \in \mathbb{C}^{m\times n}$ then there exist unitary matrices $U \in \mathbb{C}^{m\times m}$ and $V \in \mathbb{C}^{n\times n}$ such that*

$$U^* AV = \Sigma = \text{diag}(\sigma_1, \ldots, \sigma_p), \qquad p = \min(m, n), \tag{2.33}$$

where $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0$.

Proof. If $A = O$, then the theorem holds with U the $m \times m$ and V the $n \times n$ identity matrix and Σ equal to the $m \times n$ zero matrix.

We now assume, that $A \neq O$. Let $\mathbf{x}$, $\|\mathbf{x}\| = 1$, be a maximizing vector of $\|A\mathbf{x}\|$ and let $A\mathbf{x} = \sigma\mathbf{y}$ where $\sigma = \|A\| = \|A\mathbf{x}\|$ and $\|\mathbf{y}\| = 1$. As $A \neq O$, $\sigma > 0$. Consider the scalar function

$$f(\alpha) := \frac{\|A(\mathbf{x} + \alpha\mathbf{y})\|^2}{\|\mathbf{x} + \alpha\mathbf{y}\|^2} = \frac{(\mathbf{x} + \alpha\mathbf{y})^* A^* A(\mathbf{x} + \alpha\mathbf{y})}{(\mathbf{x} + \alpha\mathbf{y})^*(\mathbf{x} + \alpha\mathbf{y})}$$

Because of the extremality of $A\mathbf{x}$, the derivative $f'(0)$ of $f(\alpha)$ must vanish in $\alpha = 0$. This *for all* $\mathbf{y}$! We have

$$\frac{df}{d\alpha}(\alpha) = \frac{(\mathbf{x}^* A^* A\mathbf{y} + \bar{\alpha}\mathbf{y}^* A^* A\mathbf{y})\|\mathbf{x} + \alpha\mathbf{y}\|^2 - (\mathbf{x}^*\mathbf{y} + \bar{\alpha}\mathbf{y}^*\mathbf{y})\|A(\mathbf{x} + \alpha\mathbf{y})\|^2}{\|\mathbf{x} + \alpha\mathbf{y}\|^4}$$

Thus, we must have for all $\mathbf{y}$,

$$\frac{df}{d\alpha}(\alpha)\bigg|_{\alpha=0} = \frac{\mathbf{x}^* A^* A\mathbf{y}\|\mathbf{x}\|^2 - \mathbf{x}^*\mathbf{y}\|A(\mathbf{x})\|^2}{\|\mathbf{x}\|^4} = 0.$$

As $\|\mathbf{x}\| = 1$ and $\|A\mathbf{x}\| = \sigma$, we have

$$(\mathbf{x}^* A^* A - \sigma^2\mathbf{x}^*)\mathbf{y} = (A^* A\mathbf{x} - \sigma^2\mathbf{x})^*\mathbf{y} = 0, \qquad \text{for all } \mathbf{y},$$

from which

$$A^* A\mathbf{x} = \sigma^2\mathbf{x}$$

follows. Multiplying $A\mathbf{x} = \sigma\mathbf{y}$ from the left by A^* we get $A^* A\mathbf{x} = \sigma A^*\mathbf{y} = \sigma^2\mathbf{x}$ from which

$$A^*\mathbf{y} = \sigma\mathbf{x}$$

and $AA^*\mathbf{y} = \sigma A\mathbf{x} = \sigma^2\mathbf{y}$ follows. Therefore, $\mathbf{x}$ is an eigenvector of $A^* A$ corresponding to the eigenvalue σ^2 and $\mathbf{y}$ is an eigenvector of AA^* corresponding to the same eigenvalue.

Now, we construct a unitary matrix U_1 with first column $\mathbf{y}$ and a unitary matrix V_1 with first column $\mathbf{x}$, $U_1 = [\mathbf{y}, \bar{U}]$ and $V_1 = [\mathbf{x}, \bar{V}]$. Then

$$U_1^* A V_1 = \begin{bmatrix} \mathbf{y}^* A\mathbf{x} & \mathbf{y}^* A\overline{U} \\ \overline{U}^* A\mathbf{x} & \overline{U}^* A\overline{V} \end{bmatrix} = \begin{bmatrix} \sigma & \sigma\mathbf{x}^*\overline{U} \\ \sigma\overline{U}^*\mathbf{y} & \overline{U}^* A\overline{V} \end{bmatrix} = \begin{bmatrix} \sigma & \mathbf{0}^* \\ \mathbf{0} & \hat{A} \end{bmatrix}$$

where $\hat{A} = \overline{U}^* A\overline{V}$. ■

This proof is due to W. Gragg. It nicely shows the relation of the singular value decomposition with the spectral decomposition of the Hermitian matrices $A^* A$ and AA^*,

$$A = U\Sigma V^* \quad \Longrightarrow \quad A^* A = U\Sigma^2 U^*, \qquad AA^* = V\Sigma^2 V^*. \tag{2.34}$$

The proof given in [GvL89] is shorter and maybe more elegant.

The SVD of dense matrices is computed in a way thet is very similar to the dense Hermitian eigenvalue problem.

Let us consider the $(n+m)\times(n+m)$ Hermitian matrix

$$\begin{bmatrix} O & A \\ A^* & O \end{bmatrix}. \tag{2.35}$$

Making use of the SVD (2.33) we immediately get

$$\begin{bmatrix} O & A \\ A^* & O \end{bmatrix} = \begin{bmatrix} U & O \\ O & V \end{bmatrix} \begin{bmatrix} O & \Sigma \\ \Sigma^T & O \end{bmatrix} \begin{bmatrix} U^* & O \\ O & V^* \end{bmatrix}.$$

Now, let us assume that $m \geq n$. Then we write $U = [U_1, U_2]$ where $U_1 \in \mathbb{F}^{m\times n}$ and $\Sigma = \begin{bmatrix} \Sigma_1 \\ O \end{bmatrix}$ with $\Sigma_1 \in \mathbb{R}^{n\times n}$. Then

$$\begin{bmatrix} O & A \\ A^* & O \end{bmatrix} = \begin{bmatrix} U_1 & U_2 & O \\ O & O & V \end{bmatrix} \begin{bmatrix} O & O & \Sigma_1 \\ O & O & O \\ \Sigma_1 & O & O \end{bmatrix} \begin{bmatrix} U_1^* & O \\ U_2^* & O \\ O & V^* \end{bmatrix} = \begin{bmatrix} U_1 & O & U_2 \\ O & V & O \end{bmatrix} \begin{bmatrix} O & \Sigma_1 & O \\ \Sigma_1 & O & O \\ O & O & O \end{bmatrix} \begin{bmatrix} U_1^* & O \\ O & V^* \\ U_2^* & O \end{bmatrix}.$$

The first and third diagonal zero blocks have order n. The middle diagonal block has order $n - m$. Now we emply the fact that

$$\begin{bmatrix} 0 & \sigma \\ \sigma & 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \sigma & 0 \\ 0 & -\sigma \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

to obtain

$$\begin{bmatrix} O & A \\ A^* & O \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}}U_1 & \frac{1}{\sqrt{2}}U_1 & U_2 \\ \frac{1}{\sqrt{2}}V & -\frac{1}{\sqrt{2}}V & O \end{bmatrix} \begin{bmatrix} \Sigma_1 & O & O \\ O & -\Sigma_1 & O \\ O & O & O \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}}U_1^* & \frac{1}{\sqrt{2}}V^* \\ \frac{1}{\sqrt{2}}U_1^* & -\frac{1}{\sqrt{2}}V^* \\ U_2^* & O \end{bmatrix}. \tag{2.36}$$

Thus, there are three ways how to treat the computation of the singular value decomposition as an eigenvalue problem. One of the two forms in (2.34) is used *implicitly* in the QR algorithm for dense matrices A, see [GvL89],[ABB$^+$94]. The form (2.35) is suited if A is a sparse matrix.

Bibliography

[ABB^{+}94] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide - Release 2.0*. SIAM, Philadelphia, PA, 1994. (Software and guide are available from Netlib at URL `http://www.netlib.org/lapack/`).

[GvL89] G. H. Golub and C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 2nd edition, 1989.

[Zur64] R. Zurmühl. *Matrizen und ihre technischen Anwendungen*. Springer, Berlin, 4th edition, 1964.

Chapter 3

The QR Algorithm

The QR algorithm computes the Schur decomposition of a matrix. It is quite certainly the most important algorithm in eigenvalue computations. However it is applied to *dense* (or *full*) matrices only.

The QR algorithm consists of two separate stages. First the original matrix is similarly transformed in a finite number of steps into Hessenberg or in the Hermitian/symmetric case into real tridiagonal form. In order to retain the information on eigenvalues and vectors this is done by similarity transformations. This first stage of the algorithm prepares its second stage, the actual QR iteration that is applied to the Hessenberg or tridiagonal matrix. The overall complexity (number of floating points) of the algorithm is $\mathcal{O}(n^3)$ which we will see is not entirely trivial to obtain.

The main limit of the QR algorithm is first of all the property of the first stage that fills general sparse matrices. So, it cannot be applied to sparse matrices because of its memory requirements. However, the QR algorithm computes all eigenvalues (and eventually vectors) which is not desired in sparse matrix computations anyway.

The treatment of the QR algorithm in these lecture notes on large scale eigenvalue computation is justified in two respects. First, there are of course large or even huge *dense* eigenvalue problems. Second, the QR algorithm is employed in most other algorithms to solve 'interior' small auxiliary eigenvalue problems.

3.1 The basic QR algorithm

In 1958 Rutishauser [Rut58] of ETH Zürich experimented with a similar algorithm that we are going to present, but based on the LR factorization, i.e., based on Gaussian elimination without pivoting. That algorithm was not successful as the LR factorization (nowadays called LU factorization) is not stable without pivoting. Francis [Fra61, Fra62] noticed that the QR factorization would do and devised the QR algorithm with all bells and whistles as it is used nowadays.

Here we start with a basic iteration, given in Algorithm 3.1, discuss its properties and improve on it step by step until we arrive at Francis' algorithm.

```
Let $A \in \mathbb{C}^{n\times n}$. This algorithm computes an upper triangular matrix $T$ and a unitary matrix $U$ such
  that $A = UTU^*$ is the Schur decomposition of $A$.
Set $A_0 := A$ and $U_0 = I$.
**for** k=1,2,... **do**
    $A_{k-1} =: Q_k R_k$; {QR factorization}
    $A_k := R_k Q_k$;
    $U_k := U_{k-1} Q_k$; {Update transformation matrix}
**end for**
Set $T := A_\infty$ and $U := U_\infty$.
```

ALGORITHM 3.1: **Basic QR algorithm**

We notice first that

$$A_k = R_k Q_k = Q_k^* A_{k-1} Q_k. \tag{3.1}$$

so that A_k and A_{k-1} are unitarily similar. The matrix sequence $\{A_k\}$ converges (under certain assumptions) towards an upper triangular matrix [Wil65]. If we assume that the eigenvalues are numbered such that $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$, then the elements of A_k below the diagonal behave like

$$|a_{ij}^{(k)}| = \mathcal{O}(|\frac{\lambda_i}{\lambda_j}|), \qquad i > j. \tag{3.2}$$

From (3.1) we see that

$$A_k = Q_k^* A_{k-1} Q_k = Q_k^* Q_{k-1}^* A_{k-2} Q_{k-1} Q_k = \cdots = Q_k^* \cdots Q_1^* A_0 \underbrace{Q_1 \cdots Q_k}_{U_k}. \tag{3.3}$$

With the same assumption on the eigenvalues, A_k tends to an upper triangular and U_k converges to the matrix of Schur vectors.

3.1.1 Numerical experiments

We conduct two MATLAB experiments to illustrate the convergence rate given in (3.2). To that end, we construct a random 4×4 matrix with eigenvalues 1, 2, 3, and 4.

```
D = diag([4 3 2 1]);
rand('seed',0);
format short e
S=rand(4); S = (S - .5)*2;
A = S*D/S      % A_0 = A = S*D*S^{-1}
for i=1:20,
   [Q,R] = qr(A);  A = R*Q
end
```

This yields the matrix sequence

```
A( 0) =  [ -4.4529e-01   4.9063e+00  -8.7871e-01   6.3036e+00]
         [ -6.3941e+00   1.3354e+01   1.6668e+00   1.1945e+01]
         [  3.6842e+00  -6.6617e+00  -6.0021e-02  -7.0043e+00]
         [  3.1209e+00  -5.2052e+00  -1.4130e+00  -2.8484e+00]

A( 1) =  [  5.9284e+00   1.6107e+00   9.3153e-01  -2.2056e+01]
         [ -1.5294e+00   1.8630e+00   2.0428e+00   6.5900e+00]
         [  1.9850e-01   2.5660e-01   1.7088e+00   1.2184e+00]
         [  2.4815e-01   1.5265e-01   2.6924e-01   4.9975e-01]

A( 2) =  [  4.7396e+00   1.4907e+00  -2.1236e+00   2.3126e+01]
         [ -4.3101e-01   2.4307e+00   2.2544e+00  -8.2867e-01]
         [  1.2803e-01   2.4287e-01   1.6398e+00  -1.8290e+00]
         [ -4.8467e-02  -5.8164e-02  -1.0994e-01   1.1899e+00]

A( 3) =  [  4.3289e+00   1.0890e+00  -3.9478e+00  -2.2903e+01]
         [ -1.8396e-01   2.7053e+00   1.9060e+00  -1.2062e+00]
         [  6.7951e-02   1.7100e-01   1.6852e+00   2.5267e+00]
         [  1.3063e-02   2.2630e-02   7.9186e-02   1.2805e+00]

A( 4) =  [  4.1561e+00   7.6418e-01  -5.1996e+00   2.2582e+01]
         [ -9.4175e-02   2.8361e+00   1.5788e+00   2.0983e+00]
         [  3.5094e-02   1.1515e-01   1.7894e+00  -2.9819e+00]
         [ -3.6770e-03  -8.7212e-03  -5.7793e-02   1.2184e+00]

A( 5) =  [  4.0763e+00   5.2922e-01  -6.0126e+00  -2.2323e+01]
         [ -5.3950e-02   2.9035e+00   1.3379e+00  -2.5358e+00]
         [  1.7929e-02   7.7393e-02   1.8830e+00   3.2484e+00]
```

```
          [  1.0063e-03   3.2290e-03   3.7175e-02   1.1372e+00]

A( 6) =   [  4.0378e+00   3.6496e-01  -6.4924e+00   2.2149e+01]
          [ -3.3454e-02   2.9408e+00   1.1769e+00   2.7694e+00]
          [  9.1029e-03   5.2173e-02   1.9441e+00  -3.4025e+00]
          [ -2.6599e-04  -1.1503e-03  -2.1396e-02   1.0773e+00]

A( 7) =   [  4.0189e+00   2.5201e-01  -6.7556e+00  -2.2045e+01]
          [ -2.1974e-02   2.9627e+00   1.0736e+00  -2.9048e+00]
          [  4.6025e-03   3.5200e-02   1.9773e+00   3.4935e+00]
          [  6.8584e-05   3.9885e-04   1.1481e-02   1.0411e+00]

A( 8) =   [  4.0095e+00   1.7516e-01  -6.8941e+00   2.1985e+01]
          [ -1.5044e-02   2.9761e+00   1.0076e+00   2.9898e+00]
          [  2.3199e-03   2.3720e-02   1.9932e+00  -3.5486e+00]
          [ -1.7427e-05  -1.3602e-04  -5.9304e-03   1.0212e+00]

A( 9) =   [  4.0048e+00   1.2329e-01  -6.9655e+00  -2.1951e+01]
          [ -1.0606e-02   2.9845e+00   9.6487e-01  -3.0469e+00]
          [  1.1666e-03   1.5951e-02   1.9999e+00   3.5827e+00]
          [  4.3933e-06   4.5944e-05   3.0054e-03   1.0108e+00]

A(10) =   [  4.0024e+00   8.8499e-02  -7.0021e+00   2.1931e+01]
          [ -7.6291e-03   2.9899e+00   9.3652e-01   3.0873e+00]
          [  5.8564e-04   1.0704e-02   2.0023e+00  -3.6041e+00]
          [ -1.1030e-06  -1.5433e-05  -1.5097e-03   1.0054e+00]

A(11) =   [  4.0013e+00   6.5271e-02  -7.0210e+00  -2.1920e+01]
          [ -5.5640e-03   2.9933e+00   9.1729e-01  -3.1169e+00]
          [  2.9364e-04   7.1703e-03   2.0027e+00   3.6177e+00]
          [  2.7633e-07   5.1681e-06   7.5547e-04   1.0027e+00]

A(12) =   [  4.0007e+00   4.9824e-02  -7.0308e+00   2.1912e+01]
          [ -4.0958e-03   2.9956e+00   9.0396e-01   3.1390e+00]
          [  1.4710e-04   4.7964e-03   2.0024e+00  -3.6265e+00]
          [ -6.9154e-08  -1.7274e-06  -3.7751e-04   1.0014e+00]

A(13) =   [  4.0003e+00   3.9586e-02  -7.0360e+00  -2.1908e+01]
          [ -3.0339e-03   2.9971e+00   8.9458e-01  -3.1558e+00]
          [  7.3645e-05   3.2052e-03   2.0019e+00   3.6322e+00]
          [  1.7298e-08   5.7677e-07   1.8857e-04   1.0007e+00]

A(14) =   [  4.0002e+00   3.2819e-02  -7.0388e+00   2.1905e+01]
          [ -2.2566e-03   2.9981e+00   8.8788e-01   3.1686e+00]
          [  3.6855e-05   2.1402e-03   2.0014e+00  -3.6359e+00]
          [ -4.3255e-09  -1.9245e-07  -9.4197e-05   1.0003e+00]

A(15) =   [  4.0001e+00   2.8358e-02  -7.0404e+00  -2.1902e+01]
          [ -1.6832e-03   2.9987e+00   8.8305e-01  -3.1784e+00]
          [  1.8438e-05   1.4284e-03   2.0010e+00   3.6383e+00]
          [  1.0815e-09   6.4192e-08   4.7062e-05   1.0002e+00]

A(16) =   [  4.0001e+00   2.5426e-02  -7.0413e+00   2.1901e+01]
          [ -1.2577e-03   2.9991e+00   8.7953e-01   3.1859e+00]
          [  9.2228e-06   9.5295e-04   2.0007e+00  -3.6399e+00]
          [ -2.7039e-10  -2.1406e-08  -2.3517e-05   1.0001e+00]

A(17) =   [  4.0000e+00   2.3503e-02  -7.0418e+00  -2.1900e+01]
          [ -9.4099e-04   2.9994e+00   8.7697e-01  -3.1917e+00]
          [  4.6126e-06   6.3562e-04   2.0005e+00   3.6409e+00]
          [  6.7600e-11   7.1371e-09   1.1754e-05   1.0000e+00]

A(18) =   [  4.0000e+00   2.2246e-02  -7.0422e+00   2.1899e+01]
          [ -7.0459e-04   2.9996e+00   8.7508e-01   3.1960e+00]
          [  2.3067e-06   4.2388e-04   2.0003e+00  -3.6416e+00]
          [ -1.6900e-11  -2.3794e-09  -5.8750e-06   1.0000e+00]
```

```
A(19) =  [  4.0000e+00   2.1427e-02  -7.0424e+00  -2.1898e+01]
         [ -5.2787e-04   2.9997e+00   8.7369e-01  -3.1994e+00]
         [  1.1535e-06   2.8265e-04   2.0002e+00   3.6421e+00]
         [  4.2251e-12   7.9321e-10   2.9369e-06   1.0000e+00]

A(20) =  [  4.0000e+00   2.0896e-02  -7.0425e+00   2.1898e+01]
         [ -3.9562e-04   2.9998e+00   8.7266e-01   3.2019e+00]
         [  5.7679e-07   1.8846e-04   2.0002e+00  -3.6424e+00]
         [ -1.0563e-12  -2.6442e-10  -1.4682e-06   1.0000e+00]
```

Looking at the element-wise quotients of the last two matrices one recognizes the convergence rates claimed in (3.2).

```
A(20)./A(19) =  [  1.0000   0.9752   1.0000  -1.0000]
                [  0.7495   1.0000   0.9988  -1.0008]
                [  0.5000   0.6668   1.0000  -1.0001]
                [ -0.2500  -0.3334  -0.4999   1.0000]
```

The elements above and on the diagonal are relatively stable.

If we run the same little MATLAB script but with the initial diagonal matrix D replaced by

```
D = diag([5 2 2 1]);
```

then we obtain

```
A(19) =  [  5.0000e+00   4.0172e+00  -9.7427e+00  -3.3483e+01]
         [ -4.2800e-08   2.0000e+00   2.1100e-05  -4.3247e+00]
         [  1.3027e-08   7.0605e-08   2.0000e+00   2.1769e+00]
         [  8.0101e-14  -2.4420e-08   4.8467e-06   1.0000e+00]

A(20) =  [  5.0000e+00   4.0172e+00  -9.7427e+00   3.3483e+01]
         [ -1.7120e-08   2.0000e+00   1.0536e-05   4.3247e+00]
         [  5.2106e-09   3.3558e-08   2.0000e+00  -2.1769e+00]
         [ -1.6020e-14   1.2210e-08  -2.4234e-06   1.0000e+00]
```

So, again the eigenvalues are visible on the diagonal of A_{20}. The element-wise quotients of A_{20} relative to A_{19} are

```
A(20)./A(19) =  [  1.0000   1.0000   1.0000  -1.0000]
                [  0.4000   1.0000   0.4993  -1.0000]
                [  0.4000   0.4753   1.0000  -1.0000]
                [ -0.2000  -0.5000  -0.5000   1.0000]
```

Notice that (3.2) does not state a rate for the element at position $(3, 2)$.

These little numerical tests are intended to demonstrate that the convergence rates given in (3.2) are in fact seen in a real run of the basic QR algorithm. The conclusions we draw are the following:

1. The convergence of the algorithm is slow. In fact it can be arbitrary slow if eigenvalues are very close to each other.

2. The algorithm is expensive. Each iteration step requires the computation of the QR factorization of a full $n \times n$ matrix, i.e., each single iteration step has a complexity $\mathcal{O}(n^3)$. Even if we assume that the number of steps is proportional to n we would get an $\mathcal{O}(n^4)$ complexity. The latter assumption is not assured by point 1 of these conclusions.

In the following we want to improve on both issues. First we want to find a matrix structure that is preserved by the QR algorithm and that lowers the cost of a single iteration step. Then, we want to improve on the convergence properties of the algorithm.

3.2 The Hessenberg QR algorithm

A matrix structure that is close to upper triangular form and that is preserved by the QR algorithm is the Hessenberg form.

Definition 3.1 A matrix H is a Hessenberg matrix if its elements below the lower off-diagonal are zero,

$$h_{ij} = 0, \qquad i > j + 1.$$

Theorem 3.2. *The Hessenberg form is preserved by the QR algorithms.*

Proof. We give a constructive proof, i.e., given a Hessenberg matrix H with QR factorization $H = QR$, we show that $\overline{H} = RQ$ is again a Hessenberg matrix.

The **Givens rotation** $G(i, j, \vartheta)$ is defined by

$$G(i,j,\vartheta) := \begin{array}{cc} \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} & \begin{array}{c} \\ \\ \leftarrow i \\ \\ \leftarrow j \\ \\ \\ \end{array} \\ \begin{array}{ccccccc} & & \uparrow & & \uparrow & & \\ & & i & & j & & \end{array} & \end{array} \tag{3.4}$$

where $c = \cos(\vartheta)$ and $s = \sin(\vartheta)$. Pre-multiplication by $G(i, j, \vartheta)$ amounts to a counterclockwise rotation by ϑ radians in the (i, j) coordinate plane. Clearly, a Givens rotation is an orthogonal matrix. For a unitary version see [Dem97]. If $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y} = G(i, j, \vartheta)^*\mathbf{x}$, then

$$y_k = \begin{cases} cx_i - sx_j, & k = i \\ sx_i + cx_j, & k = j \\ x_k, & k \neq i, j \end{cases}$$

We can force y_j to be zero by setting

$$c = \frac{x_i}{\sqrt{|x_i|^2 + |x_j|^2}}, \qquad s = \frac{-x_j}{\sqrt{|x_i|^2 + |x_j|^2}}. \tag{3.5}$$

Thus it is a simple matter to zero a *single specific* entry in a vector by using a Givens rotation.

Now, let us look at a Hessenberg matrix H. We can show the principle procedure by means of a 4×4 example.

$$H = \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \xrightarrow{G(1,2,\vartheta_1)^* \cdot} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix}$$

$$\xrightarrow{G(2,3,\vartheta_2)^* \cdot} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \xrightarrow{G(3,4,\vartheta_3)^* \cdot} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} = R$$

So, with $G_k = G(k, k + 1, \vartheta_k)$, we get

$$\underbrace{G_3^* G_2^* G_1^*}_{Q^*} G = R \qquad \Longleftrightarrow \qquad H = QR.$$

Multiplying Q and R in reversed order gives

$$\overline{H} = RQ = RG_1G_2G_3,$$

or, pictorially,

$$R = \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{\cdot G(1,2,\vartheta_1)} \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix}$$

$$\xrightarrow{\cdot G(2,3,\vartheta_2)} \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{\cdot G(3,4,\vartheta_1)} \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} = \overline{H}$$

More generally, if H is $n \times n$, $n-1$ Givens rotations $G_1, \ldots, G_{n-1}$ are needed to transform H to upper triangular form. Applying the rotations from the right restores the Hessenberg form. ■

3.2.1 A numerical experiment

We repeat one of the previous two MATLAB experiments

```
D = diag([4 3 2 1]);
rand('seed',0);
S=rand(4); S = (S - .5)*2;
A = S*D/S     % A_0 = A = S*D*S^{-1}
H = hess(A);  % built-in MATLAB function

for i=1:30,
   [Q,R] = qr(H);  H = R*Q
end
```

This yields the matrix sequence

```
H( 0) =  [ -4.4529e-01  -1.8641e+00  -2.8109e+00   7.2941e+00]
         [  8.0124e+00   6.2898e+00   1.2058e+01  -1.6088e+01]
         [  0.0000e+00   4.0087e-01   1.1545e+00  -3.3722e-01]
         [  0.0000e+00   0.0000e+00  -1.5744e-01   3.0010e+00]

H( 5) =  [  4.0763e+00  -2.7930e+00  -7.1102e+00   2.1826e+01]
         [  5.6860e-02   2.4389e+00  -1.2553e+00  -3.5061e+00]
         [              -2.0209e-01   2.5681e+00  -2.1805e+00]
         [                            4.3525e-02   9.1667e-01]

H(10) =  [  4.0024e+00  -6.2734e-01  -7.0227e+00  -2.1916e+01]
         [  7.6515e-03   2.9123e+00  -9.9902e-01   3.3560e+00]
         [              -8.0039e-02   2.0877e+00   3.3549e+00]
         [                           -7.1186e-04   9.9762e-01]

H(15) =  [  4.0001e+00  -1.0549e-01  -7.0411e+00   2.1902e+01]
         [  1.6833e-03   2.9889e+00  -8.9365e-01  -3.2181e+00]
         [              -1.2248e-02   2.0111e+00  -3.6032e+00]
         [                            2.0578e-05   9.9993e-01]

H(20) =  [  4.0000e+00  -3.1163e-02  -7.0425e+00  -2.1898e+01]
         [  3.9562e-04   2.9986e+00  -8.7411e-01   3.2072e+00]
         [              -1.6441e-03   2.0014e+00   3.6377e+00]
         [                           -6.3689e-07   1.0000e-00]

H(25) =  [  4.0000e+00  -2.1399e-02  -7.0428e+00   2.1897e+01]
         [  9.3764e-05   2.9998e+00  -8.7056e-01  -3.2086e+00]
         [              -2.1704e-04   2.0002e+00  -3.6423e+00]
         [                            1.9878e-08   1.0000e-00]

H(30) =  [  4.0000e+00  -2.0143e-02  -7.0429e+00  -2.1897e+01]
         [  2.2247e-05   3.0000e+00  -8.6987e-01   3.2095e+00]
```

```
                [               -2.8591e-05   2.0000e+00   3.6429e+00]
                [                            -6.2108e-10   1.0000e-00]
```

Finally we compute the element-wise quotients of the last two matrices.

```
H(30)./H(29) =  [  1.0000   0.9954   1.0000  -1.0000]
                [  0.7500   1.0000   0.9999  -1.0000]
                [           0.6667   1.0000  -1.0000]
                [                   -0.5000   1.0000]
```

Again the elements in the lower off-diagonal reflect nicely the convergence rates in (3.2).

3.2.2 Complexity

We give the algorithm for a single Hessenberg-QR-step in a MATLAB - like way, see Algorithm 3.2. By

$$H_{k:j,m:n}$$

we denote the submatrix of H consisting of rows k through j and columns m through n.

```
Let $H \in \mathbb{C}^{n\times n}$ be an upper Hessenberg matrix. This algorithm overwrites $H$ with $\overline{H} = RQ$ where $H = QR$ is the QR factorization of $H$.
for k=1,2,...,n-1 do
    /* Generate $G_k$ and then apply it: $H = G(k, k+1, \vartheta_k)^* H$ */
    $[c_k, s_k]$ := givens($H_{k,k}, H_{k+1,k}$);
    $H_{k:k+1,k:n} = \begin{bmatrix} c_k & -s_k \\ s_k & c_k \end{bmatrix} H_{k:k+1,k:n}$;
end for
for k=1,2,...,n-1 do
    /* Apply the rotations $G_k$ from the right */
    $H_{1:k+1,k:k+1} = H_{1:k+1,k:k+1} \begin{bmatrix} c_k & s_k \\ -s_k & c_k \end{bmatrix}$;
end for
```

ALGORITHM 3.2: **A Hessenberg QR step**

If we neglect the determination of the two values c_k and s_k, see (3.5). Then both loops require

$$\sum_{i=1}^{n-1} 6i = 6\frac{n(n-1)}{2} \approx 3n^2 \quad \text{flops.}$$

A **flop** is a floating point operation $(+, -, \times, /)$. We do not distinguish between them, although they may differ in there execution time on a computer. We probably also have to execute the operation $U_k := U_{k-1}Q_k$ of Algorithm 3.1. This is achieved by the loop similar to the second loop in Algorithm 3.2. Since

```
for k=1,2,...,n-1 do
    $U_{1:n,k:k+1} = U_{1:n,k:k+1} \begin{bmatrix} c_k & s_k \\ -s_k & c_k \end{bmatrix}$;
end for
```

here the whole columns of U are involved executing the loop costs

$$\sum_{i=1}^{n-1} 6n \approx 6n^2 \quad \text{flops.}$$

Altogether, a QR step with a matrix Hessenberg including the update of the unitary transformation matrix requires $12n^2$ floating point operations. This has to be set in relation to a QR step with a full matrix that costs $\frac{7}{3}n^3$. By consequence, we have gained $\mathcal{O}(n)$ operations by moving from dense to Hessenberg form. However, we may still have very slow convergence if one of the quotients $|\lambda_k|/|\lambda_{k+1}|$ is close to 1.

3.3 The Householder reduction to Hessenberg form

In the previous section we found that is a good idea to perform the QR algorithm with Hessenberg matrices instead of with full matrices. But we have not discussed how we (*similarly*) transform a full matrix into Hessenberg form. We catch up on this issue in this section.

3.3.1 Householder reflectors

Givens rotations are designed to zero a single element in a vector. Householder reflectors are more efficient if a number of elements of a vector are to be zeroed at once. Here, we follow the presentation given in [GvL89].

Definition 3.3 A matrix of the form

$$P = I - 2\mathbf{u}\mathbf{u}^*, \qquad \|\mathbf{u}\| = 1,$$

is called a **Householder reflector**.

It is easy to verify that Householder reflectors are *Hermitian* and that $P^2 = I$. From this we deduce that P is *unitary*. It is clear that we only have to store the **Householder vector $\mathbf{u}$** to be able to multiply a vector (or a matrix) with P,

$$P\mathbf{x} = \mathbf{x} - \mathbf{u}(2\mathbf{u}^*\mathbf{x}). \tag{3.6}$$

This multiplication only costs $4n$ flops where n is the length of the vectors.

A task that we repeatedly want to carry out with Householder reflectors is to transform a vector $\mathbf{x}$ on a multiple of $\mathbf{e}_1$,

$$P\mathbf{x} = \mathbf{x} - \mathbf{u}(2\mathbf{u}^*\mathbf{x}) = \alpha\mathbf{e}_1.$$

Since P is unitary, we must have $\alpha = \rho\|\mathbf{x}\|$, where $\rho \in \mathbb{C}$ has absolute value one. Therefore,

$$\mathbf{u} = \frac{\mathbf{x} - \rho\|\mathbf{x}\|\mathbf{e}_1}{\|\mathbf{x} - \rho\|\mathbf{x}\|\mathbf{e}_1\|} = \frac{1}{\|\mathbf{x} - \rho\|\mathbf{x}\|\mathbf{e}_1\|}\begin{bmatrix} x_1 - \rho\|\mathbf{x}\| \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

We can freely choose ρ provided that $|\rho| = 1$. Let $x_1 = |x_1|e^{i\phi}$. To avoid cancellation we set $\rho = -e^{i\phi}$.

In the real case, one commonly sets $\rho = -\text{sign}(x_1)$. If $x_1 = 0$ we can set ρ in any way.

3.3.2 Reduction to Hessenberg form

Now we show how to use Householder reflectors to reduce an arbitrary square matrix to Hessenberg form. We show the idea by means of a 5×5 example. In the first step of the reduction we introduce zeros in the first column below the second element,

$$A = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \xrightarrow{P_1*} \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \end{bmatrix} \xrightarrow{*P_1} \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \end{bmatrix} = P_1^*AP_1.$$

Notice that $P_1 = P_1^*$ since it is a Householder reflector! It has the structure

$$P_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \end{bmatrix} = \begin{bmatrix} 1 & \mathbf{0}^T \\ \mathbf{0} & I_4 - 2\mathbf{u}_1\mathbf{u}_1^* \end{bmatrix}.$$

The Householder vector $\mathbf{u}_1$ is determined such that

$$(I - 2\mathbf{u}_1\mathbf{u}_1^*)\begin{bmatrix} a_{21} \\ a_{31} \\ a_{41} \\ a_{51} \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad \text{with} \quad \mathbf{u}_1 = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix}.$$

The multiplication of P_1 from the left inserts the desired zeros in column 1 of A. The multiplication from the right is necessary in order to have similarity. Because of the nonzero structure of P_1 the first column of P_1A is not affected. Hence, the zeros stay there.

The reduction continues in a similar way:

$$P_1AP_1 = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \end{bmatrix} \xrightarrow{P_2 * / * P_2} \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & \times & \times & \times \end{bmatrix}$$

$$\xrightarrow{P_3 * / * P_3} \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \end{bmatrix} = P_3P_2P_1A\,\underbrace{P_1P_2P_3}_{U}.$$

```
1:  This algorithm reduces a matrix $A \in \mathbb{C}^{n \times n}$ to Hessenberg form $H$ by a sequence of Householder
    reflections. $H$ overwrites $A$.
2:  for $k = 1$ to $n-2$ do
3:    Generate the Householder reflector $P_k$;
4:    $\{$Apply $P_k = I_k \oplus (I_{n-k} - 2\mathbf{u_k}\mathbf{u_k}^*)$ from the left to $A\}$
5:    $A_{k+1:n,k:n} := A_{k+1:n,k:n} - 2\mathbf{u_k}(\mathbf{u_k}^* A_{k+1:n,k:n})$;
6:    $\{$Apply $P_k$ from the right, $A := AP_k\}$
7:    $A_{1:n,k+1:n} := A_{1:n,k+1:n} - 2(A_{1:n,k+1:n}\mathbf{u_k})\mathbf{u_k}^*$;
8:  end for
9:  if eigenvectors are desired form $U = P_1 \cdots P_{n-2}$ then
10:   $U := I_n$;
11:   for $k = n-2$ downto 1 do
12:     $\{$Update $U := P_kU\}$
13:     $U_{k+1:n,k+1:n} := U_{k+1:n,k+1:n} - 2\mathbf{u_k}(\mathbf{u_k}^* U_{k+1:n,k+1:n})$;
14:   end for
15: end if
```

ALGORITHM 3.3: **Reduction to Hessenberg form**

Algorithm 3.3 gives the details for the general $n \times n$ case. In step 4 of this algorithm, the Householder reflector is generated such that

$$(I - 2\mathbf{u}_k\mathbf{u}_k^*)\begin{bmatrix} a_{k+1,k} \\ a_{k+2,k} \\ \vdots \\ a_{n,k} \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad \text{with} \quad \mathbf{u}_k = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-k} \end{bmatrix} \qquad \text{and} \quad |\alpha| = \|\mathbf{x}\|$$

according to the considerations of the previous subsection. The Householder vectors are stored at the locations of the zeros. Therefore the matrix $U = P_1 \cdots P_{n-2}$ that effects the similarity transformation from the full A to the Hessenberg H is computed after all Householder vectors have been generated, thus saving $(2/3)n^3$ flops. The overall complexity of the reduction is

- Application of P_k from the left: $\sum\limits_{k=1}^{n-2} 4(n-k-1)(n-k) \approx \frac{4}{3}n^3$
- Application of P_k from the right: $\sum\limits_{k=1}^{n-2} 4(n)(n-k) \approx 2n^3$
- Form $U = P_1 \cdots P_{n-2}$: $\sum\limits_{k=1}^{n-2} 4(n-k)(n-k) \approx \frac{4}{3}n^3$

Thus, the reduction to Hessenberg form costs $\frac{10}{3}n^3$ flops without forming the transforming matrix and $\frac{14}{3}n^3$ including forming this matrix.

3.4 Improving the convergence of the QR algorithms

We have seen how the QR algorithm for computing the Schur form of a matrix A can be executed more economically if the matrix A is first transformed to Hessenberg form. Now we want to show how the convergence of the Hessenberg QR algorithm can be improved dramatically by introducing **(spectral) shifts** into the algorithm.

Lemma 3.4. *Let H be an **irreducible** Hessenberg matrix, i.e., $h_{i+1,i} \neq 0$ for all $i = 1, \ldots, n-1$. Let $H = QR$ be the QR factorization of H. Then for the diagonal elements of R we have*

$$|r_{kk}| > 0 \qquad \textit{for all } k < n.$$

Thus, if H is singular then $r_{nn} = 0$.

Proof. Let us look at the k-th step of the Hessenberg QR factorization. For illustration, let us consider the case $k = 3$ in a 5×5 example, where the matrix has the structure

$$\begin{bmatrix} + & + & + & + & + \\ 0 & + & + & + & + \\ 0 & 0 & + & + & + \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \end{bmatrix}.$$

The plus-signs indicate elements that have been modified. In step 3, the (nonzero) element h_{43} will be zeroed by a Givens rotation $G(3, 4, \varphi)$ that is determined such that

$$\begin{bmatrix} \cos(\varphi) & -\sin(\varphi) \\ \sin(\varphi) & \cos(\varphi) \end{bmatrix} \begin{bmatrix} \tilde{h}_{kk} \\ h_{k+1,k} \end{bmatrix} = \begin{bmatrix} r_{kk} \\ 0 \end{bmatrix}.$$

Because the Givens rotation preserves vector lengths, we have

$$|r_{kk}|^2 = |\tilde{h}_{kk}|^2 + |h_{k+1,k}|^2 \geq |h_{k+1,k}|^2 > 0,$$

which confirms the claim. ■

We apply this Lemma to motivate a further strategy to speed up the convergence of the QR algorithm.

Let λ be an eigenvalue of the irreducible Hessenberg matrix H. Let us check what happens it we perform

```
1: $H - \lambda I = QR$ {QR factorization}
2: $\overline{H} = RQ + \lambda I$
```

First we notice that $\overline{H} \sim H$. In fact,

$$\overline{H} = Q^*(H - \lambda I)Q + \lambda I = Q^*HQ.$$

Second, by Lemma 3.4 we have

$$H - \lambda I = QR, \qquad \text{with} \qquad R = \left[\begin{array}{cc} \diagdown & \rceil \\ & 0 \end{array}\right].$$

Thus,

$$RQ = \left[\begin{array}{cc} \diagdown\!\diagdown & \rceil \\ & 0\,0 \end{array}\right]$$

and

$$\overline{H} = RQ + \lambda I = \left[\begin{array}{cc} \diagdown\!\diagdown & \rceil \\ & 0\,\lambda \end{array}\right] = \begin{bmatrix} \overline{H}_1 & \mathbf{h}_1 \\ \mathbf{0}^T & \lambda \end{bmatrix}.$$

So, if we apply a QR step with a **perfect shift** to a Hessenberg matrix, the eigenvalue drops out. We then could **deflate**, i.e., proceed the algorithm with the smaller matrix $\overline{H}_1$.

Remark 3.1. *We could prove the existence of the Schur decomposition in the following way. (1) transform the arbitrary matrix to Hessenberg form. (2) Do the perfect shift Hessenberg QR with the eigenvalues which we known to exist one after the other.*

3.4.1 A numerical example

We use a matrix of a previous MATLAB experiments to show that perfect shifts actually work.

```
D = diag([4 3 2 1]); rand('seed',0);
S=rand(4); S = (S - .5)*2;
A = S*D/S;
format short e
H = hess(A)
[Q,R] = qr(H - 2*eye(4))
H1 = R*Q + 2*eye(4)
format long
lam eig(H1(1:3,1:3))
```

MATLAB produces the output

```
    H =  [ -4.4529e-01  -1.8641e+00  -2.8109e+00   7.2941e+00]
         [  8.0124e+00   6.2898e+00   1.2058e+01  -1.6088e+01]
         [               4.0087e-01   1.1545e+00  -3.3722e-01]
         [                           -1.5744e-01   3.0010e+00]

    Q =  [ -2.9190e-01  -7.6322e-01  -4.2726e-01  -3.8697e-01]
         [  9.5645e-01  -2.3292e-01  -1.3039e-01  -1.1810e-01]
         [               6.0270e-01  -5.9144e-01  -5.3568e-01]
         [                           -6.7130e-01   7.4119e-01]

    R =  [  8.3772e+00   4.6471e+00   1.2353e+01  -1.7517e+01]
         [               6.6513e-01  -1.1728e+00  -2.0228e+00]
         [                            2.3453e-01  -1.4912e+00]
         [                                        -2.4425e-14]

   H1 =  [  3.9994e+00  -3.0986e-02   2.6788e-01  -2.3391e+01]
         [  6.3616e-01   1.1382e+00   1.9648e+00  -9.4962e-01]
         [               1.4135e-01   2.8623e+00  -1.2309e+00]
         [                            1.6396e-14   2.0000e+00]

  lam =  [9.99999999999993e-01 4.00000000000003e+00 3.00000000000000e+00]
```

3.4.2 QR algorithm with shifts

This considerations indicate that it may be good to introduce shifts into the QR algorithm. However, we cannot choose perfect shifts because we do not know the eigenvalues of the matrix! We therefore need heuristics how to *estimate* eigenvalues. One such heuristic is the **Rayleigh quotient shift**: Set the shift σ_k in the k-th step of the QR algorithm equal to the last diagonal element:

$$\sigma_k := h_{n,n}^{(k-1)}. \tag{3.7}$$

```
Let $H_0 = H \in \mathbb{C}^{n\times n}$ be an upper Hessenberg matrix. This algorithm computes its Schur normal form $H = UTU$.
$k := 0$;
for m=n,n-1,...,2 do
    repeat
        $k := k + 1$;
        $\sigma_k := h_{m,m}^{(k-1)}$;
        $H_{k-1} - \sigma_k I =: Q_k R_k$;
        $H_k := R_k Q_k + \sigma_k I$;
        $U_k := U_{k-1} Q_k$;
    until $|h_{m,m-1}^{(k)}|$ is sufficiently small
end for
$T := H_k$;
```

ALGORITHM 3.4: **The Hessenberg QR algorithm with Rayleigh quotient shift**

Algorithm 3.4 implements this heuristic. Notice that the shift changes in each iteration step! Notice also that **deflation** is incorporated in Algorithm 3.4. As soon as the last lower off-diagonal element is sufficiently small, it is *declared* zero, and the algorithm proceeds with a smaller matrix. In Algorithm 3.4 the actual size of the matrix is $m \times m$.

Lemma 3.4 guarantees that a zero is produced at position $(n, n-1)$ in the Hessenberg matrix. What happens, if $h_{n,n}$ is a *good* approximation to an eigenvalue of H? Let us assume that we have an irreducible Hessenberg matrix

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \varepsilon & h_{n,n} \end{bmatrix},$$

where ε is a small quantity. If we perform a shifted Hessenberg QR step, we first have to factor $H - h_{n,n}I$, $QR = H - h_{n,n}I$. After $n-2$ steps of this factorization the R-factor is almost upper triangular,

$$\begin{bmatrix} + & + & + & + & + \\ 0 & + & + & + & + \\ 0 & 0 & + & + & + \\ 0 & 0 & 0 & \alpha & \beta \\ 0 & 0 & 0 & \varepsilon & 0 \end{bmatrix}.$$

From (3.5) we see that the last Givens rotation has the nontrivial elements

$$c_{n-1} = \frac{\alpha}{\sqrt{|\alpha|^2 + |\varepsilon|^2}}, \qquad s_{n-1} = \frac{-\varepsilon}{\sqrt{|\alpha|^2 + |\varepsilon|^2}}.$$

Applying the Givens rotations from the right one sees that the last lower off-diagonal element of $\overline{H} = RQ + h_{n,n}I$ becomes

$$\bar{h}_{n,n-1} = \frac{\varepsilon^2 \beta}{\alpha^2 + \varepsilon^2}. \tag{3.8}$$

So, we have *quadratic convergence* unless α is not tiny either.

A second even more often used shift strategy is the **Wilkinson shift**:

$$\sigma_k := \text{eigenvalue of } \begin{bmatrix} h_{n-1,n-1}^{(k-1)} & h_{n-1,n}^{(k-1)} \\ h_{n,n-1}^{(k-1)} & h_{n,n}^{(k-1)} \end{bmatrix} \text{ that is closer to } h_{n,n}^{(k-1)}. \tag{3.9}$$

3.4.3 A numerical example

We give an example for the Hessenberg QR algorithm with shift, but without deflation. The MATLAB code

```
D = diag([4 3 2 1]);
rand('seed',0);
S=rand(4); S = (S - .5)*2;
A = S*D/S;
H = hess(A)

for i=1:8,
   [Q,R] = qr(H-H(4,4)*eye(4)); H = R*Q+H(4,4)*eye(4);
end
```

produces the output

```
H( 0) =  [ -4.4529e-01  -1.8641e+00  -2.8109e+00   7.2941e+00]
         [  8.0124e+00   6.2898e+00   1.2058e+01  -1.6088e+01]
         [  0.0000e+00   4.0087e-01   1.1545e+00  -3.3722e-01]
         [  0.0000e+00   0.0000e+00  -1.5744e-01   3.0010e+00]

H( 1) =  [  3.0067e+00   1.6742e+00  -2.3047e+01  -4.0863e+00]
         [  5.2870e-01   8.5146e-01   1.1660e+00  -1.5609e+00]
         [              -1.7450e-01   3.1421e+00  -1.1140e-01]
         [                           -1.0210e-03   2.9998e+00]

H( 2) =  [  8.8060e-01  -4.6537e-01   9.1630e-01   1.6146e+00]
         [ -1.7108e+00   5.3186e+00   2.2839e+01  -4.0224e+00]
         [              -2.2542e-01   8.0079e-01   5.2445e-01]
         [                           -1.1213e-07   3.0000e+00]

H( 3) =  [  1.5679e+00   9.3774e-01   1.5246e+01   1.2703e+00]
         [  1.3244e+00   2.7783e+00   1.7408e+01   4.1764e+00]
         [               3.7230e-02   2.6538e+00  -7.8404e-02]
         [                            8.1284e-15   3.0000e+00]

H( 4) =  [  9.9829e-01  -7.5537e-01  -5.6915e-01   1.9031e+00]
         [ -3.2279e-01   5.1518e+00   2.2936e+01  -3.9104e+00]
         [              -1.6890e-01   8.4993e-01   3.8582e-01]
         [                           -5.4805e-30   3.0000e+00]

H( 5) =  [  9.3410e-01  -3.0684e-01   3.0751e+00  -1.2563e+00]
         [  3.5835e-01   3.5029e+00   2.2934e+01   4.1807e+00]
         [               3.2881e-02   2.5630e+00  -7.2332e-02]
         [                            1.1313e-59   3.0000e+00]

H( 6) =  [  1.0005e+00  -8.0472e-01  -8.3235e-01   1.9523e+00]
         [ -7.5927e-02   5.1407e+00   2.2930e+01  -3.8885e+00]
         [              -1.5891e-01   8.5880e-01   3.6112e-01]
         [                         -1.0026e-119   3.0000e+00]

H( 7) =  [  9.7303e-01  -6.4754e-01  -8.9829e-03  -1.8034e+00]
         [  8.2551e-02   3.4852e+00   2.3138e+01   3.9755e+00]
         [               3.3559e-02   2.5418e+00  -7.0915e-02]
         [                           3.3770e-239   3.0000e+00]

H( 8) =  [  1.0002e+00  -8.1614e-01  -8.9331e-01   1.9636e+00]
         [ -1.8704e-02   5.1390e+00   2.2928e+01  -3.8833e+00]
         [              -1.5660e-01   8.6086e-01   3.5539e-01]
         [                            0.0000e+00   3.0000e+00]
```

The numerical example shows that the shifted Hessenberg QR algorithm can work very nicely. In this example the (4,3) element is about 10^{-30} after 3 steps. (We could stop there.) The example also nicely shows a quadratic convergence rate.

3.5 The double shift QR algorithm

The shifted Hessenberg QR algorithm does not always work so nicely as in the previous example. If α in (3.8) is $\mathcal{O}(\varepsilon)$ then $h_{n,n-1}$ can be large. (A small α indicates a near singular $H_{1:n-1,1:n-1}$.)

Another problem occurs if real Hessenberg matrices have complex eigenvalues. We know that for reasonable convergence rates the shifts must be complex. If an eigenvalue λ has been found we can execute a single perfect shift with $\bar{\lambda}$. It is (for rounding errors) unprobable however that we will get back to a real matrix.

Since the eigenvalues come in complex conjugate pairs it is natural to search for a pair of eigenvalues right-away. This is done by collapsing two shifted QR steps in one **double step** with the two shifts being complex conjugates of each other.

Let σ_1 and σ_2 be two eigenvalues of the real matrix

$$G = \begin{bmatrix} h_{n-1,n-1}^{(k-1)} & h_{n-1,n}^{(k-1)} \\ h_{n,n-1}^{(k-1)} & h_{n,n}^{(k-1)} \end{bmatrix} \in \mathbb{R}^{2\times 2}.$$

If $\sigma_1 \in \mathbb{C}$ then $\sigma_2 = \bar{\sigma}_1$. Let us perform two QR steps using σ_1 and σ_2 as shifts:

$$\begin{aligned} H - \sigma_1 I &= Q_1 R_1, \\ H_1 &= R_1 Q_1 + \sigma_1 I, \\ H_1 - \sigma_2 I &= Q_2 R_2, \\ H_2 &= R_2 Q_2 + \sigma_2 I. \end{aligned} \tag{3.10}$$

Here, for convenience, we wrote H, H_1, and H_2 for H_{k-1}, H_k, and H_{k+1}, respectively. From the second and third equation in (3.10) we get

$$R_1 Q_1 + (\sigma_1 - \sigma_2) I = Q_2 R_2.$$

Multiplying this with Q_1 from the left and with R_1 from the right we get

$$\begin{aligned} Q_1 R_1 Q_1 R_1 + (\sigma_1 - \sigma_2) Q_1 R_1 &= Q_1 R_1 (Q_1 R_1 + (\sigma_1 - \sigma_2) I) \\ &= (H - \sigma_1 I)(H - \sigma_2 I) = Q_1 Q_2 R_2 R_1. \end{aligned}$$

Because $\sigma_2 = \bar{\sigma}_1$ we have

$$M := (H - \sigma_1 I)(H - \sigma_2 I) = H^2 - 2\text{Re}(\sigma) H + |\sigma|^2 I = Q_1 Q_2 R_2 R_1,$$

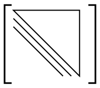
whence $(Q_1 Q_2)(R_2 R_1)$ is the QR factorization of a *real matrix*. Therefore, we can choose Q_1 and Q_2 such that $Z := Q_1 Q_2$ is real orthogonal. By consequence,

$$H_2 = (Q_1 Q_2)^* H (Q_1 Q_2) = Z^T H Z$$

is real.

A procedure to compute H_2 by avoiding complex arithmetic could consist of three steps:

1. Form the real matrix $M = H^2 - sH + tI$ with $s = 2\text{Re}(\sigma) = \text{trace}(G) = h_{n-1,n-1}^{(k-1)} + h_{n,n}^{(k-1)}$ and $t = |\sigma|^2 = \det(G) = h_{n-1,n-1}^{(k-1)} h_{n,n}^{(k-1)} - h_{n-1,n}^{(k-1)} h_{n,n-1}^{(k-1)}$. Notice that M has the form

$M =$  .

2. Compute the QR factorization $M = ZR$,
3. Set $H_2 = Z^T HZ$.

This procedure is too expensive since item 1, i.e., forming H^2 requires $\mathcal{O}(n^3)$ flops.

A remedy for the situation is provided by the Implicit Q Theorem.

Theorem 3.5. ***(The implicit Q Theorem)*** *Let $A \in \mathbb{R}^{n\times n}$. Let $Q = [\mathbf{q}_1, \ldots, \mathbf{q}_n]$ and $V = [\mathbf{v}_1, \ldots, \mathbf{v}_n]$ be matrices that both similarly transform A to Hessenberg form, $H = Q^T AQ$ and $G = V^T AV$. Let $k = n$ if H is irreducible; otherwise set k equal to the smallest positive integer with $h_{k+1,k} = 0$.*

If $\mathbf{q}_1 = \mathbf{v}_1$ then $\mathbf{q}_i = \pm\mathbf{v}_i$ and $|h_{i+1,i}| = |g_{i+1,i}|$ for $i = 2, \ldots, k$. If $k < n$ then $g_{k+1,k} = 0$.

Proof. Let $W = V^T Q$. Clearly, W is orthogonal, and $GW = WH$.

We first show that the first k columns of W form an upper triangular matrix, i.e.,

$$\mathbf{w}_i = W\mathbf{e}_i \in \text{span}\{\mathbf{e}_1, \ldots, \mathbf{e}_i\}, \qquad i \le k.$$

(Notice that orthogonal upper triangular matrices are diagonal.)

For $i = 1$ we have $\mathbf{w}_1 = \mathbf{e}_1$ by the assumption that $\mathbf{q}_1 = \mathbf{v}_1$. For $1 < i \le k$ we use the equality $GW = WH$. The $(i-1)$-th column of this equation reads

$$G\mathbf{w}_{i-1} = GW\mathbf{e}_{i-1} = WH\mathbf{e}_{i-1} = \sum_{j=1}^{i} \mathbf{w}_j h_{j,i-1}.$$

Since $h_{i,i-1} \neq 0$ we have

$$\mathbf{w}_i h_{i,i-1} = G\mathbf{w}_{i-1} - \sum_{j=1}^{i-1} \mathbf{w}_j h_{j,i-1} \in \text{span}\{\mathbf{e}_1, \ldots \mathbf{e}_i\},$$

as G is a Hessenberg matrix. Thus $\mathbf{w}_i = \pm\mathbf{e}_i$, $i \le k$.

Since $\mathbf{w}_i = \pm V^T Q\mathbf{e}_i = V^T\mathbf{q}_i = \pm\mathbf{e}_i$ we see that $\mathbf{q}_i$ is orthogonal to all columns of V except the i-th. Therefore, we must have $\mathbf{q}_i = \pm\mathbf{v}_i$. Further,

$$h_{i,i-1} = \mathbf{e}_i^T H\mathbf{e}_{i-1} = \mathbf{e}_i^T Q^T AQ\mathbf{e}_{i-1} = \mathbf{e}_i^T Q^T VGV^T Q\mathbf{e}_{i-1} = \mathbf{w}_i^T G\mathbf{w}_{i-1} = \pm g_{i,i-1},$$

thus, $|h_{i,i-1}| = |g_{i,i-1}|$. If $h_{k+1,k} = 0$ then

$$g_{k+1,k} = \mathbf{e}_{k+1}^T G\mathbf{e}_k = \pm\,\mathbf{e}_{k+1}^T GW\mathbf{e}_k = \pm\,\mathbf{e}_{k+1}^T WH\mathbf{e}_k = \pm\,\mathbf{e}_{k+1}^T \sum_{j=1}^{k} \mathbf{w}_j h_{j,k} = 0.$$

since $\mathbf{e}_{k+1}^T\mathbf{w}_j = \pm\mathbf{e}_{k+1}^T\mathbf{e}_j = 0$ for $j \le k$. ■

We apply the Implicit Q Theorem in the following way: We want to compute the Hessenberg matrix $H_2 = Z^T HZ$ where ZR is the QR factorization of $M = H^2 - sH + tI$. The Implicit Q Theorem now tells us that we essentially get H_2 by any unitary similarity transformation $H \to Z_1^* HZ_1$ provided that $Z_1^* HZ_1$ is Hessenberg and $Z_1\mathbf{e}_1 = Z\mathbf{e}_1$.

Let P_0 be the Householder reflector with

$$P_0^T M\mathbf{e}_1 = P_0^T(H^2 - 2\text{Re}(\sigma)H + |\sigma|^2 I)\mathbf{e}_1 = \alpha\mathbf{e}_1.$$

Since only the first three elements of the first column $M\mathbf{e}_1$ of M are nonzero, P_0 has the structure

$$P_0 = \begin{bmatrix} \times & \times & \times & & & \\ \times & \times & \times & & & \\ \times & \times & \times & & & \\ & & & 1 & & \\ & & & & \ddots & \\ & & & & & 1 \end{bmatrix}.$$

So,

$$H'_{k-1} := P_0^T H_{k-1} P_0 = \left[\begin{array}{ccc|cccc} \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times \\ + & \times & \times & \times & \times & \times & \times \\ \hline + & + & \times & \times & \times & \times & \times \\ & & & \times & \times & \times & \times \\ & & & & \times & \times & \times \\ & & & & & \times & \times \end{array}\right].$$

We now reduce $P_0^T H_{k-1} P_0$ similarly to Hessenberg form the same way as we did earlier, by a sequence of Householder reflectors $P_1, \ldots, P_{n-2}$. However, $P_0^T H_{k-1} P_0$ is a Hessenberg matrix up to the **bulge** at the top left. We take into account this structure when forming the $P_i = I - 2\mathbf{p}_i\mathbf{p}_i^T$. So, the structures of P_1 and of $P_1^T P_0^T H_{k-1} P_0 P_1$ are

$$P_1 = \left[\begin{array}{ccccccc} 1 & & & & & & \\ & \times & \times & \times & & & \\ & \times & \times & \times & & & \\ & \times & \times & \times & & & \\ & & & & 1 & & \\ & & & & & 1 & \\ & & & & & & 1 \end{array}\right], \quad H''_{k-1} = P_1^T H'_{k-1} P_1 = \left[\begin{array}{c|ccc|ccc} \times & \times & \times & \times & \times & \times & \times \\ \hline \times & \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times & \times \\ 0 & + & \times & \times & \times & \times & \times \\ \hline & + & + & \times & \times & \times & \times \\ & & & & \times & \times & \times \\ & & & & & \times & \times \end{array}\right].$$

The transformation with P_1 has chased the bulge one position down the diagonal. The consecutive reflectors push it further by one position each until it falls out of the matrix at the end of the diagonal. Pictorially, we have

$$H'''_{k-1} = P_2^T H''_{k-1} P_2 = \left[\begin{array}{cc|ccc|cc} \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times \\ \hline & \times & \times & \times & \times & \times & \times \\ & 0 & \times & \times & \times & \times & \times \\ & 0 & + & \times & \times & \times & \times \\ \hline & & + & + & \times & \times & \times \\ & & & & & \times & \times \end{array}\right]$$

$$H''''_{k-1} = P_3^T H'''_{k-1} P_3 = \left[\begin{array}{ccc|ccc|c} \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times & \times \\ \hline & & \times & \times & \times & \times & \times \\ & & 0 & \times & \times & \times & \times \\ & & 0 & + & \times & \times & \times \\ \hline & & & + & + & \times & \times \end{array}\right]$$

$$H'''''_{k-1} = P_4^T H''''_{k-1} P_4 = \left[\begin{array}{cccc|ccc} \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times & \times \\ & & \times & \times & \times & \times & \times \\ \hline & & & \times & \times & \times & \times \\ & & & 0 & \times & \times & \times \\ & & & 0 & + & \times & \times \end{array}\right]$$

$$H''''''_{k-1} = P_5^T H'''''_{k-1} P_5 = \left[\begin{array}{ccccc|cc} \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times & \times \\ & & \times & \times & \times & \times & \times \\ & & & \times & \times & \times & \times \\ \hline & & & & \times & \times & \times \\ & & & & 0 & \times & \times \end{array}\right]$$

It is easy to see that the Householder vector $\mathbf{p}_i$, $i < n-2$, has only three nonzero elements at position $i+1, i+2, i+3$. Of $\mathbf{p}_{n-2}$ only the last two elements are nonzero. Clearly, $P_0 P_1 \cdots P_{n-2}\mathbf{e}_1 = P_0\mathbf{e}_1 = M\mathbf{e}_1/\alpha$.

```
Let $H_0 = H \in \mathbb{R}^{n\times n}$ be an upper Hessenberg matrix. This algorithm computes its real Schur form
$H = UTU$ using the Francis double step QR algorithm. $T$ is a quasi upper triangular matrix.
$p := n$; $\{p$ indicates the ‘actual’ matrix size.$\}$
**while** $p > 2$ **do**
    $q := p - 1$;
    $s := H_{q,q} + H_{p,p}$;   $t := H_{q,q}H_{p,p} - H_{q,p}H_{p,q}$;
    $\{$compute first 3 elements of first column of $M\}$
    $x := H_{1,1}^2 + H_{1,2}H_{2,1} - sH_{1,1} + t$;
    $y := H_{2,1}(H_{1,1} + H_{2,2} - s)$;
    $z := H_{2,1}H_{3,2}$;
    **for** $k = 0$ to $p - 3$ **do**
        Determine the Householder reflector $P$ with $P^T\ [x;\ y;\ z]^T = \alpha\mathbf{e}_1$;
        $r := \max\{1, k\}$;
        $H_{k+1:k+3,r:n} := P^T H_{k+1:k+3,r:n}$;
        $r := \min\{k + 4, p\}$;
        $H_{1:r,k+1:k+3} := H_{1:r,k+1:k+3}P$;
        $x := H_{k+2,k+1}$;   $y := H_{k+3,k+1}$;
        **if** $k < p - 3$ **then**
            $z := H_{k+4,k+1}$;
        **end if**
    **end for**
    Determine the Givens rotation $P$ with $P^T\ [x;\ y]^T = \alpha\mathbf{e}_1$;
    $H_{q:p,p-2:n} := P^T H_{q:p,p-2:n}$;
    $H_{1:p,p-1:p} := H_{1:p,p-1:p}P$;
    $\{$check for convergence$\}$
    **if** $|H_{p,q}| < \varepsilon\,(|H_{q,q}| + |H_{p,p}|)$ **then**
        $H_{p,q} := 0$;   $p := p - 1$;   $q := p - 1$;
    **else if** $|H_{p-1,q-1}| < \varepsilon\,(|H_{q-1,q-1}| + |H_{q,q}|)$ **then**
        $H_{p-1,q-1} := 0$;   $p := p - 2$;   $q := p - 1$;
    **end if**
**end while**
```

ALGORITHM 3.5: **The Francis’ Double step QR algorithm**

3.5.1 A numerical example

We consider a simple MATLAB implementation of the Algorithm 3.5 to compute the eigenvalues of the real matrix

$$A = \begin{bmatrix} 7 & 3 & 4 & -11 & -9 & -2 \\ -6 & 4 & -5 & 7 & 1 & 12 \\ -1 & -9 & 2 & 2 & 9 & 1 \\ -8 & 0 & -1 & 5 & 0 & 8 \\ -4 & 3 & -5 & 7 & 2 & 10 \\ 6 & 1 & 4 & -11 & -7 & -1 \end{bmatrix}$$

that has the spectrum

$$\sigma(A) = \{1 \pm 2i, 3, 4, 5 \pm 6i\}.$$

The intermediate output of the code was (after some editing) the following:

```
>> H=hess(A)

H(0) =

    7.0000    7.2761    5.8120   -0.1397    9.0152    7.9363
   12.3693    4.1307   18.9685   -1.2071   10.6833    2.4160
         0   -7.1603    2.4478   -0.5656   -4.1814   -3.2510
         0         0   -8.5988    2.9151   -3.4169    5.7230
         0         0         0    1.0464   -2.8351  -10.9792
         0         0         0         0    1.4143    5.3415

>> PR=qr2st(H)

[it_step, p = n_true, H(p,p-1), H(p-1,p-2)]

     1     6  -1.7735e-01  -1.2807e+00
     2     6  -5.9078e-02  -1.7881e+00
     3     6  -1.6115e-04  -5.2705e+00
     4     6  -1.1358e-07  -2.5814e+00
     5     6   1.8696e-14   1.0336e+01
     6     6  -7.1182e-23  -1.6322e-01


H(6) =

    5.0000    6.0000    2.3618    5.1837  -13.4434   -2.1391
   -6.0000    5.0000    2.9918   10.0456   -8.7743  -21.0094
    0.0000   -0.0001   -0.9393    3.6939   11.7357    3.8970
    0.0000   -0.0000   -1.9412    3.0516    2.9596  -10.2714
         0    0.0000    0.0000   -0.1632    3.8876    4.1329
         0         0         0    0.0000   -0.0000    3.0000


     7     5   1.7264e-02  -7.5016e-01
     8     5   2.9578e-05  -8.0144e-01
     9     5   5.0602e-11  -4.6559e+00
    10     5  -1.3924e-20  -3.1230e+00


H(10) =

    5.0000    6.0000   -2.7603    1.3247   11.5569   -2.0920
   -6.0000    5.0000  -10.7194    0.8314   11.8952   21.0142
   -0.0000   -0.0000    3.5582    3.3765    5.9254   -8.5636
   -0.0000   -0.0000   -3.1230   -1.5582  -10.0935   -6.3406
         0         0         0    0.0000    4.0000    4.9224
         0         0         0    0.0000         0    3.0000


    11     4   1.0188e+00  -9.1705e-16


H(11) =

    5.0000    6.0000  -10.2530    4.2738  -14.9394  -19.2742
   -6.0000    5.0000   -0.1954    1.2426    7.2023   -8.6299
   -0.0000   -0.0000    2.2584   -5.4807  -10.0623    4.4380
    0.0000   -0.0000    1.0188   -0.2584   -5.9782   -9.6872
         0         0         0         0    4.0000    4.9224
         0         0         0    0.0000         0    3.0000
```

3.5.2 The complexity

We first estimate the complexity of a single step of the double step Hessenberg QR algorithm. The most expensive operations are the applications of the 3×3 Householder reflectors in steps 13 and 15 of Algorithm 3.5. Let us first count the flops for applying the Householder reflector to a 3-vector,

$$\mathbf{x} := (I - 2\mathbf{u}\mathbf{u}^T)\mathbf{x} = \mathbf{x} - \mathbf{u}(2\mathbf{u}^T\mathbf{x}).$$

The inner product $\mathbf{u}^T\mathbf{x}$ costs 5 flops, multiplying with 2 another one. The operation $\mathbf{x} := \mathbf{x} - \mathbf{u}\gamma$, $\gamma = 2\mathbf{u}^T\mathbf{x}$, cost 6 flops, altogether 12 flops.

In the k-th step of the loop there are $n - k$ of these application from the left in step 13 and $k + 4$ from the right in step 15. In this step there are thus about $12n + \mathcal{O}(1)$ flops to be executed. As k is running from 1 to $p-3$. We have about $12pn$ flops for this step. Since p runs from n down to about 2 we have $6n^3$ flops. If we assume that two steps are required per eigenvalue the flop count for Francis' double step QR algorithm to compute *all* eigenvalues of a *real* Hessenberg matrix is $12n^3$. If also the eigenvector matrix is accumulated the two additional statements have to be inserted into Algorithm 3.5. After step 15 we have

```
$Q_{1:n,k+1:k+3} := Q_{1:n,k+1:k+3}P$;
```

and after step 23 we introduce

```
$Q_{1:n,p-1:p} := Q_{1:n,p-1:p}P$;
```

which costs another $12n^3$ flops.

We earlier gave the estimate of $6n^3$ flops for a Hessenberg QR step, see Algorithm 3.2. If the latter has to be spent in *complex* then the single shift Hessenberg QR algorithm is more expensive that a the double shift Hessenberg QR algorithm that is executed in real arithmetic.

Remember that the reduction to Hessenberg form costs $\frac{10}{3}n^3$ flops without forming the transformation matrix and $\frac{14}{3}n^3$ it this matrix is formed.

3.6 The symmetric tridiagonal QR algorithm

The QR algorithm can be applied rightway to Hermitian or symmetric matrices. By (3.1) we see that the QR algorithm generates a sequence $\{A_k\}$ of symmetric matrices. By taking into account the symmetry, the performance of the algorithm can be improved considerably. Furthermore, from Theorem 2.12 we know that Hermitian matrices have a real spectrum. Therefore, we can restrict ourselves to single shifts.

3.6.1 Reduction to tridiagonal form

The reduction of a full Hermitian matrix to Hessenberg form produces a Hermitian Hessenberg matrix, which (up to rounding errors) is a tridiagonal matrix. Let us consider how to take into account symmetry. To that end let us consider the first reduction step that introduces $n-2$ zeros into the first column (and the first row) of $A = A^* \in \mathbb{C}^{n\times n}$. Let

$$P_1 = \begin{bmatrix} 1 & \mathbf{0}^T \\ \mathbf{0} & I_{n-1} - 2\mathbf{u}_1\mathbf{u}_1^* \end{bmatrix}, \qquad \mathbf{u}_1 \in \mathbb{C}^n, \quad \|\mathbf{u}_1\| = 1.$$

Then,

$$\begin{aligned} A_1 := P_1^* A P_1 &= (I - 2\mathbf{u}_1\mathbf{u}_1^*)A(I - 2\mathbf{u}_1\mathbf{u}_1^*) \\ &= A - \mathbf{u}_1(2\mathbf{u}_1^*A - 2(\mathbf{u}_1^*A\mathbf{u}_1)\mathbf{u}_1^*) - \underbrace{(2A\mathbf{u}_1 - 2\mathbf{u}_1(\mathbf{u}_1^*A\mathbf{u}_1))}_{\mathbf{v}_1}\mathbf{u}_1^* \\ &= A - \mathbf{v}_1\mathbf{u}_1^* - \mathbf{v}_1\mathbf{u}_1^*. \end{aligned}$$

In the k-th step of the reduction we similarly have

$$A_k = P_k^* A_{k-1} P_k = A_{k-1} - \mathbf{v}_{k-1}\mathbf{u}_{k-1}^* - \mathbf{v}_{k-1}\mathbf{u}_{k-1}^*,$$

where the last $n-k$ elements of $\mathbf{u}_{k-1}$ and $\mathbf{v}_{k-1}$ are nonzero. Forming

$$\mathbf{v}_{k-1} = 2A_{k-1}\mathbf{u}_{k-1} - 2\mathbf{u}_{k-1}(\mathbf{u}_{k-1}^*A_{k-1}\mathbf{u}_{k-1})$$

costs $2(n-k)^2 + \mathcal{O}(n-k)$ flops. This complexity results from $A_{k-1}\mathbf{u}_{k-1}$. The rank-2 update of A_{k-1},

$$A_k = A_{k-1} - \mathbf{v}_{k-1}\mathbf{u}_{k-1}^* - \mathbf{v}_{k-1}\mathbf{u}_{k-1}^*,$$

requires another $2(n-k)^2 + \mathcal{O}(n-k)$ flops, taking into account symmetry. By consequence, the transformation to tridiagonal form can be accomplished in

$$\sum_{k=1}^{n-1}\left(4(n-k)^2 + \mathcal{O}(n-k)\right) = \frac{4}{3}n^3 + \mathcal{O}(n^2)$$

floating point operations.

3.6.2 The tridiagonal QR algorithm

In the symmetric case the Hessenberg QR algorithm becomes a tridiagonal QR algorithm. This can be executed in an **explicit** or an **implicit** way. In the explicit form, a QR step is essentially

```
Choose a shift $\mu$
Compute the QR factorization $A - \mu I = QR$
Update $A$ by $A = RQ + \mu I$.
```

Of course, this is done by means of plane rotations and by respecting the symmetric tridiagonal structure of A.

In the more elegant implicit form of the algorithm we first compute the first Givens rotation $G_0 = G(1, 2, \vartheta)$ of the QR factorization that zeros the $(2, 1)$ element of $A - \mu I$,

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a_{11} - \mu \\ a_{21} \end{bmatrix} = \begin{bmatrix} * \\ 0 \end{bmatrix}, \qquad c = \cos(\vartheta_0), \quad s = \sin(\vartheta_0).$$

Performing a similary transformation with G_0 we have ($n = 5$)

$$G_0^* A G_0 = A' = \begin{bmatrix} \times & \times & + & & \\ \times & \times & \times & & \\ + & \times & \times & \times & \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix}$$

Similar as with the double step Hessenberg QR algorithm we chase the bulge down the diagonal. In the 5×5 example this becomes

$$A \xrightarrow[= G(1,2,\vartheta_0)]{G_0} \begin{bmatrix} \times & \times & + & & \\ \times & \times & \times & & \\ + & \times & \times & \times & \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix} \xrightarrow[= G(2,3,\vartheta_1)]{G_1} \begin{bmatrix} \times & \times & 0 & & \\ \times & \times & \times & + & \\ 0 & \times & \times & \times & \\ & + & \times & \times & \times \\ & & & \times & \times \end{bmatrix}$$

$$\xrightarrow[= G(3,4,\vartheta_2)]{G_2} \begin{bmatrix} \times & \times & 0 & & \\ \times & \times & \times & & \\ & \times & \times & \times & + \\ & 0 & \times & \times & \times \\ & & + & \times & \times \end{bmatrix} \xrightarrow[= G(4,5,\vartheta_3)]{G_3} \begin{bmatrix} \times & \times & & & \\ \times & \times & \times & & \\ & \times & \times & \times & 0 \\ & & \times & \times & \times \\ & & 0 & \times & \times \end{bmatrix} = \overline{A}.$$

The full step is given by

$$\overline{A} = Q^* A Q, \qquad Q = G_0 \, G_1 \, \cdots \, G_{n-2}.$$

Because $G_k \mathbf{e}_1 = \mathbf{e}_1$ for $k > 0$ we have

$$Q \, \mathbf{e}_1 = G_0 \, G_1 \cdots G_{n-2} \, \mathbf{e}_1 = G_0 \, \mathbf{e}_1.$$

Both explicit and implicit QR step form the same first plane rotation G_0. By referring to the Implicit Q Theorem 3.5 we see that explicit and implicit QR step compute *essentially* the same $\overline{A}$.

3.7 Summary

The QR algorithm is a very powerful algorithm to stably compute the eigenvalues and (if needed) the corresponding eigenvectors or Schur vectors. All steps of the algorithm cost $\mathcal{O}(n^3)$ floating point operations,

```
Let $T \in \mathbb{R}^{n\times n}$ be a symmetric tridiagonal matrix with diagonal entries $a_1, \ldots, a_n$ and off-diagonal entries $b_2, \ldots, b_n$.
This algorithm computes the eigenvalues $\lambda_1, \ldots, \lambda_n$ of $T$ and corresponding eigenvectors $\mathbf{q}_1, \ldots, \mathbf{q}_n$. The eigenvalues are stored in $a_1, \ldots, a_n$. The eigenvectors are stored in the matrix $Q$, such that $TQ = Q$ diag$(a_1, \ldots, a_n)$.
$m = n$ {Actual problem dimension. $m$ is reduced in the convergence check.}
while $m > 1$ do
    $d := (a_{m-1} - a_m)/2$; {Compute Wilkinson's shift}
    if $d = 0$ then
        $s := a_m - |b_m|$;
    else
        $s := a_m - b_m^2/(d + \text{sign}(d)\sqrt{d^2 + b_m^2})$;
    end if
    $x := a(1) - s$; {Implicit QR step begins here}
    $y := b(2)$;
    for $k = 1$ to $m - 1$ do
        if $m > 2$ then
            $[c, s] :=$ givens$(x, y)$;
        else
            Determine $[c, s]$ such that $\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} a_1 & b_2 \\ b_2 & a_2 \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$ is diagonal
        end if
        $w := cx - sy$;
        $d := a_k - a_{k+1}$;    $z := (2cb_{k+1} + ds)s$;
        $a_k := a_k - z$;    $a_{k+1} := a_{k+1} + z$;
        $b_{k+1} := dcs + (c^2 - s^2)b_{k+1}$;
        $x := b_{k+1}$;
        if $k > 1$ then
            $b_k := w$;
        end if
        if $k < m - 1$ then
            $y := -sb_{k+2}$;    $b_{k+2} := cb_{k+2}$;
        end if
        $Q_{1:n;\, k:k+1} := Q_{1:n;\, k:k+1} \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$;
    end for{Implicit QR step ends here}
    if $|b_m| < \varepsilon(|a_{m-1}| + |a_m|)$ then {Check for convergence}
        $m := m - 1$;
    end if
end while
```

ALGORITHM 3.6: **Symmetric tridiagonal QR algorithm with implicit Wilkinson shift**

Table 3.1 Complexity in flops to compute eigenvalues and eigen-/Schur-vectors of a real matrix

	nonsymmetric case		symmetric case	
	without	with	without	with
	Schurvectors		eigenvectors	
transformation to Hessenberg/tridiagonal form	$\frac{10}{3}n^3$	$\frac{14}{3}n^3$	$\frac{4}{3}n^3$	$\frac{8}{3}n^3$
real double step Hessenberg/tridiagonal QR algorithm (2 steps per eigenvalues assumed)	$\frac{20}{3}n^3$	$\frac{50}{3}n^3$	$24n^2$	$6n^3$
total	$10n^3$	$25n^3$	$\frac{4}{3}n^3$	$9n^3$

see Table 3.1. The one exception is the case where only eigenvalues are desired of a symmetric tridiagonal matrix. The linear algebra software package LAPACK [ABB$^+$94] contains subroutines for all possible ways the QR algorithm may be employed.

We finish by noting again, that the QR algorithm is a method for *dense* matrix problems. The reduction of a sparse matrix to tridiagonal or Hessenberg form produces **fill in**, thus destroying the sparsity structure which one almost allways tries to preserve.

Bibliography

[ABB$^+$94] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide - Release 2.0*. SIAM, Philadelphia, PA, 1994. (Software and guide are available from Netlib at URL `http://www.netlib.org/lapack/`).

[Dem97] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, PA, 1997.

[Fra61] J. G. F. Francis. The QR transformation: A unitary analogue to the LR transformation – part 1. *Computer Journal*, 4(3):265–271, 1961.

[Fra62] J. G. F. Francis. The QR transformation – part 2. *Computer Journal*, 4(4):332–345, 1962.

[GvL89] G. H. Golub and C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 2nd edition, 1989.

[Rut58] H. Rutishauser. Solution of eigenvalue problems with the LR-transformation. *NBS Appl. Math. Series*, 49:47–81, 1958.

[Wil65] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Clarendon Press, Oxford, 1965.

Chapter 4

Cuppen's Divide and Conquer Algorithm

In this chapter we deal with an algorithm that is designed for the efficient solution of the symmetric tridiagonal eigenvalue problem

$$T\mathbf{x} = \lambda\mathbf{x}, \qquad T = \begin{bmatrix} a_1 & b_1 & & \\ b_1 & a_2 & \ddots & \\ & \ddots & \ddots & b_{n-1} \\ & & b_{n-1} & a_n \end{bmatrix}. \tag{4.1}$$

We noticed from Table 3.1 that the reduction of a full symmetric matrix to a similar tridiagonal matrix requires about $\frac{8}{3}n^3$ while the tridiagonal QR algorithm needs an estimated $6n^3$ floating operations (flops) to converge. Because of the importance of this subproblem a considerable effort has been put into finding faster algorithms than the QR algorithms to solve the tridiagonal eigenvalue problem. In the mid-1980's Dongarra and Sorensen [DS87] promoted an algorithm originally proposed by Cuppen [Cup81]. This algorithm was based on a divide and conquer strategy. However, it took ten more years until a stable variant was found by Gu and Eisenstat [GE94, GE95]. Today, a stable implementation of this latter algorithm is available in LAPACK [ABB$^+$94].

4.1 The divide and conquer idea

Divide and conquer is an old strategy in military to defeat an enemy going back at least to Caesar. In computer science, divide and conquer (D&C) is an important algorithm design paradigm. It works by recursively breaking down a problem into two or more subproblems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem. Translated to our problem the strategy becomes

1. Partition the tridiagonal eigenvalue problem into two (or more) smaller eigenvalue problems.

2. Solve the two smaller problems.

3. Combine the solutions of the smaller problems to get the desired solution of the overall problem.

Evidently, this strategy can be applied recursively.

4.2 Partitioning the tridiagonal matrix

Partitioning the *irreducible* tridiagonal matrix is done in the following way. We write

$$
\begin{aligned}
T &= \left[\begin{array}{cccc|cccc}
a_1 & b_1 & & & & & & \\
b_1 & a_2 & \ddots & & & & & \\
 & \ddots & \ddots & b_{m-1} & & & & \\
 & & b_{m-1} & a_m & b_m & & & \\
\hline
 & & & b_m & a_{m+1} & b_{m+1} & & \\
 & & & & b_{m+1} & a_{m+2} & \ddots & \\
 & & & & & \ddots & \ddots & b_{n-1} \\
 & & & & & & b_{n-1} & a_n
\end{array}\right] \\
&= \left[\begin{array}{cccc|cccc}
a_1 & b_1 & & & & & & \\
b_1 & a_2 & \ddots & & & & & \\
 & \ddots & \ddots & b_{m-1} & & & & \\
 & & b_{m-1} & a_m - \mp b_m & & & & \\
\hline
 & & & & a_{m+1} - \mp b_m & b_{m+1} & & \\
 & & & & b_{m+1} & a_{m+2} & \ddots & \\
 & & & & & \ddots & \ddots & b_{n-1} \\
 & & & & & & b_{n-1} & a_n
\end{array}\right]
+ \left[\begin{array}{c|c}
 & \\
\pm b_m & b_m \\
\hline
b_m & \pm b_m \\
 &
\end{array}\right] \\
&= \left[\begin{array}{c|c}
T_1 & \\
\hline
 & T_2
\end{array}\right] + \rho \mathbf{u}\mathbf{u}^T \qquad \text{with } \mathbf{u} = \left[\begin{array}{c}
\pm \mathbf{e}_m \\
\hline
\mathbf{e}_1
\end{array}\right] \text{ and } \rho = \pm b_m,
\end{aligned}
\tag{4.2}
$$

where $\mathbf{e}_m$ is a vector of length $m \approx \frac{n}{2}$ and $\mathbf{e}_1$ is a vector of length $n - m$. Notice that the most straightforward way to partition the problem without modifying the diagonal elements leads to a rank-two modification. With the approach of (4.2) we have the original T as a sum of two smaller tridiagonal systems plus a *rank-one modification*.

4.3 Solving the small systems

We solve the half-sized eigenvalue problems,

$$
T_i = Q_i \Lambda_i Q_i^T, \quad Q_i^T Q_i = I, \qquad i = 1, 2. \tag{4.3}
$$

These two spectral decompositions can be computed by any algorithm, in particular also by this divide and conquer algorithm by which the T_i would be further split. It is clear that by this partitioning an large number of small problems can be generated that can be potentially solved in parallel. For a parallel algorithm, however, the further phases of the algorithm must be parallelizable as well.

Plugging (4.3) into (4.2) gives

$$
\left[\begin{array}{c|c}
Q_1^T & \\
\hline
 & Q_2^T
\end{array}\right]
\left(\left[\begin{array}{c|c}
T_1 & \\
\hline
 & T_2
\end{array}\right] + \rho \mathbf{u}\mathbf{u}^T \right)
\left[\begin{array}{c|c}
Q_1 & \\
\hline
 & Q_2
\end{array}\right]
= \left[\begin{array}{c|c}
\Lambda_1 & \\
\hline
 & \Lambda_2
\end{array}\right] + \rho \mathbf{v}\mathbf{v}^T \tag{4.4}
$$

with

$$
\mathbf{v} = \left[\begin{array}{c|c}
Q_1^T & \\
 & Q_2^T
\end{array}\right] \mathbf{u} = \left[\begin{array}{c}
\pm Q_1^T \mathbf{e}_m \\
Q_2^T \mathbf{e}_1
\end{array}\right] = \left[\begin{array}{c}
\pm \text{last row of } Q_1 \\
\text{first row of } Q_2
\end{array}\right]. \tag{4.5}
$$

Now we have arrived at the eigenvalue problem

$$
(D + \rho \mathbf{v}\mathbf{v}^T)\mathbf{x} = \lambda \mathbf{x}, \qquad D = \Lambda_1 \oplus \Lambda_2 = \text{diag}(\lambda_1, \ldots, \lambda_n). \tag{4.6}
$$

That is, we have to compute the spectral decomposition of a matrix that is a **diagonal plus a rank-one update**. Let

$$D + \rho \mathbf{v}\mathbf{v}^T = Q\Lambda Q^T \tag{4.7}$$

be this spectral decomposition. Then, the spectral decomposition of the tridiagonal T is

$$T = \left[\begin{array}{c|c} Q_1 & \\ & Q_2 \end{array}\right] Q\Lambda Q^T \left[\begin{array}{c|c} Q_1^T & \\ & Q_2^T \end{array}\right]. \tag{4.8}$$

Forming the product $(Q_1 \oplus Q_2)Q$ will turn out to be *the most expensive step of the algorithm*. It costs $n^3 + \mathcal{O}(n^2)$ floating point operations

4.4 Deflation

There are certain solutions of (4.7) that can be given immediately, by just looking carefully at the equation.

If there are zero entries in $\mathbf{v}$ then we have

$$\big(v_i = 0 \Leftrightarrow \mathbf{v}^T\mathbf{e}_i = 0\big) \quad \Longrightarrow \quad (D + \rho\mathbf{v}\mathbf{v}^T)\mathbf{e}_i = d_i\mathbf{e}_i. \tag{4.9}$$

Thus, if an entry of $\mathbf{v}$ vanishes we can read the eigenvalue from the diagonal of D at once and the corresponding eigenvector is a coordinate vector.

If identical entries occur in the diagonal of D, say $d_i = d_j$, with $i < j$, then we can find a plane rotation $G(i, j, \phi)$ (see (3.4)) such that it introduces a zero into the j-th position of $\mathbf{v}$,

$$G^T\mathbf{v} = G(i, j, \varphi)^T\mathbf{v} = \begin{bmatrix} \times \\ \vdots \\ \sqrt{v_i{}^2 + v_j{}^2} \\ \vdots \\ 0 \\ \vdots \\ \times \end{bmatrix} \begin{matrix} \\ \\ \leftarrow i \\ \\ \leftarrow j \\ \\ \\ \end{matrix}$$

Notice, that (for *any* φ),

$$G(i, j, \varphi)^T DG(i, j, \varphi) = D, \qquad d_i = d_j.$$

So, if there are multiple eigenvalues in D we can reduce all but one of them by introducing zeros in $\mathbf{v}$ and then proceed as previously in (4.9).

When working with floating point numbers we deflate if

$$|v_i| < C\varepsilon\|T\| \qquad \text{or} \qquad |d_i - d_j| < C\varepsilon\|T\|, \qquad\quad (\|T\| = \|D + \rho\mathbf{v}\mathbf{v}^T\|) \tag{4.10}$$

where C is a small constant. Deflation changes the eigenvalue problem for $D + \rho\mathbf{v}\mathbf{v}^T$ into the eigenvalue problem for

$$\begin{bmatrix} D_1 + \rho\mathbf{v}_1\mathbf{v}_1^T & O \\ O & D_2 \end{bmatrix} = G^T(D + \rho\mathbf{v}\mathbf{v}^T)G + E, \qquad \|E\| < C\varepsilon\sqrt{\|D\|^2 + |\rho|^2\|\mathbf{v}\|^4}, \tag{4.11}$$

where D_1 has no multiple diagonal entries and $\mathbf{v}_1$ has no zero entries. So, we have to compute the spectral decomposition of the matrix in (4.11) which is similar to a slight perturbation of the original matrix. G is the product of Givens rotations.

4.4.1 Numerical examples

Let us first consider

$$T = \left[\begin{array}{ccc|ccc} 1 & 1 & & & & \\ 1 & 2 & 1 & & & \\ & 1 & 3 & 1 & & \\ \hline & & 1 & 4 & 1 & \\ & & & 1 & 5 & 1 \\ & & & & 1 & 6 \end{array}\right] = \left[\begin{array}{ccc|ccc} 1 & 1 & & & & \\ 1 & 2 & 1 & & & \\ & 1 & 2 & 0 & & \\ \hline & & 0 & 3 & 1 & \\ & & & 1 & 5 & 1 \\ & & & & 1 & 6 \end{array}\right] + \left[\begin{array}{ccc|ccc} 0 & & & & & \\ & 0 & & & & \\ & & 1 & 1 & & \\ \hline & & 1 & 1 & & \\ & & & & 0 & \\ & & & & & 0 \end{array}\right]$$

$$= \left[\begin{array}{ccc|ccc} 1 & 1 & & & & \\ 1 & 2 & 1 & & & \\ & 1 & 2 & 0 & & \\ \hline & & 0 & 3 & 1 & \\ & & & 1 & 5 & 1 \\ & & & & 1 & 6 \end{array}\right] + \left[\begin{array}{c} 0 \\ 0 \\ 1 \\ \hline 1 \\ 0 \\ 0 \end{array}\right] \left[\begin{array}{c} 0 \\ 0 \\ 1 \\ \hline 1 \\ 0 \\ 0 \end{array}\right]^T = T_0 + \mathbf{u}\mathbf{u}^T.$$

Then a little MATLAB experiment shows that

$$Q_0^T T Q_0 = \begin{bmatrix} 0.1981 & & & & & \\ & 1.5550 & & & & \\ & & 3.2470 & & & \\ & & & 2.5395 & & \\ & & & & 4.7609 & \\ & & & & & 6.6996 \end{bmatrix} + \begin{bmatrix} 0.3280 \\ 0.7370 \\ 0.5910 \\ 0.9018 \\ -0.4042 \\ 0.1531 \end{bmatrix} \begin{bmatrix} 0.3280 \\ 0.7370 \\ 0.5910 \\ 0.9018 \\ -0.4042 \\ 0.1531 \end{bmatrix}^T$$

with

$$Q_0 = \begin{bmatrix} 0.7370 & -0.5910 & 0.3280 & & & \\ -0.5910 & -0.3280 & 0.7370 & & & \\ 0.3280 & 0.7370 & 0.5910 & & & \\ & & & 0.9018 & -0.4153 & 0.1200 \\ & & & -0.4042 & -0.7118 & 0.5744 \\ & & & 0.1531 & 0.5665 & 0.8097 \end{bmatrix}$$

Here it is not possible to deflate.

Let us now look at an example with more symmetry,

$$T = \left[\begin{array}{ccc|ccc} 2 & 1 & & & & \\ 1 & 2 & 1 & & & \\ & 1 & 2 & 1 & & \\ \hline & & 1 & 2 & 1 & \\ & & & 1 & 2 & 1 \\ & & & & 1 & 2 \end{array}\right] = \left[\begin{array}{ccc|ccc} 2 & 1 & & & & \\ 1 & 2 & 1 & & & \\ & 1 & 1 & 0 & & \\ \hline & & 0 & 1 & 1 & \\ & & & 1 & 2 & 1 \\ & & & & 1 & 2 \end{array}\right] + \left[\begin{array}{ccc|ccc} 0 & & & & & \\ & 0 & & & & \\ & & 1 & 1 & & \\ \hline & & 1 & 1 & & \\ & & & & 0 & \\ & & & & & 0 \end{array}\right]$$

$$= \left[\begin{array}{ccc|ccc} 2 & 1 & & & & \\ 1 & 2 & 1 & & & \\ & 1 & 1 & 0 & & \\ \hline & & 0 & 1 & 1 & \\ & & & 1 & 2 & 1 \\ & & & & 1 & 2 \end{array}\right] + \left[\begin{array}{c} 0 \\ 0 \\ 1 \\ \hline 1 \\ 0 \\ 0 \end{array}\right] \left[\begin{array}{c} 0 \\ 0 \\ 1 \\ \hline 1 \\ 0 \\ 0 \end{array}\right]^T = T_0 + \mathbf{u}\mathbf{u}^T.$$

Now, MATLAB gives

$$Q_0^T T Q_0 = \begin{bmatrix} 0.1981 & & & & & \\ & 1.5550 & & & & \\ & & 3.2470 & & & \\ & & & 0.1981 & & \\ & & & & 1.5550 & \\ & & & & & 3.2470 \end{bmatrix} + \begin{bmatrix} 0.7370 \\ -0.5910 \\ 0.3280 \\ 0.7370 \\ -0.5910 \\ 0.3280 \end{bmatrix} \begin{bmatrix} 0.7370 \\ -0.5910 \\ 0.3280 \\ 0.7370 \\ -0.5910 \\ 0.3280 \end{bmatrix}^T$$

with

$$Q_0 = \begin{bmatrix} 0.3280 & 0.7370 & 0.5910 & & & \\ -0.5910 & -0.3280 & 0.7370 & & & \\ 0.7370 & -0.5910 & 0.3280 & & & \\ & & & 0.7370 & -0.5910 & 0.3280 \\ & & & -0.5910 & -0.3280 & 0.7370 \\ & & & 0.3280 & 0.7370 & 0.5910 \end{bmatrix}$$

In this example we have *three* double eigenvalues. Because the corresponding components of $\mathbf{v}$ (v_i and v_{i+1}) are equal we define

$$\begin{aligned} G &= G(1,4,\pi/4)G(2,5,\pi/4)G(3,6,\pi/4) \\ &= \left[\begin{array}{ccc|ccc} 0.7071 & & & 0.7071 & & \\ & 0.7071 & & & 0.7071 & \\ & & 0.7071 & & & 0.7071 \\ \hline -0.7071 & & & 0.7071 & & \\ & -0.7071 & & & 0.7071 & \\ & & -0.7071 & & & 0.7071 \end{array}\right]. \end{aligned}$$

Then,

$$\begin{aligned} G^TQ_0^TTQ_0G &= G^TQ_0^TT_0Q_0G + G^T\mathbf{v}(G^T\mathbf{v})^T = D + G^T\mathbf{v}(G^T\mathbf{v})^T \\ &= \begin{bmatrix} 0.1981 & & & & & \\ & 1.5550 & & & & \\ & & 3.2470 & & & \\ & & & 0.1981 & & \\ & & & & 1.5550 & \\ & & & & & 3.2470 \end{bmatrix} + \begin{bmatrix} 1.0422 \\ -0.8358 \\ 0.4638 \\ 0.0000 \\ 0.0000 \\ 0.0000 \end{bmatrix} \begin{bmatrix} 1.0422 \\ -0.8358 \\ 0.4638 \\ 0.0000 \\ 0.0000 \\ 0.0000 \end{bmatrix}^T \end{aligned}$$

Therefore, (in this example) $\mathbf{e}_4$, $\mathbf{e}_5$, and $\mathbf{e}_6$ are eigenvectors of

$$D + G^T\mathbf{v}(G^T\mathbf{v})^T = D + G^T\mathbf{v}\mathbf{v}^TG$$

corresponding to the eigenvalues d_4, d_5, and d_6, respectively. The eigenvectors of T corresponding to these three eigenvalues are the last three columns of

$$Q_0G = \begin{bmatrix} 0.2319 & -0.4179 & 0.5211 & 0.5211 & -0.4179 & 0.2319 \\ 0.5211 & -0.2319 & -0.4179 & -0.4179 & -0.2319 & 0.5211 \\ 0.4179 & 0.5211 & 0.2319 & 0.2319 & 0.5211 & 0.4179 \\ -0.2319 & 0.4179 & -0.5211 & 0.5211 & -0.4179 & 0.2319 \\ -0.5211 & 0.2319 & 0.4179 & -0.4179 & -0.2319 & 0.5211 \\ -0.4179 & -0.5211 & -0.2319 & 0.2319 & 0.5211 & 0.4179 \end{bmatrix}.$$

4.5 The eigenvalue problem for $D + \rho\mathbf{v}\mathbf{v}^T$

We know that $\rho \neq 0$. Otherwise there is nothing to be done. Furthermore, after deflation, we know that all elements of $\mathbf{v}$ are nonzero and that the diagonal elements of D are all distinct, in fact,

$$|d_i - d_j| > C\varepsilon\|T\|.$$

We order the diagonal elements of D such that

$$d_1 < d_2 < \cdots < d_n.$$

Notice that this procedure permutes the elements of $\mathbf{v}$ as well. Let $(\lambda, \mathbf{x})$ be an eigenpair of

$$(D + \rho\mathbf{v}\mathbf{v}^T)\mathbf{x} = \lambda\mathbf{x}. \tag{4.12}$$

Then,

$$(D - \lambda I)\mathbf{x} = -\rho \mathbf{v}\mathbf{v}^T\mathbf{x}. \tag{4.13}$$

λ cannot be equal to one of the d_i. If $\lambda = d_k$ then the k-th element on the left of (4.13) vanishes. But then either $v_k = 0$ or $\mathbf{v}^T\mathbf{x} = 0$. The first cannot be true for our assumption about $\mathbf{v}$. If on the other hand $\mathbf{v}^T\mathbf{x} = 0$ then $(D - d_k I)\mathbf{x} = \mathbf{0}$. Thus $\mathbf{x} = \mathbf{e}_k$ and $\mathbf{v}^T\mathbf{e}_k = v_k = 0$, which cannot be true. Therefore $D - \lambda I$ is nonsingular and

$$\mathbf{x} = \rho(\lambda I - D)^{-1}\mathbf{v}(\mathbf{v}^T\mathbf{x}). \tag{4.14}$$

This equation shows that $\mathbf{x}$ is proportional to $(\lambda I - D)^{-1}\mathbf{v}$. If we require $\|\mathbf{x}\| = 1$ then

$$\mathbf{x} = \frac{(\lambda I - D)^{-1}\mathbf{v}}{\|(\lambda I - D)^{-1}\mathbf{v}\|}. \tag{4.15}$$

Multiplying (4.14) by $\mathbf{v}^T$ from the left we get

$$\mathbf{v}^T\mathbf{x} = \rho\mathbf{v}^T(\lambda I - D)^{-1}\mathbf{v}(\mathbf{v}^T\mathbf{x}). \tag{4.16}$$

Since $\mathbf{v}^T\mathbf{x} \neq 0$, λ is an eigenvalue of (4.12) if and only if

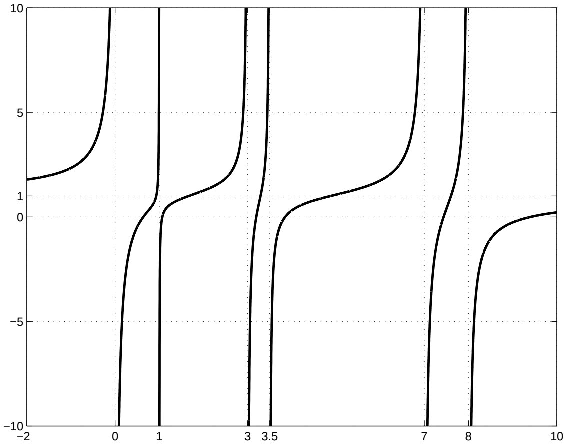


Figure 4.1: Graph of $1 + \frac{1}{0-\lambda} + \frac{0.2^2}{1-\lambda} + \frac{0.6^2}{3-\lambda} + \frac{0.5^2}{3.5-\lambda} + \frac{0.9^2}{7-\lambda} + \frac{0.8^2}{8-\lambda}$

$$\boxed{f(\lambda) := 1 - \rho\mathbf{v}^T(\lambda I - D)^{-1}\mathbf{v} = 1 - \rho\sum_{k=1}^{n}\frac{v_k^2}{\lambda - d_k} = 0.} \tag{4.17}$$

This equation is called **secular equation**. The secular equation has **poles** at the eigenvalues of D and **zeros** at the eigenvalues of $D + \rho\mathbf{v}\mathbf{v}^T$. Notice that

$$f'(\lambda) = \rho\sum_{k=1}^{n}\frac{v_k^2}{(\lambda - d_k)^2}.$$

Thus, the derivative of f is positive if $\rho > 0$ wherever it has a finite value. If $\rho < 0$ the derivative of f is negative (almost) everywhere. A typical graph of f with $\rho > 0$ is depicted in Fig. 4.1. (If ρ is negative the image can be flipped left to right.) The secular equation implies the **interlacing property** of the eigenvalues of D and of $D + \rho\mathbf{v}\mathbf{v}^T$,

$$d_1 < \lambda_1 < d_2 < \lambda_2 < \cdots < d_n < \lambda_n, \qquad \rho > 0. \tag{4.18}$$

or

$$\lambda_1 < d_1 < \lambda_2 < d_2 < \cdots < \lambda_n < d_n, \qquad \rho < 0. \tag{4.19}$$

So, we have to compute one eigenvalue in each of the intervals (d_i, d_{i+1}), $1 \leq i < n$, and a further eigenvalue in (d_n, ∞) or $(-\infty, d_1)$. The corresponding eigenvector is then given by (4.15). Evidently, these tasks are easy to parallelize.

Equations (4.17) and (4.15) can also been obtained from the relations

$$\begin{aligned}\begin{bmatrix} \frac{1}{\rho} & \mathbf{v}^T \\ \mathbf{v} & \lambda I - D \end{bmatrix} &= \begin{bmatrix} 1 & \mathbf{0}^T \\ \rho\mathbf{v} & I \end{bmatrix} \begin{bmatrix} \frac{1}{\rho} & \mathbf{0}^T \\ \mathbf{0} & \lambda I - D - \rho\mathbf{v}\mathbf{v}^T \end{bmatrix} \begin{bmatrix} 1 & \rho\mathbf{v}^T \\ \mathbf{0} & I \end{bmatrix} \\ &= \begin{bmatrix} 1 & \mathbf{v}^T(\lambda I - D)^{-1} \\ \mathbf{0} & I \end{bmatrix} \begin{bmatrix} \frac{1}{\rho} - \mathbf{v}^T(\lambda I - D)^{-1}\mathbf{v} & \mathbf{0}^T \\ \mathbf{0} & \lambda I - D \end{bmatrix} \begin{bmatrix} 1 & \mathbf{v}^T \\ (\lambda I - D)^{-1}\mathbf{v} & I \end{bmatrix}.\end{aligned}$$

These are simply block LDL^T factorizations of the first matrix. The first is the well-known one where the factorization is started with the $(1,1)$ block. The second is a ‘backward’ factorization that is started with the $(2,2)$ block. Because the determinants of the tridiagonal matrices are all unity, we have

$$\frac{1}{\rho}\mathsf{det}\,(\lambda I - D - \rho\mathbf{v}\mathbf{v}^T) = \frac{1}{\rho}(1 - \rho\mathbf{v}^T(\lambda I - D)^{-1}\mathbf{v})\mathsf{det}\,(\lambda I - D). \tag{4.20}$$

Denoting the eigenvalues of $D + \rho\mathbf{v}\mathbf{v}^T$ again by $\lambda_1 < \lambda_2 < \cdots < \lambda_n$ this implies

$$\begin{aligned}\prod_{j=1}^{n}(\lambda - \lambda_j) &= (1 - \rho\mathbf{v}^T(\lambda I - D)^{-1}\mathbf{v}) \prod_{j=1}^{n}(\lambda - d_j) \\ &= \left(1 - \rho\sum_{k=1}^{n}\frac{v_k^2}{\lambda - d_k}\right)\prod_{j=1}^{n}(\lambda - d_j) \\ &= \prod_{j=1}^{n}(\lambda - d_j) - \rho\sum_{k=1}^{n}v_k^2\prod_{j \neq k}(\lambda - d_j)\end{aligned} \tag{4.21}$$

Setting $\lambda = d_k$ gives

$$\prod_{j=1}^{n}(d_k - \lambda_j) = -\rho v_k^2 \prod_{\substack{j=1 \\ j \neq i}}^{n}(d_k - d_j) \tag{4.22}$$

or

$$\begin{aligned}v_k^2 &= \frac{-1}{\rho}\frac{\prod\limits_{j=1}^{n}(d_k - \lambda_j)}{\prod\limits_{\substack{j=1 \\ j \neq i}}^{n}(d_k - d_j)} = \frac{-1}{\rho}\frac{\prod\limits_{j=1}^{k-1}(d_k - \lambda_j)\prod\limits_{j=k}^{n}(\lambda_j - d_k)(-1)^{n-k+1}}{\prod\limits_{j=1}^{k-1}(d_k - d_j)\prod\limits_{j=k+1}^{n}(d_j - d_k)(-1)^{n-k}} \\ &= \frac{1}{\rho}\frac{\prod\limits_{j=1}^{k-1}(d_k - \lambda_j)}{\prod\limits_{j=1}^{k-1}(d_k - d_j)}\frac{\prod\limits_{j=k}^{n}(\lambda_j - d_k)}{\prod\limits_{j=k+1}^{n}(d_j - d_k)} > 0.\end{aligned} \tag{4.23}$$

Therefore, the quantity on the right side is positive, so

$$v_k = \sqrt{\frac{\prod\limits_{j=1}^{k-1}(d_k - \lambda_j)\prod\limits_{j=k}^{n}(\lambda_j - d_k)}{\rho\prod\limits_{j=1}^{k-1}(d_k - d_j)\prod\limits_{j=k+1}^{n}(d_j - d_k)}}. \tag{4.24}$$

(Similar arguments hold if $\rho < 0$.) Thus, we have the solution of the following **inverse eigenvalue problem**:

Given $D = \operatorname{diag}(d_1, \ldots, d_n)$ and values $\lambda_1, \ldots, \lambda_n$ that satisfy (4.18). Find a vector $\mathbf{v} = [v_1, \ldots, v_n]^T$ with positive components v_k such that the matrix $D + \mathbf{v}\mathbf{v}^T$ has the *prescribed* eigenvalues $\lambda_1, \ldots, \lambda_n$.

The solution is given by (4.24). The positivity of the v_k makes the solution unique.

4.6 Solving the secular equation

In this section we follow closely the exposition of Demmel [Dem97]. We consider the computation of the zero of $f(\lambda)$ in the interval (d_i, d_{i+1}). We assume that $\rho = 1$.

We may simply apply Newton's iteration to solve $f(\lambda) = 0$. However, if we look carefully at Fig. 4.1 then we notice that the tangent at certain points in (d_i, d_{i+1}) crosses the real axis outside this interval. This happens in particular if the weights v_i or v_{i+1} are small. Therefore that zero finder has to be adapted in such a way that it captures the poles at the interval endpoints. It is relatively straightforward to try the ansatz

$$h(\lambda) = \frac{c_1}{d_i - \lambda} + \frac{c_2}{d_{i+1} - \lambda} + c_3. \tag{4.25}$$

Notice that, given the coefficients c_1, c_2, and c_3, the equation $h(\lambda) = 0$ can easily be solved by means of the equivalent quadratic equation

$$c_1(d_{i+1} - \lambda) + c_2(d_i - \lambda) + c_3(d_i - \lambda)(d_{i+1} - \lambda) = 0. \tag{4.26}$$

This equation has two zeros. Precisely one of them is inside (d_i, d_{i+1}).

The coefficients c_1, c_2, and c_3 are computed in the following way. Let us assume that we have available an approximation λ_j to the zero in (d_i, d_{i+1}). We request that $h(\lambda_j) = f(\lambda_j)$ and $h'(\lambda_j) = f'(\lambda_j)$. The exact procedure is as follows. We write

$$f(\lambda) = 1 + \underbrace{\sum_{k=1}^{i} \frac{v_k^2}{d_k - \lambda}}_{\psi_1(\lambda)} + \underbrace{\sum_{k=i+1}^{n} \frac{v_k^2}{d_k - \lambda}}_{\psi_2(\lambda)} = 1 + \psi_1(\lambda) + \psi_2(\lambda). \tag{4.27}$$

$\psi_1(\lambda)$ is a sum of positive terms and $\psi_2(\lambda)$ is a sum of negative terms. Both $\psi_1(\lambda)$ and $\psi_2(\lambda)$ can be computed accurately, whereas adding them would likely provoke cancellation and loss of relative accuracy. We now choose c_1 and $\hat{c}_1$ such that

$$h_1(\lambda) := \hat{c}_1 + \frac{c_1}{d_i - \lambda} \text{ satisfies } h_1(\lambda_j) = \psi_1(\lambda_j) \text{ and } h_1'(\lambda_j) = \psi_1'(\lambda_j). \tag{4.28}$$

This means that the graphs of h_1 and of ψ_1 are tangent at $\lambda = \lambda_j$. This is similar to Newton's method. However in Newton's method a straight line is fitted to the given function. The coefficients in (4.28) are given by

$$c_1 = \psi_1'(\lambda_j)(d_i - \lambda_j)^2 > 0,$$

$$\hat{c}_1 = \psi_1(\lambda_j) - \psi_1'(\lambda_j)(d_i - \lambda_j) = \sum_{k=1}^{i} v_k^2 \frac{d_k - d_i}{(d_k - \lambda_j)^2} \le 0.$$

Similarly, the two constants c_2 and $\hat{c}_2$ are determined such that

$$h_2(\lambda) := \hat{c}_2 + \frac{c_2}{d_{i+1} - \lambda} \text{ satisfies } h_2(\lambda_j) = \psi_2(\lambda_j) \text{ and } h_2'(\lambda_j) = \psi_2'(\lambda_j) \tag{4.29}$$

with the coefficients

$$c_2 = \psi_2'(\lambda_j)(d_{i+1} - \lambda_j)^2 > 0,$$

$$\hat{c}_2 = \psi_2(\lambda_j) - \psi_2'(\lambda_j)(d_{i+1} - \lambda_j) = \sum_{k=i+1}^{n} v_k^2 \frac{d_k - d_{i+1}}{(d_k - \lambda)^2} \ge 0.$$

Finally, we set

$$h(\lambda) = 1 + h_1(\lambda) + h_2(\lambda) = \underbrace{(1 + \hat{c}_1 + \hat{c}_2)}_{c_3} + \frac{c_1}{d_i - \lambda} + \frac{c_2}{d_{i+1} - \lambda}. \tag{4.30}$$

This zerofinder is converging quadratically to the desired zero [Li94]. Usually 2 to 3 steps are sufficient to get the zero to machine precision. Therefore finding a zero only requires $\mathcal{O}(n)$ flops. Thus, finding all zeros costs $\mathcal{O}(n^2)$ floating point operations.

4.7 A first algorithm

We are now ready to give the divide and conquer algorithm, see Algorithm 4.1.

```
1:  Let $T \in \mathbb{C}^{n \times n}$ be a real symmetric tridiagonal matrix. This algorithm computes the spectral decomposition of $T = Q\Lambda Q^T$, where the diagonal $\Lambda$ is the matrix of eigenvalues and $Q$ is orthogonal.
2:  if $T$ is $1 \times 1$ then
3:      return ($\Lambda = T;\ Q = 1$)
4:  else
5:      Partition $T = \begin{bmatrix} T_1 & O \\ O & T_2 \end{bmatrix} + \rho \mathbf{u}\mathbf{u}^T$ according to (4.2)
6:      Call this algorithm with $T_1$ as input and $Q_1$, $\Lambda_1$ as output.
7:      Call this algorithm with $T_2$ as input and $Q_2$, $\Lambda_2$ as output.
8:      Form $D + \rho \mathbf{v}\mathbf{v}^T$ from $\Lambda_1, \Lambda_2, Q_1, Q_2$ according to (4.4)–(4.6).
9:      Find the eigenvalues $\Lambda$ and the eigenvectors $Q'$ of $D + \rho \mathbf{v}\mathbf{v}^T$.
10:     Form $Q = \begin{bmatrix} Q_1 & O \\ O & Q_2 \end{bmatrix} \cdot Q'$ which are the eigenvectors of $T$.
11:     return ($\Lambda;\ Q$)
12: end if
```

ALGORITHM 4.1: **The tridiagonal divide and conquer algorithm**

All steps except step 10 require $\mathcal{O}(n^2)$ operations to complete. The step 10 costs n^3 flops. Thus, the full divide and conquer algorithm, requires

$$\begin{aligned} T(n) &= n^3 + 2 \cdot T(n/2) = n^3 + 2\left(\frac{n}{2}\right)^3 + 4T(n/4) \\ &= n^3 + \frac{n^3}{4} + 4\left(\frac{n}{4}\right)^3 + 8T(n/8) = \cdots = \frac{4}{3}n^3. \end{aligned} \tag{4.31}$$

This *serial* complexity of the algorithm very often overestimates the computational costs of the algorithm due to significant deflation that is observed surprisingly often.

4.7.1 A numerical example

Let A be a 4×4 matrix

$$A = D + \mathbf{v}\mathbf{v}^T = \begin{bmatrix} 0 & & & \\ & 2-\beta & & \\ & & 2+\beta & \\ & & & 5 \end{bmatrix} + \begin{bmatrix} 1 \\ \beta \\ \beta \\ 1 \end{bmatrix} \begin{bmatrix} 1 & \beta & \beta & 1 \end{bmatrix}. \tag{4.32}$$

In this example (that is similar to one in [ST91]) we want to point at a problem that the divide and conquer algorithm possesses as it is given in Algorithm 4.1, namely the loss of orthogonality among eigenvectors.

Before we do some MATLAB tests let us look more closely at D and $\mathbf{v}$ in (4.32). This example becomes difficult to solve if β gets very small. In Figures 4.2 to 4.5 we see graphs of the function $f_\beta(\lambda)$ that appears

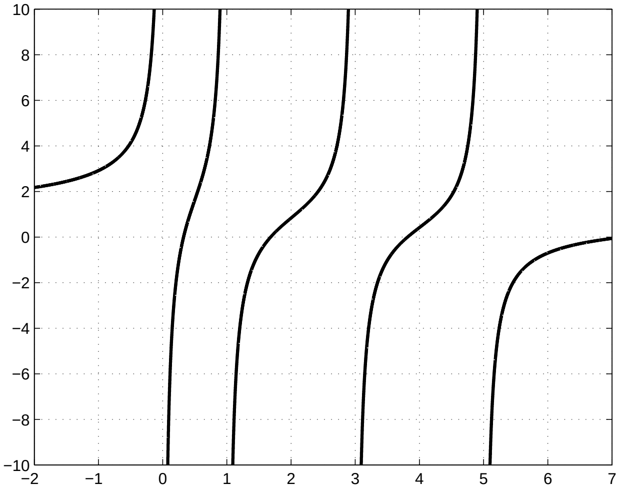


Figure 4.2: Secular equation corresponding to (4.32) for $\beta = 1$

in the secular equation for $\beta = 1$, $\beta = 0.1$, and $\beta = 0.01$. The critical zeros move towards 2 from both sides. The weights $v_2^2 = v_3^2 = \beta^2$ are however not so small that they should be deflated.

The following MATLAB code shows the problem. We execute the commands for $\beta = 10^{-k}$ for $k = 0, 1, 2, 4, 8$.

```
v = [1 beta beta 1]’;           % rank-1 modification
d = [0, 2-beta, 2+beta, 5]’;    % diagonal matrix

L = eig(diag(d) + v*v’)         % eigenvalues of the modified matrix
e = ones(4,1);
q = (d*e’-e*L’).\(v*e’);        % unnormalized eigenvectors cf. (5.15)

Q = sqrt(diag(q’*q));
q = q./(e*Q’);                  % normalized eigenvectors

norm(q’*q-eye(4))               % check for orthogonality
```

We do not bother how we compute the eigenvalues. We simply use MATLAB’s built-in function `eig`. We get the results of Table 4.1.

Table 4.1 Loss of orthogonality among the eigenvectors computed by (4.15)

β	λ_1	λ_2	λ_3	λ_4	$\|Q^TQ - I\|$
1	0.325651	1.682219	3.815197	7.176933	$5.6674 \cdot 10^{-16}$
0.1	0.797024	1.911712	2.112111	6.199153	$3.4286 \cdot 10^{-15}$
0.01	0.807312	1.990120	2.010120	6.192648	$3.9085 \cdot 10^{-14}$
10^{-4}	0.807418	1.999900	2.000100	6.192582	$5.6767 \cdot 10^{-12}$
10^{-8}	0.807418	1.99999999000000	2.00000001000000	6.192582	$8.3188 \cdot 10^{-08}$

We observe loss of orthogonality among the eigenvectors as the eigenvalues get closer and closer. This may not be surprising as we compute the eigenvectors by formula (4.15)

$$\mathbf{x} = \frac{(\lambda I - D)^{-1}\mathbf{v}}{\|(\lambda I - D)^{-1}\mathbf{v}\|}.$$

If $\lambda = \lambda_2$ and $\lambda = \lambda_3$ which are almost equal, $\lambda_2 \approx \lambda_3$ then intuitively one expects almost the same

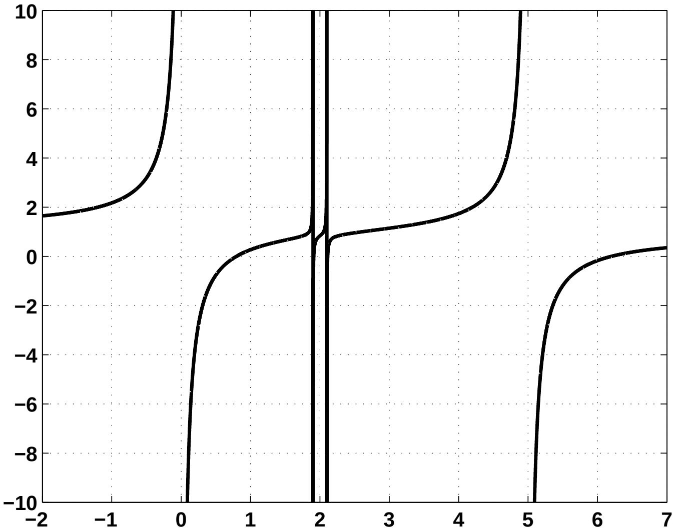


Figure 4.3: Secular equation corresponding to (4.32) for $\beta = 0.1$

eigenvectors. We have in fact

$$Q^TQ - I_4 = \begin{bmatrix} -2.2204\cdot 10^{-16} & 4.3553\cdot 10^{-8} & 1.7955\cdot 10^{-8} & -1.1102\cdot 10^{-16} \\ 4.3553\cdot 10^{-8} & 0 & -5.5511\cdot 10^{-8} & -1.8298\cdot 10^{-8} \\ 1.7955\cdot 10^{-8} & -5.5511\cdot 10^{-8} & -1.1102\cdot 10^{-16} & -7.5437\cdot 10^{-9} \\ -1.1102\cdot 10^{-16} & -1.8298\cdot 10^{-8} & -7.5437\cdot 10^{-9} & 0 \end{bmatrix}.$$

Orthogonality is lost only with respect to the vectors corresponding to the eigenvalues close to 2.

Already Dongarra and Sorensen [DS87] analyzed this problem. In their formulation they normalize the vector $\mathbf{v}$ of $D + \rho\mathbf{v}\mathbf{v}^T$ to have norm unity, $\|\mathbf{v}\| = 1$.

They formulated

Lemma 4.1. *Let*

$$\mathbf{q}_\lambda^T = \left(\frac{v_1}{d_1 - \lambda}, \frac{v_2}{d_2 - \lambda}, \ldots, \frac{v_n}{d_n - \lambda}\right)\left[\frac{\rho}{f'(\lambda)}\right]^{1/2}. \tag{4.33}$$

Then for any $\lambda, \mu \notin \{d_1, \ldots, d_n\}$ *we have*

$$|\mathbf{q}_\lambda^T\mathbf{q}_\mu| = \frac{1}{|\lambda - \mu|}\frac{|f(\lambda) - f(\mu)|}{[f'(\lambda)f'(\mu)]^{1/2}}. \tag{4.34}$$

Proof. Observe that

$$\frac{\lambda - \mu}{(d_j - \lambda)(d_j - \mu)} = \frac{1}{d_j - \lambda} - \frac{1}{d_j - \mu}.$$

Then the proof is straightforward. ■

Formula (4.34) indicates how problems may arise. In exact arithmetic, if λ and μ are eigenvalues then $f(\lambda) = f(\mu) = 0$. However, in floating point arithmetic this values may be small but nonzero, e.g., $\mathcal{O}(\varepsilon)$. If $|\lambda - \mu|$ is very small as well then we may have trouble! So, a remedy for the problem was for a long time to compute the eigenvalues in doubled precision, so that $f(\lambda) = \mathcal{O}(\varepsilon^2)$. This would counteract a potential $\mathcal{O}(\varepsilon)$ of $|\lambda - \mu|$.

This solution was quite unsatisfactory because doubled precision is in general very slow since it is implemented in software. It took a decade until a proper solution was found.

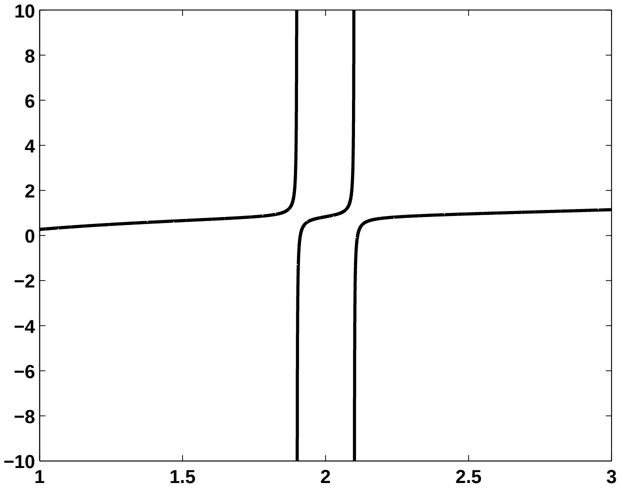


Figure 4.4: Secular equation corresponding to (4.32) for $\beta = 0.1$ for $1 \le \lambda \le 3$

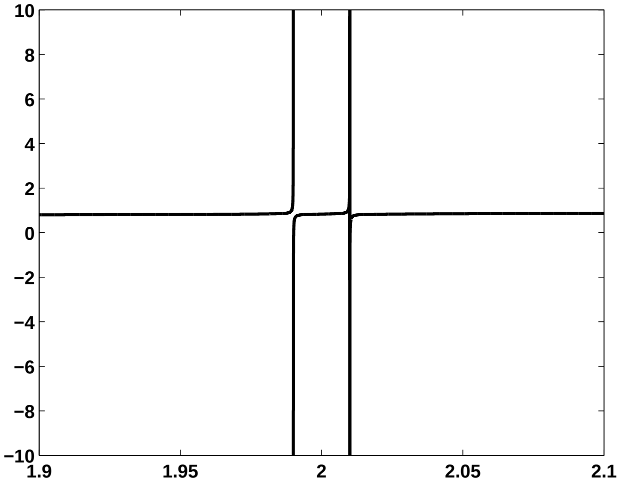


Figure 4.5: Secular equation corresponding to (4.32) for $\beta = 0.01$ for $1.9 \le \lambda \le 2.1$

4.8 The algorithm of Gu and Eisenstat

Computing eigenvector according to the formula

$$\mathbf{x} = \alpha(\lambda I - D)^{-1}\mathbf{v} = \alpha \begin{pmatrix} \frac{v_1}{\lambda - d_1} \\ \vdots \\ \frac{v_n}{\lambda - d_n} \end{pmatrix}, \qquad \alpha = \|(\lambda I - D)^{-1}\mathbf{v}\|, \tag{4.35}$$

is bound to fail if λ is very close to a pole d_k and the difference $\lambda - d_k$ has an error of size $\mathcal{O}(\varepsilon|d_k|)$ instead of only $\mathcal{O}(\varepsilon|d_k - \lambda|)$. To resolve this problem Gu and Eisenstat [GE94] found a trick that is at the same time ingenious and simple.

They observed that the v_k in (4.24) are very accurately determined by the data d_i and λ_i. Therefore, once the eigenvalues are computed *accurately* a vector $\hat{\mathbf{v}}$ could be computed such that the λ_i are *accurate* eigenvalues of $D + \hat{\mathbf{v}}\hat{\mathbf{v}}$. If $\hat{\mathbf{v}}$ approximates well the original $\mathbf{v}$ then the new eigenvectors will be the exact eigenvectors of a slightly modified eigenvalue problem, which is all we can hope for.

The zeros of the secular equation can be computed accurately by the method presented in section 4.6.

However, a shift of variables is necessary. In the interval (d_i, d_{i+1}) the origin of the real axis is moved to d_i if λ_i is closer to d_i than to d_{i+1}, i.e., if $f((d_i + d_{i+1})/2) > 0$. Otherwise, the origin is shifted to d_{i+1}. This shift of the origin avoids the computation of the smallest difference $d_i - \lambda$ (or $d_{i+1} - \lambda$) in (4.35), thus avoiding cancellation in this most sensitive quantity. Equation (4.26) can be rewritten as

$$\underbrace{(c_1\Delta_{i+1} + c_2\Delta_i + c_3\Delta_i\Delta_{i+1})}_{b} - \underbrace{(c_1 + c_2 + c_3(\Delta_i + \Delta_{i+1}))}_{-a}\,\eta + \underbrace{c_3}_{c}\,\eta^2 = 0, \tag{4.36}$$

where $\Delta_i = d_i - \lambda_j$, $\Delta_{i+1} = d_{i+1} - \lambda_j$, and $\lambda_{j+1} = \lambda_j + \eta$ is the next approximate zero. With equations (4.28)–(4.30) the coefficients in (4.36) get

$$\begin{aligned} a &= c_1 + c_2 + c_3(\Delta_i + \Delta_{i+1}) = (1 + \Psi_1 + \Psi_2)(\Delta_i + \Delta_{i+1}) - (\Psi_1' + \Psi_2')\Delta_i\Delta_{i+1}, \\ b &= c_1\Delta_{i+1} + c_2\Delta_i + c_3\Delta_i\Delta_{i+1} = \Delta_i\Delta_{i+1}(1 + \Psi_1 + \Psi_2), \\ c &= c_3 = 1 + \Psi_1 + \Psi_2 - \Delta_i\Psi_1' - \Delta_{i+1}\Psi_2'. \end{aligned} \tag{4.37}$$

If we are looking for a zero that is closer to d_i than to d_{i+1} then we move the origin to λ_j, i.e., we have e.g. $\Delta_i = -\lambda_j$. The solution of (4.36) that lies inside the interval is [Li94]

$$\eta = \begin{cases} \dfrac{a - \sqrt{a^2 - 4bc}}{2c}, & \text{if } a \leq 0, \\ \dfrac{2b}{a + \sqrt{a^2 - 4bc}}, & \text{if } a > 0. \end{cases} \tag{4.38}$$

The following algorithm shows how step 9 of the tridiagonal divide and conquer algorithm 4.1 must be implemented.

```
This algorithm stably computes the spectral decomposition of $D + \mathbf{v}\mathbf{v}^T = Q\Lambda Q^T$ where $D = \mathrm{diag}(d_1, \dots d_n)$, $\mathbf{v} = [v_1, \dots, v_n] \in \mathbb{R}^n$, $\Lambda = \mathrm{diag}(\lambda_1, \dots \lambda_n)$, and $Q = [\mathbf{q}_1, \dots, \mathbf{q}_n]$.
$d_{i+1} = d_n + \|\mathbf{v}\|^2$.
In each interval $(d_i, d_{i+1})$ compute the zero $\lambda_i$ of the secular equation $f(\lambda) = 0$.
Use the formula (4.24) to compute the vector $\hat{\mathbf{v}}$ such that the $\lambda_i$ are the ‘exact’ eigenvalues of $D + \hat{\mathbf{v}}\hat{\mathbf{v}}$.
In each interval $(d_i, d_{i+1})$ compute the eigenvectors of $D + \hat{\mathbf{v}}\hat{\mathbf{v}}$ according to (4.15),
    $\mathbf{q}_i = \frac{(\lambda_i I - D)^{-1}\hat{\mathbf{v}}}{\|(\lambda_i I - D)^{-1}\hat{\mathbf{v}}\|}.$
return $(\Lambda; Q)$
```

ALGORITHM 4.2: **A stable eigensolver for** $D + \mathbf{v}\mathbf{v}^T$

4.8.1 A numerical example [continued]

We continue the discussion of the example on page 48 where the eigenvalue problem of

$$A = D + \mathbf{v}\mathbf{v}^T = \begin{bmatrix} 0 & & & \\ & 2-\beta & & \\ & & 2+\beta & \\ & & & 5 \end{bmatrix} + \begin{bmatrix} 1 \\ \beta \\ \beta \\ 1 \end{bmatrix} \begin{bmatrix} 1 & \beta & \beta & 1 \end{bmatrix}. \tag{4.39}$$

The MATLAB code that we showed did not give orthogonal eigenvectors. We show in the following script that the formulae (4.24) really solve the problem.

```
dlam = zeros(n,1);
for k=1:n,
  [dlam(k), dvec(:,k)] = zerodandc(d,v,k);
```

```
end

V = ones(n,1);
for k=1:n,
  V(k) = prod(abs(dvec(k,:)))/prod(d(k) - d(1:k-1))/prod(d(k+1:n) - d(k));
  V(k) = sqrt(V(k));
end

Q = (dvec).\(V*e');
diagq = sqrt(diag(Q'*Q));
Q = Q./(e*diagq');

for k=1:n,
  if dlam(k)>0,
    dlam(k) = dlam(k) + d(k);
  else
    dlam(k) = d(k+1) + dlam(k);
  end
end

norm(Q'*Q-eye(n))
norm((diag(d) + v*v')*Q - Q*diag(dlam'))
```

A zero finder returns for each interval the quantity $\lambda_i - d_i$ and the vector $[d_1 - \lambda_i, \ldots, d_n - \lambda_i]^T$ to high precision. These vector elements have been computed as $(d_k - d_i) - (\lambda_i - d_i)$. The zerofinder of Li [Li94] has been employed here. At the end of this section we list the zerofinder written in MATLAB that was used here. The formulae (4.37) and (4.38) have been used to solve the quadratic equation (4.36). Notice that only one of the `while` loops is traversed, depending on if the zero is closer to the pole on the left or to the right of the interval. The v_k of formula (4.24) are computed next. Q contains the eigenvectors.

Table 4.2 Loss of orthogonality among the eigenvectors computed by the straightforward algorithm (I) and the Gu-Eisenstat approach (II)

β	Algorithm	$\|Q^TQ - I\|$	$\|AQ - Q\Lambda\|$
0.1	I	$3.4286 \cdot 10^{-15}$	$5.9460 \cdot 10^{-15}$
	II	$2.2870 \cdot 10^{-16}$	$9.4180 \cdot 10^{-16}$
0.01	I	$3.9085 \cdot 10^{-14}$	$6.9376 \cdot 10^{-14}$
	II	$5.5529 \cdot 10^{-16}$	$5.1630 \cdot 10^{-16}$
10^{-4}	I	$5.6767 \cdot 10^{-12}$	$6.3818 \cdot 10^{-12}$
	II	$2.2434 \cdot 10^{-16}$	$4.4409 \cdot 10^{-16}$
10^{-8}	I	$8.3188 \cdot 10^{-08}$	$1.0021 \cdot 10^{-07}$
	II	$2.4980 \cdot 10^{-16}$	$9.4133 \cdot 10^{-16}$

Again we ran the code for $\beta = 10^{-k}$ for $k = 0, 1, 2, 4, 8$. The numbers in Table 4.2 confirm that the new formulae are much more accurate than the straight forward ones. The norms of the errors obtained for the Gu-Eisenstat algorithm always are in the order of machine precision, i.e., 10^{-16}.

```
function [lambda,dl] = zerodandc(d,v,i)
% ZERODANDC - Computes eigenvalue lambda in the i-th interval
%             (d(i), d(i+1)) with Li's 'middle way' zero finder
%             dl is the n-vector [d(1..n) - lambda]

n = length(d);
di = d(i);
v = v.^2;
if i < n,
  di1 = d(i+1);  lambda = (di + di1)/2;
```

```
else
  di1 = d(n) + norm(v)ˆ2; lambda = di1;
end
eta = 1;
psi1 = sum((v(1:i).ˆ2)./(d(1:i) - lambda));
psi2 = sum((v(i+1:n).ˆ2)./(d(i+1:n) - lambda));

if  1 + psi1 + psi2 > 0,  %  zero is on the left half of the interval

  d = d - di;  lambda = lambda - di; di1 = di1 - di; di = 0;

  while  abs(eta) > 10*eps
    psi1 = sum(v(1:i)./(d(1:i) - lambda));
    psi1s = sum(v(1:i)./((d(1:i) - lambda)).ˆ2);

    psi2 = sum((v(i+1:n))./(d(i+1:n) - lambda));
    psi2s = sum(v(i+1:n)./((d(i+1:n) - lambda)).ˆ2);

    % Solve for zero
    Di = -lambda; Di1 = di1 - lambda;
    a = (Di + Di1)*(1 + psi1 + psi2) - Di*Di1*(psi1s + psi2s);
    b = Di*Di1*(1 + psi1 + psi2);
    c = (1 + psi1 + psi2) - Di*psi1s - Di1*psi2s;
    if a > 0,
      eta = (2*b)/(a + sqrt(aˆ2 - 4*b*c));
    else
      eta = (a - sqrt(aˆ2 - 4*b*c))/(2*c);
    end
    lambda = lambda + eta;
  end

else  %  zero is on the right half of the interval

  d = d - di1; lambda = lambda - di1; di = di - di1; di1 = 0;

  while  abs(eta) > 10*eps
    psi1 = sum(v(1:i)./(d(1:i) - lambda));
    psi1s = sum(v(1:i)./((d(1:i) - lambda)).ˆ2);

    psi2 = sum((v(i+1:n))./(d(i+1:n) - lambda));
    psi2s = sum(v(i+1:n)./((d(i+1:n) - lambda)).ˆ2);

    % Solve for zero
    Di = di - lambda; Di1 = - lambda;
    a = (Di + Di1)*(1 + psi1 + psi2) - Di*Di1*(psi1s + psi2s);
    b = Di*Di1*(1 + psi1 + psi2);
    c = (1 + psi1 + psi2) - Di*psi1s - Di1*psi2s;
    if a > 0,
      eta = (2*b)/(a + sqrt(aˆ2 - 4*b*c));
    else
      eta = (a - sqrt(aˆ2 - 4*b*c))/(2*c);
    end
    lambda = lambda + eta;
  end

end

dl = d - lambda;
```

```
    return
```

Bibliography

[ABB$^+$94] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide - Release 2.0*. SIAM, Philadelphia, PA, 1994. (Software and guide are available from Netlib at URL `http://www.netlib.org/lapack/`).

[Cup81] J. J. M. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numer. Math.*, 36:177–195, 1981.

[Dem97] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, PA, 1997.

[DS87] J. J. Dongarra and D. C. Sorensen. A fully parallel algorithm for the symmetric eigenvalue problem. *SIAM J. Sci. Stat. Comput.*, 8(2):s139–s154, 1987.

[GE94] M. Gu and S. C. Eisenstat. A stable and efficient algorithm for the rank-one modification of the symmetric eigenproblem. *SIAM J. Matrix Anal. Appl.*, 15:1266–1276, 1994.

[GE95] M. Gu and S. C. Eisenstat. A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem. *SIAM J. Matrix Anal. Appl.*, 16:172–191, 1995.

[Li94] R.-C. Li. Solving secular equations stably and efficiently. Technical Report CS-94-260, University of Tennessee, Knoxville, TN, November 1994. LAPACK Working Note No. 93.

[ST91] D. C. Sorensen and P. T. P. Tang. On the orthogonality of eigenvectors computed by divide-and-conquer techniques. *SIAM J. Numer. Anal.*, 28:1752–1775, 1991.

Chapter 5

LAPACK and the BLAS

5.1 LAPACK

(This section is essentially compiled from the LAPACK User's Guide [ABB$^+$94] that is available online from `http://www.netlib.org/lapack/lug/`.)

LAPACK [ABB$^+$94] is a library of Fortran 77 subroutines for solving the most commonly occurring problems in numerical linear algebra. It has been designed to be efficient on a wide range of modern high-performance computers. The name LAPACK is an acronym for **L**inear **A**lgebra **PACK**age.

LAPACK can solve systems of linear equations, linear least squares problems, eigenvalue problems and singular value problems. LAPACK can also handle many associated computations such as matrix factorizations or estimating condition numbers.

LAPACK contains **driver routines** for solving standard types of problems, **computational routines** to perform a distinct computational task, and **auxiliary routines** to perform a certain subtask or common low-level computation. Each driver routine typically calls a sequence of computational routines. Taken as a whole, the computational routines can perform a wider range of tasks than are covered by the driver routines. Many of the auxiliary routines may be of use to numerical analysts or software developers, so we have documented the Fortran source for these routines with the same level of detail used for the LAPACK routines and driver routines.

Dense and banded matrices are provided for, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices.

LAPACK is designed to give high efficiency on vector processors, high-performance "super-scalar" workstations, and shared memory multiprocessors. It can also be used satisfactorily on all types of scalar machines (PC's, workstations, mainframes). A distributed-memory version of LAPACK, **ScaLAPACK** [BCC$^+$97], has been developed for other types of parallel architectures (for example, massively parallel SIMD machines, or distributed memory machines).

LAPACK has been designed to supersede LINPACK [DBMS79] and EISPACK [SBD$^+$76, GBDM77], principally by restructuring the software to achieve much greater efficiency, where possible, on modern high-performance computers; also by adding extra functionality, by using some new or improved algorithms, and by integrating the two sets of algorithms into a unified package.

LAPACK routines are written so that as much as possible of the computation is performed by calls to the Basic Linear Algebra Subprograms (**BLAS**) [LHKK79, DDCHH88a, DCDH90]. Highly efficient machine-specific implementations of the BLAS are available for many modern high-performance computers. The BLAS enable LAPACK routines to achieve high performance with portable code.

The BLAS are not strictly speaking part of LAPACK, but Fortran 77 code for the BLAS is distributed with LAPACK, or can be obtained separately from `netlib` where "model implementations" are found.

The model implementation is not expected to perform as well as a specially tuned implementation on most high-performance computers – on some machines it may give much worse performance – but it allows users to run LAPACK codes on machines that do not offer any other implementation of the BLAS.

The complete LAPACK package or individual routines from LAPACK are freely available from the

World Wide Web or by anonymous ftp. The LAPACK homepage can be accessed via the URL `http://www.netlib.org/lapack/`.

5.2 BLAS

By 1976 it was clear that some standardization of basic computer operations on vectors was needed [LHKK79]. By then it was already known that coding procedures that worked well on one machine might work very poorly on others. In consequence of these observations, Lawson, Hanson, Kincaid and Krogh proposed a limited set of **B**asic **L**inear **A**lgebra **S**ubprograms (**BLAS**) to be (hopefully) optimized by hardware vendors, implemented in assembly language if necessary, that would form the basis of comprehensive linear algebra packages [LHKK79]. These so-called Level 1 BLAS consisted of vector operations and some attendant co-routines. The first major package which used these BLAS kernels was LINPACK [DBMS79]. Soon afterward, other major software libraries such as the IMSL library and NAG rewrote portions of their existing codes and structured new routines to use these BLAS. Early in their development, vector computers saw significant optimizations using the BLAS. Soon, however, such machines were clustered together in tight networks and somewhat larger kernels for numerical linear algebra were developed [DDCHH88a, DDCHH88b] to include matrix-vector operations (Level 2 BLAS). Additionally, FORTRAN compilers were by then optimizing vector operations as efficiently as hand coded Level 1 BLAS. Subsequently, in the late 1980s, distributed memory machines were in production and shared memory machines began to have significant numbers of processors. A further set of matrix-matrix operations was proposed [DCDH87] and soon standardized [DCDH90] to form a Level 3. The first major package for linear algebra which used the Level 3 BLAS was LAPACK [ABB$^+$94] and subsequently a scalable (to large numbers of processors) version was released as ScaLAPACK [BCC$^+$97]. Vendors focused on Level 1, Level 2, and Level 3 BLAS which provided an easy route to optimizing LINPACK, then LAPACK. LAPACK not only integrated pre-existing solvers and eigenvalue routines found in EISPACK [SBD$^+$76] (which did not use the BLAS) and LINPACK (which used Level 1 BLAS), but incorporated the latest dense and banded linear algebra algorithms available. It also used the Level 3 BLAS which were optimized by much vendor effort. Later, we will illustrate several BLAS routines. Conventions for different BLAS are indicated by

- A **root** operation. For example, `_axpy` for the operation

$$\mathbf{y} := a{\cdot}\mathbf{x} + \mathbf{y} \tag{5.1}$$

- A prefix (or combination prefix) to indicate the datatype of the operands, for example `saxpy` for **s**ingle precision `_axpy` operation, or `isamax` for the index of the maximum absolute element in an array of type **single**.

- a suffix if there is some qualifier, for example `cdot`**c** or `cdot`**u** for conjugated or unconjugated complex dot product, respectively:

$$\texttt{cdotc(n,x,1,y,1)} = \sum_{i=0}^{n-1} x_i \bar{y}_i$$
$$\texttt{cdotu(n,x,1,y,1)} = \sum_{i=0}^{n-1} x_i y_i$$

  where both $\mathbf{x}, \mathbf{y}$ are vectors of complex elements.

Tables 5.1 and 5.2 give the prefix/suffix and root combinations for the BLAS, respectively.

5.2.1 Typical performance numbers for the BLAS

Let us look at typical representations of all three levels of the BLAS, `daxpy`, `ddot`, `dgemv`, and `dgemm`, that perform some basic operations. Additionally, we look at the rank-1 update routine `dger`. An overview

Table 5.1 Basic Linear Algebra Subprogram prefix/suffix conventions.

Prefixes:	
S	REAL
D	DOUBLE PRECISION
C	COMPLEX
Z	DOUBLE COMPLEX
Suffixes:	
U	transpose
C	Hermitian conjugate

on the number of memory accesses and floating point operations is given in Table 5.3. The Level 1 BLAS comprise basic vector operations. A call of one of the Level 1 BLAS thus gives rise to $\mathcal{O}(n)$ floating point operations and $\mathcal{O}(n)$ memory accesses. Here, n is the vector length. The Level 2 BLAS comprise operations that involve matrices *and* vectors. If the involved matrix is n-by-n then both the memory accesses and the floating point operations are of $\mathcal{O}(n^2)$. In contrast, the Level 3 BLAS have a higher order of floating point operations than memory accesses. The most prominent operation of the Level 3 BLAS, matrix-matrix multiplication costs $\mathcal{O}(n^3)$ floating point operations while there are only $\mathcal{O}(n^2)$ reads and writes. The last column in Table 5.3 shows the crucial difference between the Level 3 BLAS and the rest.

Table 5.4 gives some performance numbers for the five BLAS of Table 5.3. Notice that the timer has a resolution of only 1 μsec! Therefore, the numbers in Table 5.4 have been obtained by timing a loop inside of which the respective function is called many times. The Mflop/s rates of the Level 1 BLAS `ddot` and `daxpy` quite precisely reflect the ratios of the memory accesses of the two routines, $2n$ vs. $3n$. The high rates are for vectors that can be held in the on-chip cache of 512 MB. The low 240 and 440 Mflop/s with the very long vectors are related to the memory bandwidth of about 1900 MB/s.

The Level 2 BLAS `dgemv` has about the same performance as `daxpy` if the matrix can be held in cache ($n = 100$). Otherwise it is considerably reduced. `dger` has a high volume of read and write operations, while the number of floating point operations is limited. This leads to a very low performance rate. The Level 3 BLAS `dgemm` performs at a good fraction of the peak performance of the processor (4.8Gflop/s). The performance increases with the problem size. We see from Table 5.3 that the ratio of computation to memory accesses increases with the problem size. This ratio is analogous to a volume to surface area effect.

5.3 Blocking

In the previous section we have seen that it is important to use Level 3 BLAS. However, in the algorithm we have treated so far, there were no blocks. For instance, in the reduction to Hessenberg form we applied Householder (elementary) reflectors from left and right to a matrix to introduce zeros in one of its columns.

The essential point here is to *gather* a number of reflectors to a single block transformation. Let $P_i = I - 2\mathbf{u}_i\mathbf{u}_i^*$, $i = 1, 2, 3$, be three Householder reflectors. Their product is

$$\begin{aligned} P = P_3P_2P_1 &= (I - 2\mathbf{u}_3\mathbf{u}_3^*)(I - 2\mathbf{u}_2\mathbf{u}_2^*)(I - 2\mathbf{u}_1\mathbf{u}_1^*) \\ &= I - 2\mathbf{u}_3\mathbf{u}_3^* - 2\mathbf{u}_2\mathbf{u}_2^* - 2\mathbf{u}_1\mathbf{u}_1^* + 4\mathbf{u}_3\mathbf{u}_3^*\mathbf{u}_2\mathbf{u}_2^* + 4\mathbf{u}_3\mathbf{u}_3^*\mathbf{u}_1\mathbf{u}_1^* + 4\mathbf{u}_2\mathbf{u}_2^*\mathbf{u}_1\mathbf{u}_1^* \\ &\qquad + 8\mathbf{u}_3\mathbf{u}_3^*\mathbf{u}_2\mathbf{u}_2^*\mathbf{u}_1\mathbf{u}_1^* \\ &= I - [\mathbf{u}_1\mathbf{u}_2\mathbf{u}_3] \begin{bmatrix} 2 & & \\ 4\mathbf{u}_2^*\mathbf{u}_1 & 2 & \\ 4\mathbf{u}_3^*\mathbf{u}_1 + 8(\mathbf{u}_3^*\mathbf{u}_2)(\mathbf{u}_2^*\mathbf{u}_1) & 4\mathbf{u}_3^*\mathbf{u}_2 & 2 \end{bmatrix} [\mathbf{u}_1\mathbf{u}_2\mathbf{u}_3]^*. \end{aligned} \tag{5.2}$$

So, if e.g. three rotations are to be applied on a matrix in blocked fashon, then the three Householder vectors $\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3$ have to be found first. To that end the rotations are first applied only on the first three columns of the matrix, see Fig. 5.1. Then, the blocked rotation is applied to the rest of the matrix.

Table 5.2 Summary of the Basic Linear Algebra Subroutines.

Level 1 BLAS	
`_rotg`, `_rot`	Generate/apply plane rotation
`_rotmg`, `_rotm`	Generate/apply modified plane rotation
`_swap`	Swap two vectors: $\mathbf{x} \leftrightarrow \mathbf{y}$
`_scal`	Scale a vector: $\mathbf{x} \leftarrow \alpha\mathbf{x}$
`_copy`	Copy a vector: $\mathbf{x} \leftarrow \mathbf{y}$
`_axpy`	_axpy operation: $\mathbf{y} \leftarrow \mathbf{y} + \alpha\mathbf{x}$
`_dot_`	Dot product: $s \leftarrow \mathbf{x} \cdot \mathbf{y} = \mathbf{x}^*\mathbf{y}$
`_nrm2`	2-norm: $s \leftarrow \lVert\mathbf{x}\rVert_\mathbf{2}$
`_asum`	1-norm: $s \leftarrow \lVert\mathbf{x}\rVert_\mathbf{1}$
`i_amax`	Index of largest vector element: first i such $\lvert x_i\rvert \geq \lvert x_k\rvert$ for all k
Level 2 BLAS	
`_gemv`, `_gbmv`	General (banded) matrix-vector multiply: $\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$
`_hemv`, `_hbmv`, `_hpmv`	Hermitian (banded, packed) matrix-vector multiply: $\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$
`_semv`, `_sbmv`, `_spmv`	Symmetric (banded, packed) matrix-vector multiply: $\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$
`_trmv`, `_tbmv`, `_tpmv`	Triangular (banded, packed) matrix-vector multiply: $\mathbf{x} \leftarrow A\mathbf{x}$
`_trsv`, `_tbsv`, `_tpsv`	Triangular (banded, packed) system solves (forward/backward substitution): $\mathbf{x} \leftarrow A^{-1}\mathbf{x}$
`_ger`, `_geru`, `_gerc`	Rank-1 updates: $A \leftarrow \alpha\mathbf{x}\mathbf{y}^* + A$
`_her`, `_hpr`, `_syr`, `_spr`	Hermitian/symmetric (packed) rank-1 updates: $A \leftarrow \alpha\mathbf{x}\mathbf{x}^* + A$
`_her2`, `_hpr2`,`_syr2`, `_spr2`	Hermitian/symmetric (packed) rank-2 updates: $A \leftarrow \alpha\mathbf{x}\mathbf{y}^* + \alpha^*\mathbf{y}\mathbf{x}^* + A$
Level 3 BLAS	
`_gemm`, `_symm`, `_hemm`	General/symmetric/Hermitian matrix-matrix multiply: $C \leftarrow \alpha AB + \beta C$
`_syrk`, `_herk`	Symmetric/Hermitian rank-k update: $C \leftarrow \alpha AA^* + \beta C$
`_syr2k`, `_her2k`	Symmetric/Hermitian rank-k update: $C \leftarrow \alpha AB^* + \alpha^*BA^* + \beta C$
`_trmm`	Multiple triangular matrix-vector multiplies: $B \leftarrow \alpha AB$
`_trsm`	Multiple triangular system solves: $B \leftarrow \alpha A^{-1}B$

Table 5.3 Number of memory references and floating point operations for vectors of length n.

	read	write	flops	flops / mem access
`ddot`	$2n$	1	$2n$	1
`daxpy`	$2n$	n	$2n$	2/3
`dgemv`	$n^2 + n$	n	$2n^2$	2
`dger`	$n^2 + 2n$	n^2	$2n^2$	1
`dgemm`	$2n^2$	n^2	$2n^3$	$2n/3$

Table 5.4 Some performance numbers for typical BLAS in Mflop/s for a 2.4 GHz Pentium 4.

	$n = 100$	500	2'000	10'000'000
`ddot`	1480	1820	1900	440
`daxpy`	1160	1300	1140	240
`dgemv`	1370	740	670	—
`dger`	670	330	320	—
`dgemm`	2680	3470	3720	—

Remark 5.1. *Notice that a similar situation holds for* ***Gaussian elimination*** *because*

$$\begin{bmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ l_{31} & & 1 & & \\ \vdots & & & \ddots & \\ l_{n1} & & & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & l_{32} & 1 & & \\ & \vdots & & \ddots & \\ & l_{n2} & & & 1 \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ l_{31} & l_{32} & 1 & & \\ \vdots & \vdots & & \ddots & \\ l_{n1} & l_{n2} & & & 1 \end{bmatrix}.$$

However, things are a complicated because of pivoting.

5.4 LAPACK solvers for the symmetric eigenproblems

To give a feeling how LAPACK is organized we consider solvers for the symmetric eigenproblem (SEP). Except for this problem there are driver routines for linear systems, least squares problems, nonsymmetric eigenvalue problems, the computation of the singular value decomposition (SVD).

The basic task of the symmetric eigenproblem routines is to compute values of λ and, optionally, corresponding vectors $\mathbf{z}$ for a given matrix A.

There are four types of driver routines for symmetric and Hermitian eigenproblems. Originally LAPACK had just the simple and expert drivers described below, and the other two were added after improved algorithms were discovered. Ultimately we expect the algorithm in the most recent driver (called RRR below) to supersede all the others, but in LAPACK 3.0 the other drivers may still be faster on some problems, so we retain them.

- A simple driver computes all the eigenvalues and (optionally) eigenvectors.
- An expert driver computes all or a selected subset of the eigenvalues and (optionally) eigenvectors. If few enough eigenvalues or eigenvectors are desired, the expert driver is faster than the simple driver.
- A divide-and-conquer driver solves the same problem as the simple driver. It is much faster than the simple driver for large matrices, but uses more workspace. The name divide-and-conquer refers to the underlying algorithm.
- A relatively robust representation (RRR) driver computes all or (in a later release) a subset of the eigenvalues, and (optionally) eigenvectors. It is the fastest algorithm of all (except for a few cases), and uses the least workspace. The name RRR refers to the underlying algorithm.

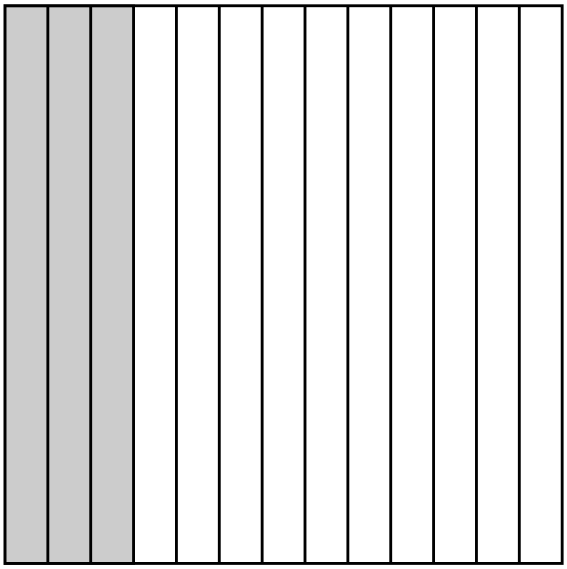


Figure 5.1: Blocking Householder reflections

This computation proceeds in the following stages:

1. The real symmetric or complex Hermitian matrix A is reduced to real tridiagonal form T. If A is real symmetric this decomposition is $A = QTQ^T$ with Q orthogonal and T symmetric tridiagonal. If A is complex Hermitian, the decomposition is $A = QTQ^H$ with Q unitary and T, as before, real symmetric tridiagonal.
2. Eigenvalues and eigenvectors of the real symmetric tridiagonal matrix T are computed. If all eigenvalues and eigenvectors are computed, this is equivalent to factorizing T as $T = S\Lambda S^T$, where S is orthogonal and Λ is diagonal. The diagonal entries of Λ are the eigenvalues of T, which are also the eigenvalues of A, and the columns of S are the eigenvectors of T; the eigenvectors of A are the columns of $Z = QS$, so that $A = Z\Lambda Z^T$ ($Z\Lambda Z^H$ when A is complex Hermitian).

In the real case, the decomposition $A = QTQ^T$ is computed by one of the routines `_sytrd`, `_sptrd`, or `_sbtrd`, depending on how the matrix is stored. The complex analogues of these routines are called `_hetrd`, `_hptrd`, and `_hbtrd`. The routine `_sytrd` (or `_hetrd`) represents the matrix Q as a product of elementary reflectors. The routine `_orgtr` (or in the complex case `_unmtr`) is provided to form Q explicitly; this is needed in particular before calling `_steqr` to compute all the eigenvectors of A by the QR algorithm. The routine `_ormtr` (or in the complex case `_unmtr`) is provided to multiply another matrix by Q without forming Q explicitly; this can be used to transform eigenvectors of T computed by `_stein`, back to eigenvectors of A.

For the names of the routines for packed and banded matrices, see [ABB$^+$94].

There are several routines for computing eigenvalues and eigenvectors of T, to cover the cases of computing some or all of the eigenvalues, and some or all of the eigenvectors. In addition, some routines run faster in some computing environments or for some matrices than for others. Also, some routines are more accurate than other routines.

`_steqr` This routine uses the implicitly shifted QR algorithm. It switches between the QR and QL variants in order to handle graded matrices. This routine is used to compute all the eigenvalues and eigenvectors.

`_sterf` This routine uses a square-root free version of the QR algorithm, also switching between QR and QL variants, and can only compute all the eigenvalues. This routine is used to compute all the eigenvalues and no eigenvectors.

`_stedc` This routine uses Cuppen's divide and conquer algorithm to find the eigenvalues and the eigenvectors. `_stedc` can be many times faster than `_steqr` for large matrices but needs more work space ($2n^2$ or $3n^2$). This routine is used to compute all the eigenvalues and eigenvectors.

`_stegr` This routine uses the relatively robust representation (RRR) algorithm to find eigenvalues and eigenvectors. This routine uses an LDL^T factorization of a number of translates $T - \sigma I$ of T, for one shift σ near each cluster of eigenvalues. For each translate the algorithm computes very accurate eigenpairs for the tiny eigenvalues. `_stegr` is faster than all the other routines except in a few cases, and uses the least workspace.

`_stebz` This routine uses bisection to compute some or all of the eigenvalues. Options provide for computing all the eigenvalues in a real interval or all the eigenvalues from the ith to the jth largest. It can be highly accurate, but may be adjusted to run faster if lower accuracy is acceptable.

`_stein` Given accurate eigenvalues, this routine uses inverse iteration to compute some or all of the eigenvectors.

5.5 Generalized Symmetric Definite Eigenproblems (GSEP)

Drivers are provided to compute all the eigenvalues and (optionally) the eigenvectors of the following types of problems:

1. $A\mathbf{z} = \lambda B\mathbf{z}$
2. $AB\mathbf{z} = \lambda\mathbf{z}$
3. $BA\mathbf{z} = \lambda\mathbf{z}$

where A and B are symmetric or Hermitian and B is positive definite. For all these problems the eigenvalues λ are real. The matrices Z of computed eigenvectors satisfy $Z^TAZ = \Lambda$ (problem types 1 and 3) or $Z^{-1}AZ^{-T} = I$ (problem type 2), where Λ is a diagonal matrix with the eigenvalues on the diagonal. Z also satisfies $Z^TBZ = I$ (problem types 1 and 2) or $Z^TB^{-1}Z = I$ (problem type 3).

There are three types of driver routines for generalized symmetric and Hermitian eigenproblems. Originally LAPACK had just the simple and expert drivers described below, and the other one was added after an improved algorithm was discovered.

- a simple driver computes all the eigenvalues and (optionally) eigenvectors.
- an expert driver computes all or a selected subset of the eigenvalues and (optionally) eigenvectors. If few enough eigenvalues or eigenvectors are desired, the expert driver is faster than the simple driver.
- a divide-and-conquer driver solves the same problem as the simple driver. It is much faster than the simple driver for large matrices, but uses more workspace. The name divide-and-conquer refers to the underlying algorithm.

5.6 An example of a LAPACK routines

The double precision subroutine `dsytrd.f` implements the reduction to tridiagonal form. We give it here in full length.

```
      SUBROUTINE DSYTRD( UPLO, N, A, LDA, D, E, TAU, WORK, LWORK, INFO )
*
*  -- LAPACK routine (version 3.0) --
*     Univ. of Tennessee, Univ. of California Berkeley, NAG Ltd.,
*     Courant Institute, Argonne National Lab, and Rice University
*     June 30, 1999
*
*     .. Scalar Arguments ..
```

```
      CHARACTER          UPLO
      INTEGER            INFO, LDA, LWORK, N
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, * ), D( * ), E( * ), TAU( * ),
     $                   WORK( * )
*     ..
*
*  Purpose
*  =======
*
*  DSYTRD reduces a real symmetric matrix A to real symmetric
*  tridiagonal form T by an orthogonal similarity transformation:
*  Q**T * A * Q = T.
*
*  Arguments
*  =========
*
*  UPLO    (input) CHARACTER*1
*          = ’U’:  Upper triangle of A is stored;
*          = ’L’:  Lower triangle of A is stored.
*
*  N       (input) INTEGER
*          The order of the matrix A.  N >= 0.
*
*  A       (input/output) DOUBLE PRECISION array, dimension (LDA,N)
*          On entry, the symmetric matrix A.  If UPLO = ’U’, the leading
*          N-by-N upper triangular part of A contains the upper
*          triangular part of the matrix A, and the strictly lower
*          triangular part of A is not referenced.  If UPLO = ’L’, the
*          leading N-by-N lower triangular part of A contains the lower
*          triangular part of the matrix A, and the strictly upper
*          triangular part of A is not referenced.
*          On exit, if UPLO = ’U’, the diagonal and first superdiagonal
*          of A are overwritten by the corresponding elements of the
*          tridiagonal matrix T, and the elements above the first
*          superdiagonal, with the array TAU, represent the orthogonal
*          matrix Q as a product of elementary reflectors; if UPLO
*          = ’L’, the diagonal and first subdiagonal of A are over-
*          written by the corresponding elements of the tridiagonal
*          matrix T, and the elements below the first subdiagonal, with
*          the array TAU, represent the orthogonal matrix Q as a product
*          of elementary reflectors. See Further Details.
*
*  LDA     (input) INTEGER
*          The leading dimension of the array A.  LDA >= max(1,N).
*
*  D       (output) DOUBLE PRECISION array, dimension (N)
*          The diagonal elements of the tridiagonal matrix T:
*          D(i) = A(i,i).
*
*  E       (output) DOUBLE PRECISION array, dimension (N-1)
*          The off-diagonal elements of the tridiagonal matrix T:
*          E(i) = A(i,i+1) if UPLO = ’U’, E(i) = A(i+1,i) if UPLO = ’L’.
*
*  TAU     (output) DOUBLE PRECISION array, dimension (N-1)
*          The scalar factors of the elementary reflectors (see Further
*          Details).
*
*  WORK    (workspace/output) DOUBLE PRECISION array, dimension (LWORK)
*          On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
*
*  LWORK   (input) INTEGER
*          The dimension of the array WORK.  LWORK >= 1.
*          For optimum performance LWORK >= N*NB, where NB is the
*          optimal blocksize.
*
```

```
*          If LWORK = -1, then a workspace query is assumed; the routine
*          only calculates the optimal size of the WORK array, returns
*          this value as the first entry of the WORK array, and no error
*          message related to LWORK is issued by XERBLA.
*
*  INFO    (output) INTEGER
*          = 0:  successful exit
*          < 0:  if INFO = -i, the i-th argument had an illegal value
*
*  Further Details
*  ===============
*
*  If UPLO = ’U’, the matrix Q is represented as a product of elementary
*  reflectors
*
*     Q = H(n-1) . . . H(2) H(1).
*
*  Each H(i) has the form
*
*     H(i) = I - tau * v * v’
*
*  where tau is a real scalar, and v is a real vector with
*  v(i+1:n) = 0 and v(i) = 1; v(1:i-1) is stored on exit in
*  A(1:i-1,i+1), and tau in TAU(i).
*
*  If UPLO = ’L’, the matrix Q is represented as a product of elementary
*  reflectors
*
*     Q = H(1) H(2) . . . H(n-1).
*
*  Each H(i) has the form
*
*     H(i) = I - tau * v * v’
*
*  where tau is a real scalar, and v is a real vector with
*  v(1:i) = 0 and v(i+1) = 1; v(i+2:n) is stored on exit in A(i+2:n,i),
*  and tau in TAU(i).
*
*  The contents of A on exit are illustrated by the following examples
*  with n = 5:
*
*  if UPLO = ’U’:                       if UPLO = ’L’:
*
*    (  d   e   v2  v3  v4 )              (  d                  )
*    (      d   e   v3  v4 )              (  e   d              )
*    (          d   e   v4 )              (  v1  e   d          )
*    (              d   e  )              (  v1  v2  e   d      )
*    (                  d  )              (  v1  v2  v3  e   d  )
*
*  where d and e denote diagonal and off-diagonal elements of T, and vi
*  denotes an element of the vector defining H(i).
*
*  =====================================================================
*
*     .. Parameters ..
      DOUBLE PRECISION   ONE
      PARAMETER          ( ONE = 1.0D+0 )
*     ..
*     .. Local Scalars ..
      LOGICAL            LQUERY, UPPER
      INTEGER            I, IINFO, IWS, J, KK, LDWORK, LWKOPT, NB,
     $                   NBMIN, NX
*     ..
*     .. External Subroutines ..
      EXTERNAL           DLATRD, DSYR2K, DSYTD2, XERBLA
*     ..
*     .. Intrinsic Functions ..
```

```
      INTRINSIC          MAX
*     ..
*     .. External Functions ..
      LOGICAL            LSAME
      INTEGER            ILAENV
      EXTERNAL           LSAME, ILAENV
*     ..
*     .. Executable Statements ..
*
*     Test the input parameters
*
      INFO = 0
      UPPER = LSAME( UPLO, 'U' )
      LQUERY = ( LWORK.EQ.-1 )
      IF( .NOT.UPPER .AND. .NOT.LSAME( UPLO, 'L' ) ) THEN
         INFO = -1
      ELSE IF( N.LT.0 ) THEN
         INFO = -2
      ELSE IF( LDA.LT.MAX( 1, N ) ) THEN
         INFO = -4
      ELSE IF( LWORK.LT.1 .AND. .NOT.LQUERY ) THEN
         INFO = -9
      END IF
*
      IF( INFO.EQ.0 ) THEN
*
*        Determine the block size.
*
         NB = ILAENV( 1, 'DSYTRD', UPLO, N, -1, -1, -1 )
         LWKOPT = N*NB
         WORK( 1 ) = LWKOPT
      END IF
*
      IF( INFO.NE.0 ) THEN
         CALL XERBLA( 'DSYTRD', -INFO )
         RETURN
      ELSE IF( LQUERY ) THEN
         RETURN
      END IF
*
*     Quick return if possible
*
      IF( N.EQ.0 ) THEN
         WORK( 1 ) = 1
         RETURN
      END IF
*
      NX = N
      IWS = 1
      IF( NB.GT.1 .AND. NB.LT.N ) THEN
*
*        Determine when to cross over from blocked to unblocked code
*        (last block is always handled by unblocked code).
*
         NX = MAX( NB, ILAENV( 3, 'DSYTRD', UPLO, N, -1, -1, -1 ) )
         IF( NX.LT.N ) THEN
*
*           Determine if workspace is large enough for blocked code.
*
            LDWORK = N
            IWS = LDWORK*NB
            IF( LWORK.LT.IWS ) THEN
*
*              Not enough workspace to use optimal NB:  determine the
*              minimum value of NB, and reduce NB or force use of
*              unblocked code by setting NX = N.
*
```

```
               NB = MAX( LWORK / LDWORK, 1 )
               NBMIN = ILAENV( 2, 'DSYTRD', UPLO, N, -1, -1, -1 )
               IF( NB.LT.NBMIN )
     $            NX = N
            END IF
         ELSE
            NX = N
         END IF
      ELSE
         NB = 1
      END IF
*
      IF( UPPER ) THEN
*
*        Reduce the upper triangle of A.
*        Columns 1:kk are handled by the unblocked method.
*
         KK = N - ( ( N-NX+NB-1 ) / NB )*NB
         DO 20 I = N - NB + 1, KK + 1, -NB
*
*           Reduce columns i:i+nb-1 to tridiagonal form and form the
*           matrix W which is needed to update the unreduced part of
*           the matrix
*
            CALL DLATRD( UPLO, I+NB-1, NB, A, LDA, E, TAU, WORK,
     $                   LDWORK )
*
*           Update the unreduced submatrix A(1:i-1,1:i-1), using an
*           update of the form:  A := A - V*W' - W*V'
*
            CALL DSYR2K( UPLO, 'No transpose', I-1, NB, -ONE, A( 1, I ),
     $                   LDA, WORK, LDWORK, ONE, A, LDA )
*
*           Copy superdiagonal elements back into A, and diagonal
*           elements into D
*
            DO 10 J = I, I + NB - 1
               A( J-1, J ) = E( J-1 )
               D( J ) = A( J, J )
   10       CONTINUE
   20    CONTINUE
*
*        Use unblocked code to reduce the last or only block
*
         CALL DSYTD2( UPLO, KK, A, LDA, D, E, TAU, IINFO )
      ELSE
*
*        Reduce the lower triangle of A
*
         DO 40 I = 1, N - NX, NB
*
*           Reduce columns i:i+nb-1 to tridiagonal form and form the
*           matrix W which is needed to update the unreduced part of
*           the matrix
*
            CALL DLATRD( UPLO, N-I+1, NB, A( I, I ), LDA, E( I ),
     $                   TAU( I ), WORK, LDWORK )
*
*           Update the unreduced submatrix A(i+ib:n,i+ib:n), using
*           an update of the form:  A := A - V*W' - W*V'
*
            CALL DSYR2K( UPLO, 'No transpose', N-I-NB+1, NB, -ONE,
     $                   A( I+NB, I ), LDA, WORK( NB+1 ), LDWORK, ONE,
     $                   A( I+NB, I+NB ), LDA )
*
*           Copy subdiagonal elements back into A, and diagonal
*           elements into D
```

```
*
            DO 30 J = I, I + NB - 1
               A( J+1, J ) = E( J )
               D( J ) = A( J, J )
   30       CONTINUE
   40    CONTINUE
*
*        Use unblocked code to reduce the last or only block
*
         CALL DSYTD2( UPLO, N-I+1, A( I, I ), LDA, D( I ), E( I ),
     $                TAU( I ), IINFO )
      END IF
*
      WORK( 1 ) = LWKOPT
      RETURN
*
*     End of DSYTRD
*
      END
```

Notice that most of the lines (indicated by '$*$') contain comments. The initial comment lines also serve as manual pages. Notice that the code only looks at one half (upper or lower triangle) of the symmetric input matrix. The other triangle is used to store the Householder vectors. These are normed such that the first component is one,

$$I - 2\mathbf{u}\mathbf{u}^* = I - 2|u_1|^2(\mathbf{u}/u_1)(\mathbf{u}/u_1)^* = I - \tau\mathbf{v}\mathbf{v}^*.$$

In the main loop of `dsytrd` there is a call to a subroutine `dlatrd` that generates a block reflektor. (The blocksize is NB.) Then the block reflector is applied by the routine `dsyr2k`.

Directly after the loop there is a call to the 'unblocked `dsytrd`' named `dsytd2` to deal with the first/last few (<NB) rows/columns of the matrix. This excerpt concerns the situation when the upper triangle of the matrix A is stored. In that routine the mentioned loop looks very much the way we derived the formulae.

```
      ELSE
*
*        Reduce the lower triangle of A
*
         DO 20 I = 1, N - 1
*
*           Generate elementary reflector H(i) = I - tau * v * v'
*           to annihilate A(i+2:n,i)
*
            CALL DLARFG( N-I, A( I+1, I ), A( MIN( I+2, N ), I ), 1,
     $                   TAUI )
            E( I ) = A( I+1, I )
*
            IF( TAUI.NE.ZERO ) THEN
*
*              Apply H(i) from both sides to A(i+1:n,i+1:n)
*
               A( I+1, I ) = ONE
*
*              Compute  x := tau * A * v  storing y in TAU(i:n-1)
*
               CALL DSYMV( UPLO, N-I, TAUI, A( I+1, I+1 ), LDA,
     $                     A( I+1, I ), 1, ZERO, TAU( I ), 1 )
*
*              Compute  w := x - 1/2 * tau * (x'*v) * v
*
               ALPHA = -HALF*TAUI*DDOT( N-I, TAU( I ), 1, A( I+1, I ),
     $                 1 )
               CALL DAXPY( N-I, ALPHA, A( I+1, I ), 1, TAU( I ), 1 )
*
```

```
*              Apply the transformation as a rank-2 update:
*                 A := A - v * w’ - w * v’
*
               CALL DSYR2( UPLO, N-I, -ONE, A( I+1, I ), 1, TAU( I ), 1,
     $                     A( I+1, I+1 ), LDA )
*
               A( I+1, I ) = E( I )
            END IF
            D( I ) = A( I, I )
            TAU( I ) = TAUI
   20    CONTINUE
         D( N ) = A( N, N )
      END IF
```

Bibliography

[ABB$^+$94] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide - Release 2.0*. SIAM, Philadelphia, PA, 1994. (Software and guide are available from Netlib at URL `http://www.netlib.org/lapack/`).

[BCC$^+$97] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997. (Software and guide are available at URL `http://www.netlib.org/scalapack/`).

[DBMS79] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM, Philadelphia, PA, 1979.

[DCDH87] J. J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A proposal for a set of level 3 basic linear algebra subprograms. *ACM SIGNUM Newsletter*, 22(3), September 1987.

[DCDH90] J. J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990.

[DDCHH88a] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14:1–17, 1988.

[DDCHH88b] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms: Model implementation and test programs. *ACM Transactions on Mathematical Software*, 14:18–32, 1988.

[GBDM77] B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler. *Matrix Eigensystem Routines – EISPACK Guide Extension*. Lecture Notes in Computer Science 51. Springer-Verlag, Berlin, 1977.

[LHKK79] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.*, 5:308–325, 1979.

[SBD$^+$76] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines – EISPACK Guide*. Lecture Notes in Computer Science 6. Springer-Verlag, Berlin, 2nd edition, 1976.

Part II

Sparse Linear Algebra

Chapter 6

Finite Element Discretisations of Elliptic PDEs

The goal of this chapter is to introduce you to the use of the *finite element method* (FEM) to solve boundary value problems that are formulated by *partial differential equations* (PDEs). To this end, we will have a look at an example problem which will allow us to demonstrate the typical use of the FEM and of the related tools. This chapter is meant as a quick overview of the subject, for a detailed description of the method we refer to [Sch91], [Bra97], [AB01] and [CL91].

6.1 Model Problem

As an example of a PDE problem, we consider a 2D electrostatic problem which is typically described by using the electrostatic scalar potential $\phi(x,y)$, the electric field $\mathbf{E}(x,y)$ induced by the potential, the charge density $\rho(x,y)$ and the material's dielectricity $\epsilon(x,y)$. In many applications one is interested in finding the electric field distribution $\mathbf{E}(x,y)$ in the interior of a domain $\Omega \subset \mathbb{R}^2$ for given $\rho(x,y)$ and $\epsilon(x,y)$ and a set of (boundary) conditions.

From the conservativity property of static fields expressible as

$$\mathbf{E}(x,y) = -\textbf{grad}\phi(x,y) \tag{6.1}$$

and the Gaussian law

$$\text{div}\big(\epsilon(x,y)\mathbf{E}(x,y)\big) = \rho(x,y) \tag{6.2}$$

one obtains

$$-\text{div}\big(\epsilon(x,y)\textbf{grad}\phi(x,y)\big) = \rho(x,y), \tag{6.3}$$

also referred to as *Poisson problem*, see [Jac99]. This is the equation we are aiming to solve.

In order to be well posed, Equation (6.3) requires an appropriate set of boundary conditions. Typically, one either prescribes function values $\phi(x,y) = f(x,y)$ and/or (directional) derivatives $\partial_{\mathbf{n}} = g(x,y)$ along the boundary $\Gamma = \partial\Omega$.

Let us now turn our attention to a concrete problem. As shown in Figure 6.1, we consider a dielectric ring immersed in an electric field induced by an artificially applied potential difference. We would like to verify numerically, that the field in the interior of the ring is heavily attenuated by the field that is established in the ring itself — an electric shielding effect known as Faraday effect. In terms of the quantities introduced above, we thus tackle the problem

$$\begin{aligned} -\text{div}\big(\epsilon(x,y)\textbf{grad}\phi(x,y)\big) &= \rho(x,y) \quad (x,y) \in \Omega && (6.4)\\ \phi(x,y) &= f(x,y) \quad (x,y) \in \Gamma_D && (6.5)\\ \partial_{\mathbf{n}}\phi(x,y) &= g(x,y) \quad (x,y) \in \Gamma_N && (6.6) \end{aligned}$$

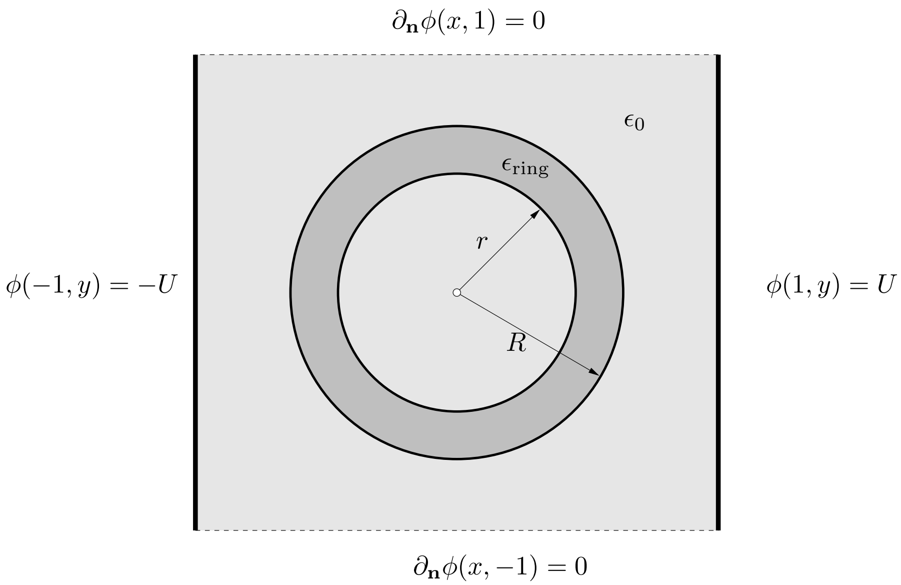


Figure 6.1: Dielectric ring immersed in an electric field induced by an applied potential difference (along the left and the right-hand boundary segments). The Neumann conditions imposed along the top and the bottom boundary segment enforce a horizontal field. Note that the ring is assumed to be surrounded by vacuum.

where Γ_D (Γ_N), the *Dirichlet* (*Neumann*) boundary, denotes the portion of the boundary on which the function (derivative) value is prescribed and $\Gamma = \Gamma_D \cup \Gamma_N$. In general, these problems are too difficult to be solved exactly, which is mainly due to the arbitrariness of the domain geometry. In the following we will therefore be concerned with recasting this formulation into a form which is more appropriate for numerical handling.

6.1.1 Derivation of a Weak Form

A very popular tool to approximately solve PDE problems such as (6.4), (6.5), (6.6) is the so called FEM. The first step when applying this method consists in rewriting the governing PDE in its *weak form* [CH53]. This weak form offers the advantage of having a lower order of derivatives. To obtain it, we multiply (6.4) with a (still arbitrary) *test function* $v(x,y)$ and integrate over the entire domain, obtaining

$$-\int_\Omega v(x,y)\mathrm{div}\big((\epsilon(x,y)\textbf{grad}\phi(x,y))\big)\,d\Omega = \int_\Omega v(x,y)\rho(x,y)\,d\Omega. \tag{6.7}$$

By making use of the identity $f\mathrm{div}\big(()\mathbf{F}) = \mathrm{div}\big((f\mathbf{F})\big) - \textbf{grad}(f)^{\mathrm{T}}\mathbf{F}$, see [BS79], we transform the left-hand integral and obtain

$$\int_\Omega \Big\{\textbf{grad}v(x,y)^{\mathrm{T}}\epsilon(x,y)\textbf{grad}\phi(x,y) - \mathrm{div}\big((v(x,y)\epsilon(x,y)\textbf{grad}\phi(x,y))\big)\Big\}\;d\Omega. \tag{6.8}$$

Finally, by applying the divergence theorem to the second term, we obtain

$$\int_\Omega \textbf{grad}v(x,y)^{\mathrm{T}}\epsilon(x,y)\textbf{grad}\phi(x,y)\,d\Omega - \int_\Gamma v(x,y)\epsilon(x,y)\textbf{grad}\phi(x,y)^{\mathrm{T}}\mathbf{n}\,d\Gamma \tag{6.9}$$

where $\mathbf{n}$ denotes the outward normal vector of the domain boundary. By replacing the left-hand side of (6.7) with (6.9) we obtain a weak representation of equation (6.4) which reads

$$\int_\Omega \textbf{grad}v^{\mathrm{T}}\epsilon\,\textbf{grad}\phi\,d\Omega - \int_\Gamma v\,\epsilon\,\textbf{grad}\phi^{\mathrm{T}}\mathbf{n}\,d\Gamma = \int_\Omega v\rho\,d\Omega, \tag{6.10}$$

where we have omitted the coordinates for the sake of readability. Before we continue, let us investigate this equation a little further. As specified by equations (6.5) and (6.6), we impose two types of boundary conditions namely Dirichlet and Neumann ones. But what if we didn't impose any boundary conditions?

Not imposing any conditions would be equivalent to solve the problem specified in (6.10) without considering the boundary integral. But this would be equivalent to the tacit assumption of having a vanishing $\mathbf{grad}\phi^{\mathrm{T}}\mathbf{n}$ term. In fact, this *natural* boundary condition $\mathbf{grad}\phi^{\mathrm{T}}\mathbf{n} = \mathbf{0}$ arises in boundary regions, where no conditions whatsoever are imposed. Note that this stands in stark contrast to *essential* boundary conditions (such as the Dirichlet ones), which do not come naturally and hence have to be enforced artificially.

For the sake of simplicity, we continue by introducing appropriate *bilinear forms*, allowing a simpler formulation of the weak form derived above. We introduce

$$a(v,\phi) - b_{\Gamma}(v,\phi) = l(v,\rho), \tag{6.11}$$

where $a(\cdot,\cdot)$, $b_{\Gamma}(\cdot,\cdot)$ and $l(\cdot,\cdot)$ correspond to the first, second and third integral in (6.10). Having done so, the second step that is usually performed concerns the boundary conditions. Here, the main idea consists in representing the function one is interested in by a combination of the type

$$\phi(x,y) = \phi_0(x,y) + u(x,y), \tag{6.12}$$

where $\phi_0(x,y)$ satisfies the Dirichlet condition (6.5) and $u(x,y)$, as well as the test function $v(x,y)$, is assumed to satisfy *homogeneous* boundary conditions on the segments Γ_D but are arbitrary otherwise. By plugging the ansatz (6.12) into the weak form (6.11) one obtains

$$a(v,\phi_0) + a(v,u) - \underbrace{b_{\Gamma_D}(v,\phi)}_{=0} - b_{\Gamma_N}(v,\phi) = l(v,\rho), \tag{6.13}$$

where one expressions vanishes due to the homogeneity condition imposed on $v(x,y)$. By rearranging the expressions and by replacing the conditions (6.5) and (6.6) when appropriate we finally obtain

$$a(v,u) = l(v,\rho) - a(v,\phi_0) + b_{\Gamma_N}(v,\phi). \tag{6.14}$$

6.1.2 Of Meshes, Element Functions and Discrete Operators

As mentioned before, the original problem is usually too difficult to be solved exactly, be it in strong or in weak form. Hence, we approximate the sought solution $u(x,y)$ and the test function $v(x,y)$ by means of linear combinations of simple functions $\psi_i(x,y)$. By doing so, i.e. by letting

$$u(x,y) = \sum_i s_i \psi_i(x,y) \quad \text{and} \quad v(x,y) = \sum_i t_i \psi_i(x,y) \tag{6.15}$$

the weak form (6.14) becomes

$$\sum_i a(\psi_j,\psi_i)\, s_i = l(\psi_j,\rho) - a(\psi_j,\phi_0) + b_{\Gamma_N}(\psi_j,\phi), \tag{6.16}$$

which has to hold for any choice of $v(x,y)$ and hence for any single ψ_j (satisfying homogeneous Dirichlet boundary conditions on Γ_D). These equations, one for each index j, can be rewritten as matrix-vector and vector-vector products, finally leading to

$$\mathbf{A}\mathbf{s} = \mathbf{l} - \mathbf{a} + \mathbf{b}. \tag{6.17}$$

One immediately sees that the matrix $\mathbf{A}$ is defined as

$$\mathbf{A}_{i,j} = a(\psi_j,\psi_i), \tag{6.18}$$

and given that the bilinear form is symmetric, so is the matrix $\mathbf{A}$. Choosing the functions ψ_i such that their supports will scarcely overlap will induce a matrix $\mathbf{A}$ with many zero entries, i.e. a *sparse matrix*.

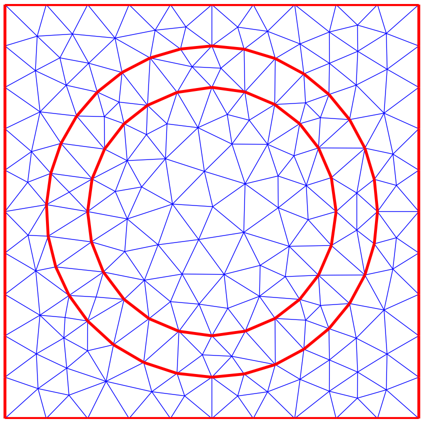


Figure 6.2: Discretisation of the original domain Ω by means of a triangle mesh T_h. Curvilinear segments are thereby approximated by polygonal domains, this way introducing a model error.

We still need to define a procedure to specify the ψ_i's. This can be done by discretising the computational domain Ω by means of a triangulation T_h, a set of polygons (in our case only triangles) whose union approximates Ω, see Figure 6.2.

It is easy to see that by placing a so called *hat function* ψ_i, see [Sch91], on each vertex of the mesh, one can readily generate piecewise linear function approximations, enforce the boundary conditions by setting the coefficients along the boundary appropriately (as we will show later) and at the same time, this choice guarantees a disjointness of the supports of ψ_i (to some extent), see Figure 6.3.

Now that we have implicitly defined the functions ψ_i as piecewise polynomial functions with a support extending over a small polygonal domain (also referred to as element, thence the name *element functions*), it is time to construct the matrix $\mathbf{A}$ given in (6.18). To do so, we start by replacing the domain Ω by means of the aforementioned polygonal approximation T_h and, consequently, all domain integrals with sums of integrals over the single triangles τ_k representing T_h, i.e.

$$\mathbf{A}_{i,j} = a(\psi_j, \psi_i) = \int_\Omega \mathbf{grad}\psi_j{}^{\mathrm{T}} \epsilon\, \mathbf{grad}\psi_i \, d\Omega \approx \sum_k \int_{\tau_k} \mathbf{grad}\psi_j{}^{\mathrm{T}} \epsilon\, \mathbf{grad}\psi_i \, d\tau_k, \tag{6.19}$$

where $\Omega \approx T_h = \bigcup_k \tau_k$. Given that the supports of the element functions are disjoint in most cases, it would be highly inefficient to iterate over all indices i, j and k in order to construct $\mathbf{A}$. Instead, it is more convenient to iterate over one triangle τ_k at a time and consider the element functions ψ_i and ψ_j whose support overlap with τ_k. Since there are exactly three element functions affecting each triangle, the construction process is considerably improved.

In order to carry out the construction process, i.e. in order to be able to evaluate the integrals appearing in (6.19), we have to dispose over the projections of the hat functions onto the triangular segments which make up their support. In virtue of the verbal definition of the hat functions given above, we can deduce these projections, which read

$$\begin{aligned} \psi^{(1)}(\xi,\eta) &= 1 - \xi - \eta \\ \psi^{(2)}(\xi,\eta) &= \xi \\ \psi^{(3)}(\xi,\eta) &= \eta \end{aligned}$$

when defined over the *reference triangle* $(0,0)$, $(1,0)$, $(0,1)$, see Figure 6.4 for an illustration. Assume now, that we are given an arbitrary triangle τ_k specified by its corners $\mathbf{p}_{\ell_1}$, $\mathbf{p}_{\ell_2}$ and $\mathbf{p}_{\ell_3}$. By virtue of the

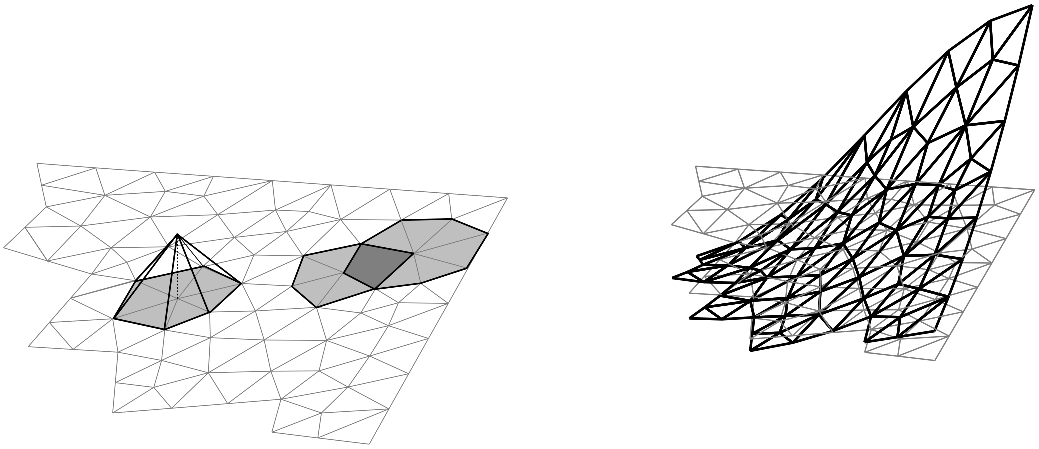


Figure 6.3: Mesh fragment. *(left)* Each of the hat functions ψ_i is centered on a mesh vertex, has unit height, decays linearly towards the neighboring vertices and, as a consequence, has an umbrella like support consisting of several adjacent triangular segments. Note, that some (actually a minor part) of the hat functions have overlapping supports. (*right*) By linearly combining the hat functions one can easily generate piecewise linear functions over the approximated domain T_h.

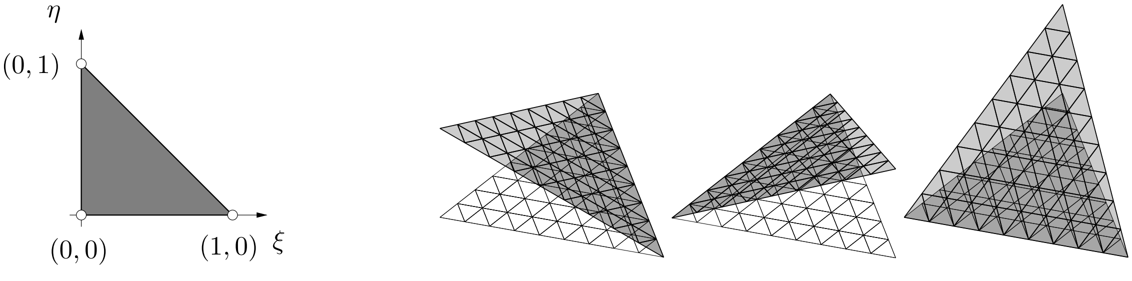


Figure 6.4: *(left)* Reference triangle in the (ξ, η) coordinate system *(right)* Projections of the hat functions $\psi^{(1)}$, $\psi^{(2)}$ and $\psi^{(3)}$ centered at the corners of the reference triangle.

mappings

$$\begin{pmatrix} x \\ y \end{pmatrix} = \mathbf{p}_{\ell_1} + \mathbf{M}_k \begin{pmatrix} \xi \\ \eta \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} \xi \\ \eta \end{pmatrix} = \mathbf{M}_k^{-1} \left[\begin{pmatrix} x \\ y \end{pmatrix} - \mathbf{p}_{\ell_1} \right], \tag{6.20}$$

where $\mathbf{M} = (\mathbf{p}_{\ell_2} - \mathbf{p}_{\ell_1}, \mathbf{p}_{\ell_3} - \mathbf{p}_{\ell_1})$, we can readily define the projections $\psi_{\ell_1}|_{\tau_k}$, $\psi_{\ell_2}|_{\tau_k}$ and $\psi_{\ell_3}|_{\tau_k}$ in the (x, y) coordinate system as

$$\begin{aligned} \psi_{\ell_1}(x,y)|_{\tau_k} &= \psi^{(1)}(\xi,\eta) = \psi^{(1)}(\mathbf{M}_k^{-1}[(x,y)^{\mathrm{T}} - \mathbf{p}_{\ell_1}]) \\ \psi_{\ell_2}(x,y)|_{\tau_k} &= \psi^{(2)}(\xi,\eta) = \psi^{(2)}(\mathbf{M}_k^{-1}[(x,y)^{\mathrm{T}} - \mathbf{p}_{\ell_1}]) \\ \psi_{\ell_3}(x,y)|_{\tau_k} &= \psi^{(3)}(\xi,\eta) = \psi^{(3)}(\mathbf{M}_k^{-1}[(x,y)^{\mathrm{T}} - \mathbf{p}_{\ell_1}]) \end{aligned}$$

By proceeding along the same lines, we can derive appropriate expressions for the gradients of the projections which read

$$\begin{aligned} \mathbf{grad}\psi_{\ell_1}(x,y)|_{\tau_k} &= \mathbf{M}_k^{-\mathrm{T}}\mathbf{grad}\psi^{(1)}(\mathbf{M}_k^{-1}[(x,y)^{\mathrm{T}} - \mathbf{p}_{\ell_1}]) \\ \mathbf{grad}\psi_{\ell_2}(x,y)|_{\tau_k} &= \mathbf{M}_k^{-\mathrm{T}}\mathbf{grad}\psi^{(2)}(\mathbf{M}_k^{-1}[(x,y)^{\mathrm{T}} - \mathbf{p}_{\ell_1}]) \\ \mathbf{grad}\psi_{\ell_3}(x,y)|_{\tau_k} &= \mathbf{M}_k^{-\mathrm{T}}\mathbf{grad}\psi^{(3)}(\mathbf{M}_k^{-1}[(x,y)^{\mathrm{T}} - \mathbf{p}_{\ell_1}]) \end{aligned}$$

With these ingredients at hand we can now evaluate the integrals on the right-hand side of (6.19) triangle by triangle and insert the so obtained entries in the appropriate places of the matrix $\mathbf{A}$. The coordinates of the entries are thereby given by the indices of the triangle vertices.

The remaining integrals appearing in (6.16) are handled in the same way. All domain integrals are thereby replaced by sums of integrals over triangles and evaluated in element wise fashion. Boundary integrals, such as $b_{\Gamma_N}(\cdot,\cdot)$ for example, undergo a similar treatment. In fact, being given a specification of the normal derivative $\partial_{\mathbf{n}}$ such as in (6.6), one can replace the term $\mathbf{grad}\phi^{\mathrm{T}}\mathbf{n}$ by the appropriate expression

$$b_{\Gamma_N}(v,\phi) = \int_{\Gamma_N} v\,\epsilon\,\mathbf{grad}\phi^{\mathrm{T}}\mathbf{n}\,d\Gamma_N = \int_{\Gamma_N} v\,\epsilon\,g\,d\Gamma_N \approx \sum_k \int_{\Gamma_N(\tau_k)} v\,\epsilon\,g\,d\Gamma_N(\tau_k), \tag{6.21}$$

which can again be evaluated triangle by triangle, i.e. boundary segment by boundary segment, respectively. Clearly, the test function v is finally replaced by the ψ_j's in order to construct the vector $\mathbf{b}$.

6.1.3 Assembling the Parts

In the previous subsections we have introduced all the ingredients that are necessary to compute the different quantities needed to come up with a discrete representation (6.17) of the Poisson problem (6.4, 6.5, 6.6). To reduce confusion, we will perform all of the above computations considering a concrete example.

Let τ be an arbitrary triangle specified by the coordinates and vertex indices

$$\tau : (2,3), (5,1), (6,4), \quad \ell_\tau : [14, 23, 9]. \tag{6.22}$$

The corresponding mapping matrix $\mathbf{M}_\tau$ can then be written as

$$\mathbf{M}_\tau = \begin{pmatrix} 5-2 & 6-2 \\ 1-3 & 4-3 \end{pmatrix} = \begin{pmatrix} 3 & 4 \\ -2 & 1 \end{pmatrix}. \tag{6.23}$$

Since all of the integrals involved contain problem dependent and hence arbitrary functions (like ϵ or ρ) we cannot expect to be able to express the arising element integrals in closed form. Therefore, it is customary to use a numerical quadrature formula (such as Gaussian quadrature [Sch93]) to evaluate the integrals. For particular expressions of the functions ψ_i, ϵ and ρ one can evaluate the integrals in a closed form.

Assuming that we are given abscissae (ξ_q, η_q) and weights ω_q for the Gaussian quadrature over the reference triangle, we can thus continue by first computing the corresponding points in the (x, y) coordinate systems, i.e.

$$\begin{pmatrix} x_q \\ y_q \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \end{pmatrix} + \mathbf{M}_\tau \begin{pmatrix} \xi_q \\ \eta_q \end{pmatrix}, \tag{6.24}$$

and by adapting the weights, i.e. $w_q = \mathsf{det}\,(\mathbf{M}_\tau)\omega_q = 11\omega_q$.

The rightmost integral (summands) of (6.19) can now be recast into the form

$$\begin{aligned}
\mathbf{A}^{(\tau)}_{r,s} &= \int_\tau \textbf{grad}\psi_{\ell_r}(x,y)^{\mathrm{T}}\,\epsilon(x,y)\,\textbf{grad}\psi_{\ell_s}(x,y)\,d\tau \\
&= \int_\tau \textbf{grad}\psi_{\ell_r}(x,y)^{\mathrm{T}}\,\epsilon(x,y)\,\textbf{grad}\psi_{\ell_s}(x,y)\,d\tau \\
&= \int_\tau \textbf{grad}\psi_{\ell_r}(x,y)|_\tau{}^{\mathrm{T}}\,\epsilon(x,y)\,\textbf{grad}\psi_{\ell_s}(x,y)|_\tau\,d\tau \\
&= \int_\tau \textbf{grad}\psi^{(r)}(x,y)^{\mathrm{T}}\mathbf{M}_\tau^{-1}\,\epsilon(x,y)\,\mathbf{M}_\tau^{-\mathrm{T}}\textbf{grad}\psi^{(s)}(x,y)\,d\tau \\
&= \sum_q w_q\,\textbf{grad}\psi^{(r)}(\xi_q,\eta_q)^{\mathrm{T}}\mathbf{M}_\tau^{-1}\,\epsilon(x_q,y_q)\,\mathbf{M}_\tau^{-\mathrm{T}}\textbf{grad}\psi^{(s)}(\xi_q,\eta_q)
\end{aligned}$$

where $r, s = 1, 2, 3$. Since the single entries of $\mathbf{A}^{(\tau)}$ are part of a global sum, we obtain the entire matrix $\mathbf{A}$ by adding up the quantities, i.e.

$$\mathbf{A}(\ell_\tau, \ell_\tau) = \mathbf{A}(\ell_\tau, \ell_\tau) + \mathbf{A}^{(\tau)} \tag{6.25}$$

borrowing MATLAB notation. With the remaining integrals we proceed in the same fashion. Note that the boundary integral is somewhat simpler to compute since it requires only a parametrisation of a single boundary segment and thus makes use of a one dimensional quadrature.

One further point to notice concerns the appropriate choice of ϕ_0. So far no restrictions have been imposed on ϕ_0 besides the one of satisfying the Dirichlet boundary conditions on Γ_D. Hence any choice (residing in an appropriate function space) would be fine. Hence, given that the chosen element functions ψ_i can be used to piecewise linearly interpolate between mesh vertices, it seems natural to set ϕ_0 to the linearly interpolated function $f(x, y)$ along Γ_D and zero elsewhere. In virtue of this choice, the vector $\mathbf{a}$ needs not be explicitly constructed, but can be easily obtained by linearly combining the corresponding columns of $\mathbf{A}$.

We conclude this subsection by stressing that once the Dirichlet boundary conditions have been incorporated into the model, one can eliminate the associated degrees of freedom (this way reducing the size of the problem).

6.2 A MATLAB implementation

So far, we have dealt with the analytical part of the Poisson problem, only. In order to demonstrate how simple it is to compute all of the required components (at least for this example), we will present a MATLAB implementation for the Poisson problem (6.4, 6.5, 6.6) with the geometry and the boundary conditions as shown in Figure 6.1. More precisely, we solve

$$\begin{aligned}
-\mathrm{div}\big(\epsilon(x,y)\textbf{grad}\phi(x,y)\big) &= 0 && (x,y) \in \Omega = (-1,1)^2 \\
\phi(x,y) &= \mathrm{sign}\,(x)U && (x,y) \in \{(\pm 1, y)\} \\
\partial_{\mathbf{n}}\phi(x,y) &= 0 && (x,y) \in \{(x, \pm 1)\}
\end{aligned}$$

where the permittivity of the ring $\epsilon_{\mathrm{ring}} = 100$, $\epsilon_0 = 1$, the radii correspond to $r = 0.6$ and $R = 0.8$ and the applied potential $U = 20$. Notice that we assume a charge free domain, i.e. $\rho = 0$.

We start with initialising some of the values that remain constant throughout the entire process.

```
% Problem/Material parameters
U     = 20;
R     = 0.8;
r     = 0.6;
Ering = 100;
Evac  = 1;
```

```
% Gaussian quadrature points and weights
XIETAQ = [ 7.503111022260811e-02     2.800199154990741e-01
           1.785587282636164e-01     6.663902460147014e-01
           2.800199154990741e-01     7.503111022260811e-02
           6.663902460147014e-01     1.785587282636164e-01 ]’;

OMEGAQ = [ 9.097930912801129e-02
           1.590206908719884e-01
           9.097930912801129e-02
           1.590206908719884e-01 ]’;
```

In the next step, we define the domain geometry. In MATLAB, assuming that the `pde` toolbox is installed, one does so by

```
% Parameters defining the geometry
gd = [ 3.0000    1.0000    1.0000
       4.0000    0.0000    0.0000
      -1.0000    0.0000    0.0000
       1.0000         R         r
       1.0000         0         0
      -1.0000         0         0
      -1.0000         0         0
      -1.0000         0         0
       1.0000         0         0
       1.0000         0         0 ];

ns = [ 83    67    67
       81    49    50
       49     0     0 ];

sf = ’SQ1+C1+C2’;

% Setting up the mesh
GEO     = decsg(gd, sf, ns);
[P,E,T] = initmesh(GEO);
[P,E,T] = refinemesh(GEO, P, E, T);
```

Once that the mesh has been generated we can iterate over all triangles and perform the computations described in section 6.1.3. Note that the storage for the matrix **A** is allocated beforehand by means of the `spalloc` command, since **A** is going to be a sparse matrix. The underlying matrix datastructure will be explained in Chapter 7.

```
% Setting up the system matrix and the right-hand side.
% Number of unknowns: n
n = size(P,2);
A = spalloc(n,n,10*n);
a = zeros(n,1);

% Gradients of the hat functions (grad phiˆ(i))
G = [ -1 1 0
      -1 0 1 ];

% Iterate over the triangles and construct the matrix
for tau=T
```

```
  % Retrieve vertex coordinates of the triangle tau
  V = P(:,tau(1:3));

  % Compute mapping matrix ind its inverse (and determinant)
  M  = [V(:,2)-V(:,1), V(:,3)-V(:,1)];
  iM = inv(M);
  dM = det(M);

  % Determine material parameters depending on the position of the
  % triangle (and its region)
  if (tau(4) == 3), Eact = Ering; else Eact = Evac; end

  % Evaluate the bilinear forms for the triangle tau
  Aloc = zeros(3);
  XYQ  = V(:,1)*ones(1,Q^2) + M*XIETAQ;
  WQ   = det(M)*OMEGAQ;
  for q=1:Q^2
    GPHI = iM’*G;
    Aloc = Aloc + WQ(q)*GPHI’*Eact*GPHI;
  end

  % Fill in the matrix entries
  A(tau(1:3), tau(1:3)) = A(tau(1:3), tau(1:3)) + Aloc;
end
```

Once that the (discrete) operators have been assembled, we proceed by identifying the indices of the unknown residing on the Dirichlet boundary.

```
% Determine indices of the hat functions placed on the
% Dirichlet boundary
DindL = find(abs(P(1,:) + 1.0) < 1e-10)’;
DindR = find(abs(P(1,:) - 1.0) < 1e-10)’;
Dind  = unique([DindL; DindR]);
rem   = setdiff(1:n, Dind)’;
```

The index vectors `DindL` and `DindR` contain the indices of the hat functions residing on the left and on the right sides of the square, respectively. Thus, `Dind` contains all indices of boundary hat functions. Note that `rem` has been computed for commodity and is the complementary (index) vector to `Dind`.

As mentioned at the end of Subsection 6.1.3, one can easily incorporate the Dirichlet boundary conditions by make use of the fact, that the hat functions allow for a simple piecewise interpolation over Ω. Hence, by simply letting all the hat functions which reside on Γ_D equal $\pm U$ (depending on which segment they reside) we are set.

```
% Compute Dirichlet contribution and take it to the right-hand side
phi0 = zeros(n,1);
phi0(DindL) = -U*ones(length(DindL),1);
phi0(DindR) =  U*ones(length(DindR),1);
a = a - A*phi0;
```

The matrix and the right-hand side are constructed. Since some of the boundary values are prescribed (the ones indexed by `Dind`), there is no need in keeping these degrees of freedom any longer. Thus, the only ones to be kept are the complementary ones, i.e. the ones indexed by `rem`.

```
% Reduce and solve the problem by throwing away the Dirichlet points
Ared = A(rem,rem);
ared = a(rem);
sred = Ared\ared;
```

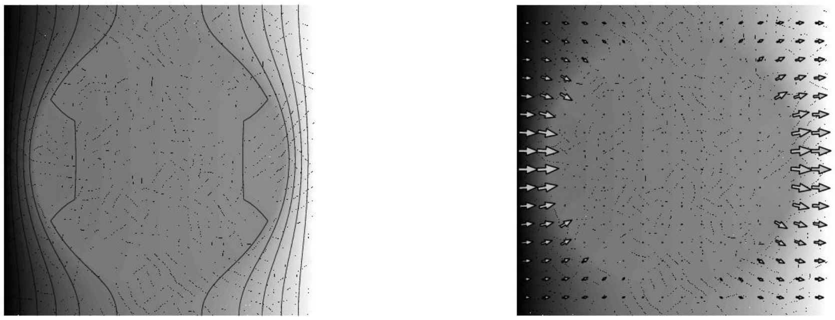


Figure 6.5: *(left)* Contour plot of the computed potential ϕ. *(right)* Field plot of the electric field $\mathbf{E} = \textbf{grad}\phi$. Note how the field vanishes in the interior of the ring.

We are done! For the sake of commodity we extend the solution and augment it by the Dirichlet points. This way, we can readily visualise the potential ϕ that we have computed, as shown in Figure 6.5.

```
% Blow up reduced problem such as to construct the entire solution.
% Corresponds to computing phi= phi0 + u
s       = phi0;
s(rem)  = sred;
```

6.3 The Solution — Assessing Correctness

Now that we have solved the problem, let us verify that the Faraday effect really kicks in. To this end, we compute the intensity of the electric field in the area enclosed by the ring which we define as

$$I^2_{\text{Faraday}} = \int_{x^2+y^2<r^2} \|\mathbf{E}(x,y)\|^2 \, dxdy = \int_{x^2+y^2<r^2} \|\textbf{grad}\phi(x,y)\|^2 \, dxdy. \tag{6.26}$$

By gradually refining the meshes used to approximate the domain Ω we should observe a reduction of this intensity, given that the approximate solution becomes increasingly accurate. In fact, by generating a sequence of refined meshes (using more than one `refinemesh` command), we can verify the predicted intensity drop, see Figure 6.6.

Another property that we want to verify concerns the convergence behavior of the FEM. To this end, we consider the Poisson problem

$$\begin{array}{rcll} -\partial_{xx}\phi(x,y) - \partial_{yy}\phi(x,y) = -\text{div}\big(\textbf{grad}\phi(x,y)\big) & = & -2(x^2-1) - 2(y^2-1) & (x,y) \in \Omega \\ \phi(x,y) & = & 0 & (x,y) \in \partial\Omega \end{array}$$

whose exact solution is

$$\phi^\star(x,y) = (x^2-1)(y^2-1). \tag{6.27}$$

By appropriately adapting our MATLAB code, i.e. by adding the computation of the right-hand side vector $\mathbf{l}$ and by modifying the handling of the Dirichlet boundary conditions, e.g.

```
[...]

% Iterate over the triangles and construct the matrix
for t=T
```

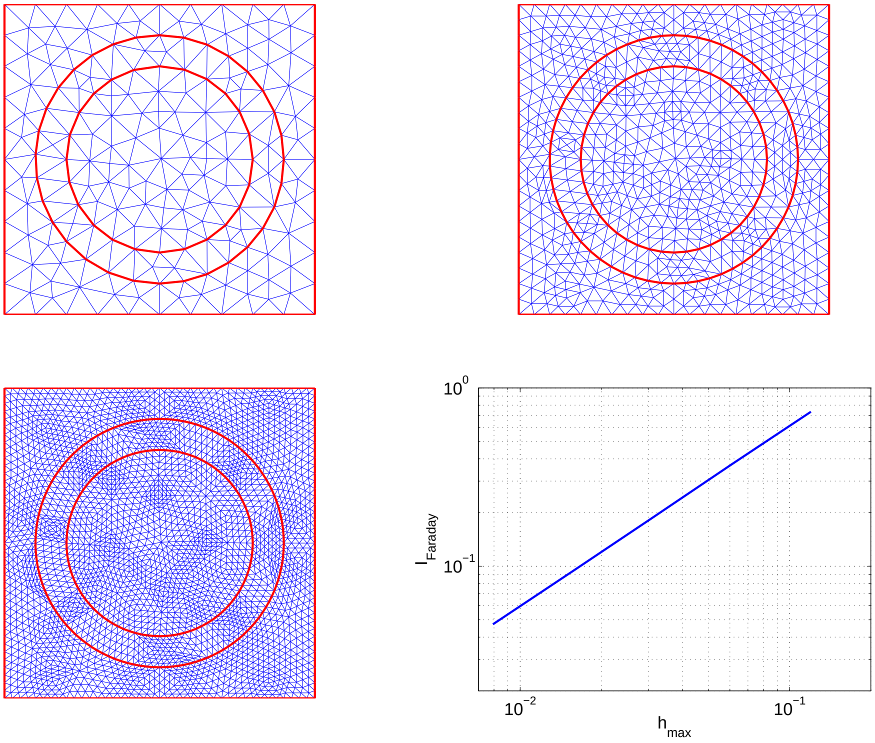


Figure 6.6: By refining the meshes, i.e. by reducing the meshwidth typically defined as the maximum of all element diameters $h_{\max}$, we can observe a reduction in the approximation error. Since we do not know a closed form expression for the solution, we gauge this reduction by considering the field intensity I_{Faraday}.

```
% Retrieve vertex coordinates of the triangle tau
V = P(:,t(1:3));

% Compute mapping matrix ind its inverse (and determinant)
M  = [V(:,2)-V(:,1), V(:,3)-V(:,1)];
iM = inv(M);
dM = det(M);

% Evaluate the bilinear forms for the triangle tau
Aloc = zeros(3);
lloc = zeros(3,1);
XYQ  = V(:,1)*ones(1,Q^2) + M*XIETAQ;
WQ   = det(M)*OMEGAQ;

for q=1:Q^2
  PHI  = [1-XIETAQ(1,q)-XIETAQ(2,q); XIETAQ(1,q); XIETAQ(2,q)];
  RHO  = -2*(XYQ(1,q)^2-1) - 2*(XYQ(2,q)^2-1);
  GPHI = iM'*G;

  Aloc = Aloc + WQ(q)*GPHI'*GPHI;
  lloc = lloc + WQ(q)*PHI*RHO;
end
```

```
  % Fill in the stuff
  A(t(1:3), t(1:3)) = A(t(1:3), t(1:3)) + Aloc;
  l(t(1:3))         = l(t(1:3))         + lloc;
end

% Determine indices of hat functions on Dirichlet boundary
DindL = find(abs(P(1,:) + 1.0) < 1e-10)’;
DindR = find(abs(P(1,:) - 1.0) < 1e-10)’;
DindT = find(abs(P(2,:) + 1.0) < 1e-10)’;
DindB = find(abs(P(2,:) - 1.0) < 1e-10)’;

Dind  = unique([DindT; DindB; DindL; DindR]);
rem   = setdiff(1:n, Dind)’;

% Reduce and solve the problem by throwing away the Dirichlet points
Ared = A(rem,rem);
lred = l(rem);
sred = Ared\lred;

% Blow up reduced problem such as to construct the entire solution
s = zeros(n,1);
s(rem)  = sred;

[...]
```

we obtain the desired FEM approximation $\phi(x, y)$ to $\phi^\star(x, y)$. Once computed, we can easily compute the so called L_2 error norm, i.e.

$$\|\phi(x,y) - \phi^\star(x,y)\|_2 = \left(\int_\Omega (\phi(x,y) - \phi^\star(x,y))^2 \, d\Omega \right)^{1/2} \tag{6.28}$$

which according to FEM theory, see [AB01], is element of $\mathcal{O}(h^2)$, given that we are using piecewise *linear* element functions. In fact, a quick look at Figure 6.7 suffices to verify this claim.

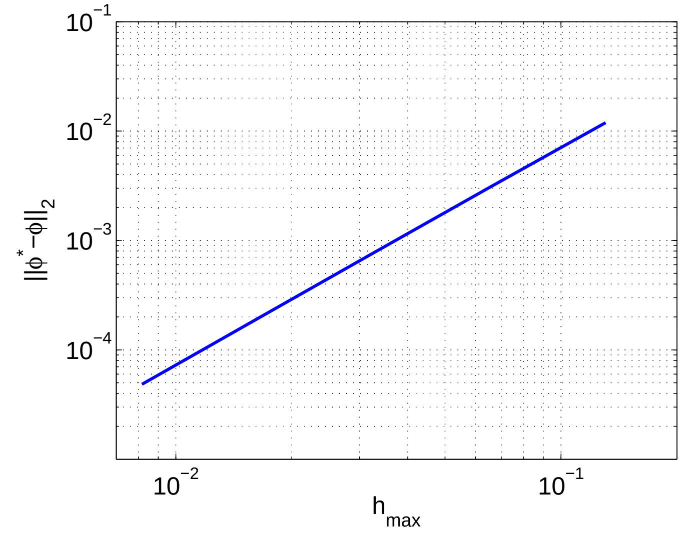


Figure 6.7: The predicted $\mathcal{O}(h^2)$ behavior of the L_2 error (6.28).

Bibliography

[AB01] O. Axelsson and V. A. Barker. *Finite Element Solution of Boundary Value Problems*. Classics in Applied Mathematics. SIAM, Philadelphia, 2nd edition, 2001.

[Bra97] D. Braess. *Finite Elemente*. Springer Berlin Heidelberg, 2nd edition, 1997.

[BS79] I. N. Bronštejn and K. A. Semendjaev. *Taschenbuch der Mathematik*. Harri Deutsch Verlag, Thun Frankfurt/Main, 24th edition, 1979.

[CH53] R. Courant and D. Hilbert. *Methods of Mathematical Physics*. Interscience Publishers, Inc., New York, 1953.

[CL91] P. G. Ciarlet and J. L. Lions. *Finite Element Methods (Part 1)*, volume 2nd of *Handbook of Numerical Analysis*. Elsevier Science Publishers B.V. (North-Holland), 1991.

[Jac99] J. D. Jackson. *Classical Electrodynamics*. Wiley, New York, 3rd edition, 1999.

[Sch91] H. R. Schwarz. *Methode der Finiten Elemente*. B. G. Teubner, Stuttgart, 3rd edition, 1991.

[Sch93] H. R. Schwarz. *Numerische Mathematik*. B. G. Teubner, Stuttgart, 3rd edition, 1993.

Chapter 7

Storage Schemes for Sparse Matrices

In the previous chapter we were concerned with the finite element approximation of physical models that can be described by partial differential equations. As shown in Section 6.3, these approximations require domain discretisations with a relatively small mesh width in order to yield accurate results. Unfortunately, using meshes with small mesh widths amounts to approximating the domain with a large number of elements which in turn leads to large and sparse matrices, say of order $n \gg 1$. In order to store such matrices one has to devise appropriate *storage schemes*, given the prohibitive amount of $\mathcal{O}(n^2)$ memory units required by the naive (dense) approach.

As we will see in Chapter 9, certain iterative solution schemes rely on the availability of the matrix vector product with the system matrix. Since these products represent one of the most time consuming ingredients in all these iterative schemes, it is of paramount importance to use data structures which allow for a swift product computation.

In order to better understand why the storage schemes that we are going to present in the following are appropriate, we start by considering some typical sparse matrices obtained from FEM discretisations, such as the one explained in Chapter 6. As shown in Figure 7.1, these matrices have relatively "few" *non-zero entries*, i.e. $\mathrm{nnz}(\mathbf{A}) \ll n^2$. In fact, the amount of non-zero entries is typically linear in the size n of the matrix $\mathbf{A}$, i.e. $\mathrm{nnz}(\mathbf{A}) \approx cn$, where the constant c can often be predicted depending on the element function type (FEM), the discretisation stencil (FDM), etc. It is thus not surprising, that all the popular data structures aim at storing non-zero entries only, exploiting symmetry whenever possible.

Before we start with an enumeration of the most prominent storage schemes taken from [Saa90], let us introduce some notation that will be useful for a proper specification of the structures. Given a sparse matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ containing $\mathrm{nnz}(\mathbf{A})$ non-zero entries, let

$$\mathcal{C}(i) = \{\, j \mid \mathbf{A}_{i,j} \neq 0\} \quad \text{and} \quad \mathcal{R}(j) = \{\, i \mid \mathbf{A}_{i,j} \neq 0\} \quad \text{for} \quad i,j = 1,\ldots,n \tag{7.1}$$

be ordered sets containing the column and the row indices of row i and column j, respectively. All the quantities set in `typewriter` style represent variables declared in a C/C++ routine. In particular, array counts (`i,j,k`) start from zero, whereas matrix indices (i,j,k) start from one. For the sake of simplicity we assume that `int`'s use one whereas `double`'s use two memory units, where one memory unit corresponds to 1 word (or 4 bytes). Finally, let us introduce the example matrix shown in Figure 7.2 which will be used to illustrate the storage schemes we are going to present.

7.1 Compressed Sparse Row (CSR), Compressed Sparse Column (CSC)

The *compressed sparse row* (CSR) scheme stores the matrix using the three arrays `row`, of length $n+1$, and `col` and `val` of length $\mathrm{nnz}(\mathbf{A})$, each. The array `row` implicitly contains the number of entries per row, i.e.

$$\texttt{row[i+1]} - \texttt{row[i]} = |\mathcal{C}(i+1)| \quad \text{where} \quad \texttt{row[0]} = 0 \tag{7.2}$$

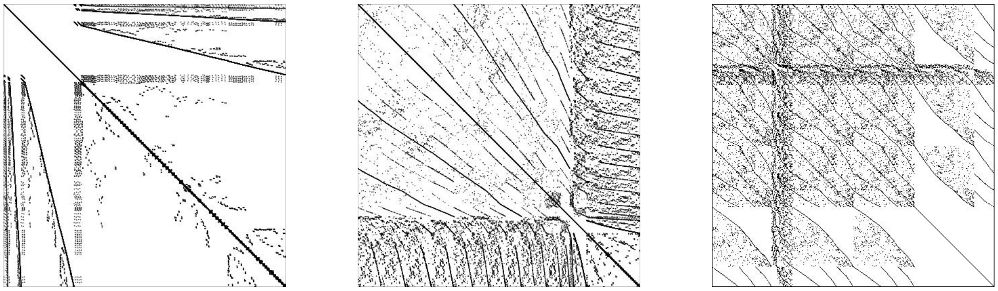


Figure 7.1: Typical sparsity patterns of matrices arising in (different) FEM discretisations. *(left)* Size: 11’583, nnz: 80’435 *(middle)* Size: 4’317, nnz: 58’802 *(right)* Size: 9’204, nnz: 311’218

	●	●		●	●	
●		●		●		
	●				●	
			●			
	●	●	●	●		
●		●				
	●		●			●

$$\begin{aligned}
\mathcal{C}(1) &= \{2,3,5,6\}\\
\mathcal{C}(2) &= \{1,3,5\}\\
\mathcal{C}(3) &= \{2,6\}\\
\mathcal{C}(4) &= \{4\}\\
\mathcal{C}(5) &= \{2,3,4,5\}\\
\mathcal{C}(6) &= \{1,3\}\\
\mathcal{C}(7) &= \{2,4,7\}
\end{aligned}$$

Figure 7.2: An example of a non-symmetric sparse 7×7 matrix with corresponding column sets $\mathcal{C}(i)$.

and hence `row[n]` $= \mathrm{nnz}(\mathbf{A})$. These indices are then used as start and end points for the column indices (of row i) stored in `col` and the corresponding entries stored in `val`. To iterate over all the entries we then write

```
for(int i=0; i<n; i++){
  for(int k=row[i]; k<row[i+1]; k++){
    int     j = col[k];     // j = C(i+1)[k-row[i]+1] - 1
    double  v = val[k];     // v = A(i+1,j+1)
  }
}
```

This scheme requires $3\,\mathrm{nnz}(\mathbf{A}) + (n+1)$ memory units. Note that if the matrix is symmetric, it may be convenient to store only the lower/upper triangle.

Example: The matrix shown in Figure 7.2 would lead to the following three arrays

```
row[] = {0,4,7,9,10,14,16,19};
col[] = {1,2,4,5,0,2,4,1,5,3,1,2,3,4,0,2,1,3,6};
val[] = {...};
```

where the entries in `val` correspond to the real values of the entries.

If instead of storing the matrix in a row-wise fashion one would prefer to do so in a column-wise way, the roles of $\mathcal{C}$ and $\mathcal{R}$ (i.e. the roles of `row` and `col`) could simply be interchanged, leading to a *compressed sparse column* (CSC) storage scheme. Note that the CSC scheme is used by MATLAB’s sparse functions toolbox (`sparfun`), containing commands like `spalloc`, `spdiags`, `spconvert`, etc.

As mentioned in the introduction, it is essential that the chosen data structure allows for a fast matrix-vector product, considering its importance in all (matrix-free) iterative solvers. Here, fast means that the amount of work should be proportional to $\mathrm{nnz}(\mathbf{A})$ and that the amount of index arithmetic should be kept low. For a matrix stored in CSR scheme, the computation of the product

$$\mathbf{y} := \mathbf{A}\mathbf{x} \tag{7.3}$$

could be implemented in C++ the following way.

```
double  y[n];

for(int i=0; i<n; i++){
  double  s=0.0;
  for(int k=row[i]; k<row[i+1]; k++){
    int&       j=col[k];
    double&  Aij=val[k];

    s += Aij*x[j];
  }
  y[i] = s;
}
```

7.2 Modified Sparse Row (MSR), Modified Sparse Column (MSC)

In some circumstances it is preferable to store the main diagonal of the matrix $\mathbf{A}$ separately, in order to grant quick access to its elements. Once that these elements have been “treated” the remaining matrix is stored in a CSR similar format by using only two arrays, `rowcol` and `val`, justifying the name *modified compressed sparse row* (MSR).

In fact, the first $n + 1$ entries of `rowcol` correspond to the `row` array of the CSR scheme (taking into account the removal of the diagonal entries), whereas the first $n + 1$ entries of `val` contain the diagonal elements (plus an additional blank entry). The followup positions of `rowcol` will then contain the column indices of the off-diagonal elements, i.e. `col` in the CSR scheme (taking into account the removal of the diagonal entries), whereas `val` will contain the corresponding matrix entries. Hence, the arrays have (both) a length of

$$m = (n + 1) + (\mathrm{nnz}(\mathbf{A}) - \sum_{i=1}^{n}\{\mathbf{A}_{i,i} \neq 0\}) \tag{7.4}$$

leading to a total of $3m$ memory units.

Example: The matrix shown in Figure 7.2 would lead to the following three arrays

```
  rowcol[] = {0,4,7,9,9,12,14,16,
               1,2,4,5,0,2,4,1,5,1,2,3,0,2,1,3};
  val[]    = {x,x,x,x,x,x,x,0, ... };
```

where the entries in `val` correspond to the real values of the entries. Note, that the blank/linebreak separates the diagonal part from the off-diagonal one.

If instead of storing the matrix in a row-wise fashion one would do so in a column-wise way, this would lead to the *modified compressed sparse column* (MSC) storage scheme.

7.3 Coordinate Format (AIJ)

The *coordinate format* is doubtlessly the simplest storage scheme for sparse matrices. It consists of three arrays `row`, `col` and `val` each of length $\mathrm{nnz}(\mathbf{A})$, leading to an overall memory consumption of $4\,\mathrm{nnz}(\mathbf{A})$

memory units. As suggested by the name, each matrix element can be represented by a triple $(i, j, \mathbf{A}_{i,j})$ whose entries are stored in the aforementioned arrays, respectively.

Example: The matrix shown in Figure 7.2 would lead to the following three arrays (or any valid permutation of the latter)

```
row[] = {0,0,0,0,1,1,1,2,2,3,4,4,4,4,5,5,6,6,6};
col[] = {1,2,4,5,0,2,4,1,5,3,1,2,3,4,0,2,1,3,6};
val[] = {...};
```

where the entries in `val` correspond to the real values of the entries.

Although this matrix format can not compete with the CSR(C) and the MSR(C) schemes from the memory consumption point of view, it may be more convenient when it comes to matrix assembly. Whenever a new matrix element needs to be inserted it can just be appended (if no precise ordering is imposed), whereas in the compressed storage schemes possibly large memory blocks need to be shifted. However, checking if an element is already set is not cheap (unless an ordering is imposed).

7.4 Linked List Format

All the schemes presented so far allow for a fast traversal but suffer from major shortcomings when it comes to sparse matrix assembly. A format which strikes a good balance between both requirements is the *linked list* storage scheme (though its memory requirements are considerable).

It consists of the arrays `row`, of length n, and `col`, `val` and `lnk` of length $\operatorname{nnz}(\mathbf{A})$, each, making up for a total of $4\operatorname{nnz}(\mathbf{A}) + n$ memory units. Similar to the CSR format, the array `row` contains the entry points into the other arrays, i.e. `row[k]` indexes the first entry in row $k + 1$, whose column and value are stored in `col[row[k]]` and `val[row[k]]`. The indices of the next elements are stored in `lnk[row[k]]`, `lnk[lnk[row[k]]]`, etc. this way building a link of indices which can be followed to obtain the next entries. As soon as the link points to -1, the end of the row is reached.

Example: The matrix shown in Figure 7.2 would lead to the four arrays (or an appropriate permutation/adaption)

```
row[] = {0,4,7,9,10,14,16,19};
lnk[] = {1,2,3,-1,5,6,-1,8,-1,-1,11,12,13,-1,15,-1,17,18,-1}
col[] = {1,2,4,5,0,2,4,1,5,3,1,2,3,4,0,2,1,3,6};
val[] = {...};
```

where the entries in `val` correspond to the real values of the entries.

This storage format allows for a (rather) fast traversal *and* a fast detection/insertion/deletion of matrix elements, in particular if the linked lists for each row are sorted. This can be achieved at a modest price, given the relatively low number of entries per row for typical matrices. It goes without saying, that the same scheme could also be set up in a column-wise fashion.

7.5 $\mathscr{H}$–Matrices*

The sparsity pattern exhibited by matrices that are assembled following the FEM paradigm reflects the arrangement of the basis functions w.r.t. their support intersections. The main idea behind $\mathscr{H}$-Matrices, see [Hac99], consists in using this sparsity pattern dependency, which in combination with the geometrical information of the mesh can be used to find appropriate *function clusters*. Functions whose supports are located closely to each other are thereby said to lie in their mutual *near-field*, whereas functions which are located too far apart barely exhibit any interaction and are said to lie in their mutual *far-field*. This property is taken advantage of, by first grouping functions which lie in their mutual near field into *clusters*. The interaction of the latter is then very efficiently expressed by means of *hierarchical structures*, typically *trees*. A very detailed analysis and discussion of the involved data structures and algorithms can be found in [Hac99], [GH03] and [BGH05].

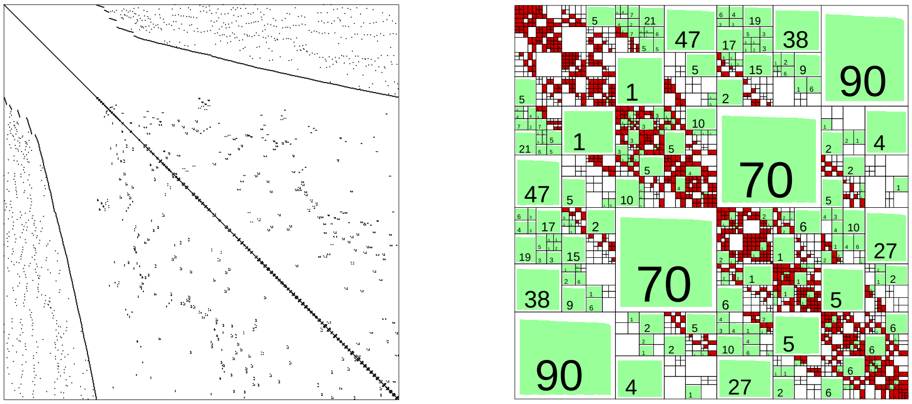


Figure 7.3: *(left)* Sparsity pattern of a matrix obtained by discretising a scalar Helmholtz problem by means of the FEM. No reordering has been applied to the matrix. *(right)* Visualisation of the data structure used to store the input matrix. The entries have been clustered according to their Euclidean distance yielding numerous small dense blocks. The so obtained clusters are then reordered in such a way as to have their near-field neighbour-clusters in possibly adjacent index positions. The remaining far-field clusters are summed up into low-rank block approximations. The figures in the panels represent the rank of the corresponding block.

The reordering of the discretisation arrangement induced by the tree structure is reflected also in the sparsity pattern. Functions belonging to the same cluster obtain similar indices. Hence, the arising matrix will exhibit dense blocks mostly along the diagonal and sparser ones in off-diagonal regions, mainly depending on the *cluster size* that is specified[1]. In order to save storage, dense blocks are stored explicitly, whereas sparse blocks are approximated by *low rank matrices* whose accuracy and maximum rank can be specified by the user. The "sparsity pattern" corresponding to a typical *cluster tree structure* is shown in Figure 7.3.

[1] $\mathscr{H}$-matrices are typically used to *approximately* store matrices. This is particularly usefull in cases in which (approximate) factorisations need to be computed. If an exact representation is needed, it is not advisable to use this storage scheme

Bibliography

[BGH05] S. Börm, L. Grasedyck, and W. Hackbusch. *Hierarchical Matrices*. Max–Planck-Institut für Mathematik in den Naturwissenschaften, revised edition, 2005. Lecture notes, `http://www.mis.mpg.de/scicomp/Fulltext/WS_HMatrices.pdf`.

[GH03] L. Grasedyck and W. Hackbusch. Construction and arithmetics of $\mathcal{H}$-matrices. *Computing*, 70:295–334, 2003.

[Hac99] W. Hackbusch. A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: Introduction to $\mathcal{H}$-matrices. *Computing*, 62:89–108, 1999.

[Saa90] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.

Chapter 8

Reorderings and Sparse Direct Solvers

Solving discretised problems, such as the ones described in Chapter 6, often corresponds to solving linear systems of equations. As mentioned in the previous chapter, the system matrices are typically (very) large and *direct solution methods*, i.e. methods that rely on factorisations, can no longer be used without the application of sophisticated preprocessing steps, which aim at avoiding a prohibitive growth in memory consumption (and hence in computation time). One of these preprocessing phases controls the matrix structure and alters it by *reordering* the entries according to the needs.

8.1 Fill-in Reducing (and other) Reorderings

As mentioned in the introduction, factorising sparse matrices without taking appropriate precautions might lead to factors whose memory consumption is (extremely) large. To demonstrate this, we compute the Gaussian factorisation of the leftmost matrix shown in Figure 7.1. Due to the way Gaussian elimination[1] works, the bottommost/rightmost rows/columns of the factors will start to become more and more dense, causing a so called *fill-in* of the matrix. Since the original structure is somewhat arbitrary, we can not expect the fill-in to be bounded (and low) and hence a lot of additional memory will be used, as shown in Figure 8.1. Fortunately, by applying an appropriate reordering (permutation) $\boldsymbol{\pi}$ to the rows and columns, we can drastically reduce the amount of fill-in. But how can we obtain a good permutation $\boldsymbol{\pi}$?

Consider the sparsity structure of a matrix and think of it as adjacency matrix of a (undirected) graph $\mathscr{G}^{(0)}(\mathbf{A})$. Then cut the graph into two subgraphs $\mathscr{G}_1^{(1)}(\mathbf{A})$ and $\mathscr{G}_2^{(1)}(\mathbf{A})$ of (almost) the same size in such a way that their common "interface" is (almost) minimal, i.e. in such a way that the amount of edges that are cut is minimal. After having computed this *minimal Graph bisector*, reorder the matrix such that the first indices are the vertex indices of $\mathscr{G}_1^{(1)}(\mathbf{A})$, followed by the vertex indices of $\mathscr{G}_2^{(1)}(\mathbf{A})$ followed by the vertex indices in the "cut" $\mathscr{G}_1^{(1)}(\mathbf{A}) \cap \mathscr{G}_2^{(1)}(\mathbf{A})$. What we obtain is a blocked sparse matrix exhibiting a sparsity pattern as shown in 8.2. Clearly, applying a Gaussian elimination to this new matrix will generate fill-in "only" in the single blocks, but not in the $(1,2)$/$(2,1)$ blocks, respectively. By recursively applying this idea to the newly obtained $(1,1)$ and $(2,2)$ blocks we end up with the so called *nested dissection reordering*, also shown in Figure 8.2.

The nested dissection scheme presented above has been designed for sparse symmetric matrices onto which a Gaussian (or similar) factorisation needs to be applied. Depending on the particular sparsity structure, however, one might prefer to use the *minimum degree algorithm* — a greedy algorithm which reduces fill-in in the very next elimination step.

If the matrix at hand is non-symmetric or if the reordered structure has to satisfy properties which are favourable for other applications, other kinds of permutations need to be computed. Among these, one finds the *(reverse) Cuthill–McKee algorithm* [CM69] and the *Gibbs–Poole–Stockmeyer algorithm* [GaPKS76], both aiming at reducing the matrix' bandwidth, the *(symmetric) maximum weight matching algorithm* [DG02]

[1] Notice that the following is also usefull when computing a Cholesky or a $\mathrm{LDL}^{\mathrm{T}}$ factorisation.

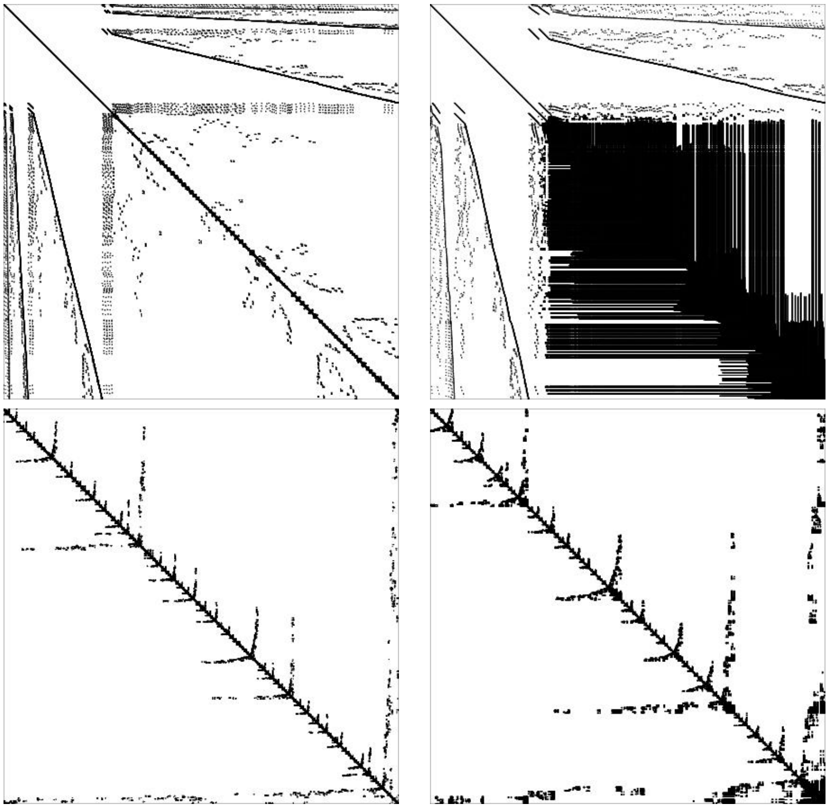


Figure 8.1: Influence of a reordering on the sparsity structure of Gaussian factors. *(first)* Original matrix $\mathbf{A}$ $[n : 11'583, \mathrm{nnz}(\mathbf{A}) : 80'435]$ *(second)* Sum $\mathbf{L}+\mathbf{U}$ of the Gaussian factors $\mathbf{A} = \mathbf{LU}$ $[\mathrm{nnz}(\mathbf{L}+\mathbf{U}) : 10'206'985]$ *(third)* Reordered matrix $\mathbf{A}(\boldsymbol{\pi},\boldsymbol{\pi})$ *(fourth)* Sum $\mathbf{L}_{\boldsymbol{\pi}}+\mathbf{U}_{\boldsymbol{\pi}}$ of the Gaussian factors $\mathbf{A}(\boldsymbol{\pi},\boldsymbol{\pi}) = \mathbf{L}_{\boldsymbol{\pi}}\mathbf{U}_{\boldsymbol{\pi}}$ $[\mathrm{nnz}(\mathbf{L}_{\boldsymbol{\pi}}+\mathbf{U}_{\boldsymbol{\pi}}) : 689'817]$

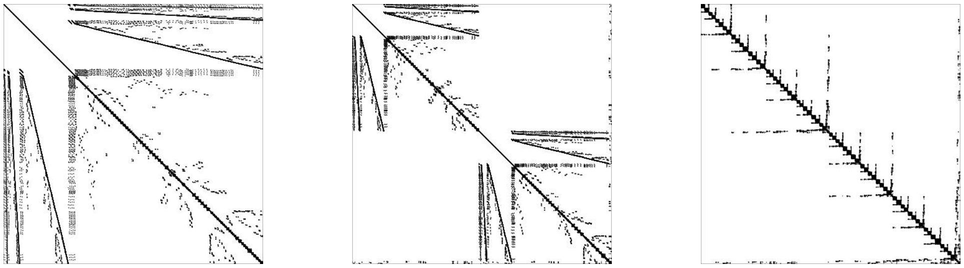


Figure 8.2: *(left)* Original sparse matrix *(middle)* Matrix obtained after one graph bisection step *(right)* Matrix obtained after performing the whole nested dissection loop.

which pushes the large entries towards the diagonal and ensures desirable pivots, etc. Some additional comments on these reorderings can be found in the following section.

8.2 Direct Solvers

Let us return to our original task, i.e. the one of computing a Gauss like factorisation of the system matrix. Following [DDSvdV98, Chapter 6], a direct solution algorithm for sparse matrices can be divided in the following phases:

(P0) **Preordering** This phase determines a reordering of the original matrix such that fill-in is reduced in the $\mathbf{L}$ and $\mathbf{U}$ factors. This phase is difficult to parallelize and is typically computed redundantly on a parallel machine, or on a single processor with results then broadcast to other processors.

(P1) **Analysis** Here the factorization is computed using the structure only, producing the patterns of $\mathbf{L}$ and $\mathbf{U}$. In particular, many solvers use this phase to identify dense supernodes that will improve cache and register performance; and also determine elimination trees for additional parallelism during the numerical phases (P2) and (P3).

(P2) **Numerical factorization** In this phase, the actual values of $\mathbf{L}$ and $\mathbf{U}$ are computed. In most cases this phase is by far the most expensive in terms of serial operation count.

(P3) **Forward/Backward substitution** This phase finds a solution $\mathbf{x}$ given a right-hand side $\mathbf{b}$. Compared to the factorization, the serial cost of this a solve phase is low, perhaps by a factor of 10–100 or more.

Some codes combine phases (P1) and (P2). Phase (P0) is typical of sparse matrices: since $\mathbf{A}$ is sparse, $\mathbf{L}$ and $\mathbf{U}$ are still sparse, though they may have some fill-in, i.e. non-zero entries which are zero in $\mathbf{A}$. Therefore, the factorization is performed on $\mathbf{PAQ}$. In a typical solver for an nonsymmetric matrix, the column permutation, $\mathbf{Q}$ is chosen to minimize fill-in, while $\mathbf{P}$, the row permutation, is chosen to maintain numerical stability. Many ordering methods exist to reduce fill-in, for example multiple variations on minimum degree orderings and graph partitioning algorithms. Solvers designed for symmetric and nearly symmetric matrices typically use symmetric permutations to maintain symmetry. No single ordering method is best for all matrices, nor has a heuristic been found that consistently chooses the best ordering [BFM03, ADD04].

Since pivoting adds complexity, which significantly increases execution time, many solvers offer options to reduce the cost of pivoting [LD98, SG04b]. Some equilibrate the rows and columns of the matrix to improve diagonal dominance. Many codes will consider the effect on execution time when choosing a pivot, accepting some loss of numerical stability [Mal91]. Typically this just means accepting the diagonal pivot if it is within some threshold of the best available, but some codes will take the predicted effect on execution time into consideration when choosing an off diagonal pivot [Dav04].

It is possible to combine similar rows and columns into blocks to improve locality and allow high performing BLAS to be called; this combination can consider only rows and columns that are identical, or accept minor differences in the rows and columns that they treat as blocks. Such blocking can reduce the cost of symbolic factorization as well [ADLL01].

We have just mentioned few reasons that explained why the performance of a sparse direct solver depends on the underlying matrix, computer, application, algorithms, and libraries as well as the code and how it is compiled. Given the differences in matrix, computer, application, algorithms, and libraries, it is unlikely that a single sparse direct solver will outperform all others across all usages.

8.3 Available Software

We conclude this Chapter with a brief overview of freely available sparse direct solvers.

- **UMFPACK** is a C package copyrighted by Timothy A. Davis. More information can be obtained at the web page `http://www.cise.ufl.edu/research/sparse/umfpack`.

- **TAUCS**, authored by S. Toledo, is a serial Cholesky solver [RT04b, RT04a, IST04] which can be obtained at `http://http://www.tau.ac.il/˜stoledo/taucs/`

- The **SuperLU** suite of solvers, written by Xiaoye S. Li, are written in ANSI C. It is copyrighted by The Regents of the University of California, through Lawrence Berkeley National Laboratory. We refer to the web site `http://www.nersc.gov/˜xiaoye/SuperLU` and to the SuperLU manual [DGL03] for more information.

Other solvers are available for parallel environments. Parallel aspects will be discussed in Chapter 13.

- **PARDISO** is package to solve large sparse symmetric and non-symmetric linear systems on shared memory multi-processors, developed at the Computer Science Department of the University of Basel. A discussion of the algorithms used in PARDISO and more information on the solver can be found at `http://www.computational.unibas.ch/cs/scicomp` and in documents [SG04a, SG04b].

- **MUMPS** ("MUltifrontal Massively Parallel Solver") is a parallel direct solver, written in FORTRAN 90 with a C interface, copyrighted by P. R. Amestoy, I. S. Duff, J. Koster, J.-Y. L'Excellent. Up-to-date copies of the MUMPS package can be obtained from the Web page `http://www.enseeiht.fr/apo/MUMPS/`. Reordering techniques can take advantage of PORD (distributed within MUMPS), or METIS. For details about the algorithms and the implementation, as well as of the input parameters, we refer to [ADLK03]

- **DSCPACK**, written by Padma Raghavan, is a domain-separator code for the parallel solution of sparse linear system. DSCPACK provides a variety of sparsity preserving (fill-reducing) ordering and computes either an LL^T (Cholesky) or LDL^T factorization of the linear system matrix. This solver is written in C, and it uses MPI for inter-processor communication, and the BLAS library for improved chace-performances. The implementation is based on the idea of partitioning the sparse matrix into domains and separators. More details can be found on the web site `http://www.cse.psu.edu/˜ragavan/dscpack` and on the DSCPACK manual [Rag02].

Bibliography

[ADD04] P. R. Amestoy, T. A. Davis, and I. S. Duff. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software*, 30(3):381–388, 2004.

[ADLK03] P.R. Amestoy, I.S. Duff, J.-Y. L'Excellent, and J. Koster. *MUltifrontal Massively Parallel Solver (MUMPS Versions 4.3.1) Users' Guide*. CERFACS, Toulouse, France, 2003.

[ADLL01] P. R. Amestoy, I. S. Duff, J.-Y. L'excellent, and X. S. Li. Analysis and comparison of two general sparse solvers for distributed memory computers. *ACM Trans. Math. Softw.*, 27(4):388–421, 2001.

[BFM03] M. Baumann, P. Fleischmann, and O. Mutzbauer. Double ordering and fill-in for the LU factorization. *SIAM J. Matrix Anal. Appl.*, 25(3):630–641, 2003.

[CM69] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. 24th Nat. Conf. ACM*, pages 157–172, 1969.

[Dav04] Timothy A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):165–195, 2004.

[DDSvdV98] Jack J Dongarra, Iain S Duff, Danny C Sorensen, and Henk van der Vorst. *Numerical Linear Algebra for High-Performance Computing*. SIAM, Philadelphia, PA, 1998.

[DG02] I. S. Duff and J. R. Gilbert. Maximum-weighted matching and block pivoting for symmetric indenite systems. In *Abstract book of Householder Symposium XV*, pages 73–75, 2002.

[DGL03] J. W. Demmel, J. R. Gilbert, and X. S. Li. *SuperLU Users' Guide*. Computer Science Division, University of California, Berkely, 2003.

[GaPKS76] N. E. Gibbs and W. G. Poole Jr. and P. K. Stockmeyer. An algorithm for reducing the bandwidht and profile of a sparse matrix. *SIAM J. Num. Anal.*, 13:236–249, 1976.

[IST04] D. Irony, G. Shklarski, and S. Toledo. Parallel and fully recursive multifrontal supernodal sparse cholesky. *Future Generation Computer Systems*, 20(3):425–440, April 2004.

[LD98] Xiaoye S. Li and James W. Demmel. Making sparse gaussian elimination scalable by static pivoting. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–17, Washington, DC, USA, 1998. IEEE Computer Society.

[Mal91] Joel Malard. Threshold pivoting for dense lu factorization on distributed memory multiprocessors. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 600–607, New York, NY, USA, 1991. ACM Press.

[Rag02] P. Raghavan. Domain-separator codes for the parallel solution of sparse linear systems. Technical Report CSE-02-004, Department of Computer Science and Engineering, The Pennsylvania State University, 2002.

[RT04a] V. Rotkin and S. Toledo. The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Transactions on Mathematical Software*, 30:19–46, 2004.

[RT04b] E. Rozin and S. Toledo. Locality of reference in sparse Cholesky methods. To appear in Electronic Transactions on Numerical Analysis, August 2004.

[SG04a] O. Schenk and K. Gärtner. On fast factorization pivoting methods for sparse symmetric indefinite systems. Technical Report, Department of Computer Science, University of Basel, 2004. Submitted.

[SG04b] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems*, 20(3):475–487, 2004.

Chapter 9

Iterative Solvers

For the solution of the linear system, one can use either direct or iterative solvers. The former methods, described in Chapter 8, are usually a better choice for small-size problems, since they are much less sensitive with respect to the conditioning of the problem. As such, direct solvers are typically very reliable; however, they are prohibitively expensive for large-size problems. The substantially sequential algorithm makes efficient parallel implementations of direct solvers on modern parallel computers troublesome. Furthermore direct solvers require a lot of storage to perform the factorisation. As the dimension of the linear system increases, iterative solvers are very often the matter of choice.

Other reasons may suggest the use of iterative solvers. In an iterative solver, the system matrix A is not needed in explicit form but only a procedure to compute the matrix-vector product $A\mathbf{x}$ for a given vector $\mathbf{x}$ is required. Furthermore, prior knowledge about the solution can be introduced into the solution process via the initial approximation. Finally, iterative methods can be terminated when the desired accuracy is achieved. In fact, the linear system under consideration is the discrete version of a system with an infinite number of degrees of freedom (which itself is only a model of the real system). It does not make any sense to compute the solution of the linear system with accuracy higher than the discretisation error. Hence, they require only the employment of those resources (time, memory) which are necessary to obtain the required accuracy.

Unfortunately, the performance of iterative solvers depends strongly on the spectral properties of the linear system matrix, and in particular its condition number. If the matrix is ill-conditioned, one is forced to devise an efficient preconditioner. A preconditioner is an operator that transforms the initial linear system into another one having the same solution, but better conditioned and hence easier to solve. Preconditioners are covered in Chapter 10.

9.1 Fixed Point and Jacobi Iteration

Pure mathematicians often write nonlinear equations in fixed point form as $\mathbf{x} = \mathbf{\Phi}(\mathbf{x})$, and solve them under the assumption that $\mathbf{\Phi}$ is a contraction by the so-called **Picard iteration** or **fixed point iteration** [*Fixpunkt-Iteration*] $\mathbf{x}_{n+1} := \mathbf{\Phi}(\mathbf{x}_n)$. This idea is also applicable to a linear system $\mathbf{Ax} = \mathbf{b}$: we rewrite it in fixed point form as

$$\boxed{\mathbf{x} = \mathbf{Bx} + \mathbf{b}} \qquad \text{with} \qquad \boxed{\mathbf{B} :\equiv \mathbf{I} - \mathbf{A}} \tag{9.1}$$

and apply fixed point iteration: starting with some $\mathbf{x}_0$, we compute for $n = 0, 1, 2, \ldots$

$$\boxed{\mathbf{x}_{n+1} := \mathbf{Bx}_n + \mathbf{b}\,.} \tag{9.2}$$

Here, $\mathbf{x}_n$ is the nth approximation of the fixed point $\mathbf{x}_\star$ satisfying $\mathbf{x}_\star = \mathbf{Bx}_\star + \mathbf{b}$, *i.e.,* $\mathbf{Ax}_\star = \mathbf{b}$; it is also called nth **iterate** [*Iterierte*].

If $a_{k,k} = 1$ $(\forall k)$, *i.e.,* $b_{k,k} = 0$ $(\forall k)$, then iteration (9.2) is called **Jacobi iteration** [*Gesamtschritt-Verfahren*]. Note that $a_{k,k} = 1$ can be achieved whenever $a_{k,k} \neq 0$ by scaling the kth equation.

In the linear case, $\mathbf{\Phi}: \ \mathbf{x} \mapsto \mathbf{Bx}$ is called a **contraction** [*Kontraktion*] if $\|\mathbf{B}\| < 1$ in some norm. Then the linear fixed point iteration converges globally, that is, for every $\mathbf{x}_0$. The proof is left as an easy exercise. But we can establish a sharper result because the underlying space is finite dimensional:

THEOREM 9.1.1. *For the* (*linear*) *fixed point iteration* (9.2) *holds*

$$\boxed{\mathbf{x}_n \longrightarrow \mathbf{x}_\star \ \textit{ for any } \mathbf{x}_0 \quad \Longleftrightarrow \quad \rho(\mathbf{B}) < 1\,,} \tag{9.3}$$

where $\rho(\mathbf{B}) :\equiv \max\{|\lambda| \bigm| \lambda$ *eigenvalue of* $\mathbf{B}\}$ *is the* ***spectral radius*** *of* $\mathbf{B}$.

PROOF. Let $\mathbf{BV} = \mathbf{V\Lambda}$ be an **eigenvalue decomposition** [*Eigenwert-Zerlegung*], so that $\mathbf{\Lambda}$ is diagonal or, in general, a **Jordan canonical form** [*Jordansche Normalform*] of $\mathbf{B}$. Clearly,

$$\mathbf{x}_n - \mathbf{x}_\star = \mathbf{B}(\mathbf{x}_{n-1} - \mathbf{x}_\star) = \mathbf{B}^n(\mathbf{x}_0 - \mathbf{x}_\star)\,,$$

so convergence occurs if and only if $\mathbf{B}^n \longrightarrow \mathbf{O}$ as $n \longrightarrow \infty$. This, in turn, happens if and only if $\mathbf{\Lambda}^n \longrightarrow \mathbf{O}$ as $n \longrightarrow \infty$.

If $\mathbf{\Lambda}$ is diagonal, the equivalence (9.3) clearly follows. But in general, $\mathbf{\Lambda}$ is block diagonal and may contain bidiagonal **Jordan blocks** [*Jordansche Blöcke*] like

$$\mathbf{J}_k :\equiv \begin{pmatrix} \lambda_k & 1 & 0 & \cdots & 0 \\ 0 & \lambda_k & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \lambda_k & 1 \\ 0 & \cdots & \cdots & 0 & \lambda_k \end{pmatrix}. \tag{9.4}$$

Since $\mathbf{\Lambda}^n$ is block diagonal too, it remains to show that $\mathbf{J}_k^n \longrightarrow \mathbf{O}$ as $n \longrightarrow \infty$. Assume $\mathbf{J}_k$ is of size $m_k \times m_k$. We write it as $\mathbf{J}_k = \lambda_k \mathbf{I} + \mathbf{S}$ and note that $\mathbf{S}^\ell$ is zero except for ones on the ℓth upper codiagonal. In particular, $\mathbf{S}^\ell = \mathbf{O}$ when $\ell \geq m_k$. Therefore, if $n \geq m_k - 1$,

$$\mathbf{J}_k^n = (\lambda_k \mathbf{I} + \mathbf{S})^n = \sum_{i=0}^{n} \binom{n}{i} \lambda_k^{n-i} \mathbf{S}^i = \sum_{i=0}^{m_k-1} \binom{n}{i} \lambda_k^{n-i} \mathbf{S}^i\,. \tag{9.5}$$

In this finite sum, in view of $|\lambda_k| < 1$, in each term we have, as $n \longrightarrow \infty$,

$$\binom{n}{i} \lambda_k^{n-i} = \frac{n(n-1)\cdot(n-i+1)}{i!}\, \lambda_k^{n-i} \longrightarrow 0\,.$$

Consequently, $\mathbf{J}_k^n \longrightarrow \mathbf{O}$ as $n \longrightarrow \infty$. □

REMARK. A matrix norm is called **compatible** [*kompatibel*] with a given vector norm if $\|\mathbf{Cy}\| \leq \|\mathbf{C}\| \, \|\mathbf{y}\|$ for any square matrix $\mathbf{C}$ and any vector $\mathbf{y}$ of matching size. For a compatible matrix norm we have $\|\mathbf{C}\| \geq \rho(\mathbf{C})$ as is seen by choosing for $\mathbf{y}$ an eigenvector corresponding to an eigenvalue of largest absolute value, so that $\|\mathbf{Cy}\| = \rho(\mathbf{C}) \, \|\mathbf{y}\|$. ▼

Theorem 9.1.1 can be recast as follows: *linear fixed point iteration in* $\mathbb{E}^N$ *is globally convergent if and only if* $\rho(\mathbf{B}) < 1$. By a careful analysis of its proof we can also specify the speed of convergence[1].

[1] A sequence $\{\mathbf{x}_n\} \subset \mathbb{C}^N$ is said to converge R-linearly to $\mathbf{x}_\star$ if the **R-factor** or **root-convergence factor** [*Wurzel-Konvergenzfaktor*]

$$\kappa_R\{\mathbf{x}_n\} :\equiv \limsup_{n\to\infty} \|\mathbf{x}_n - \mathbf{x}_\star\|^{1/n}$$

lies in $(0,1)$. An iterative process $\mathcal{J}$ that generates a nonempty set $\mathcal{C}(\mathcal{J}, \mathbf{x}_\star)$ of sequences that converge R-linearly to the limit point $\mathbf{x}_\star$ has R-factor

$$\kappa_R(\mathcal{J}) :\equiv \sup\{R\{\mathbf{x}_n\} \mid \{\mathbf{x}_n\} \in \mathcal{C}(\mathcal{J}, \mathbf{x}_\star)\}\,.$$

THEOREM 9.1.2. *If $\rho(\mathbf{B}) < 1$ then, for any $\mathbf{x}_0$, the iterates of the linear fixed point iteration* (9.2) *satisfy*

$$\boxed{\limsup_{n\to\infty} \|\mathbf{x}_n - \mathbf{x}_\star\|^{1/n} \le \rho(\mathbf{B})\,,} \tag{9.6}$$

and for some $\mathbf{x}_0$ equality holds.
So, if $0 < \rho(\mathbf{B}) < 1$, linear fixed point iteration converges linearly with a root-convergence factor of at most $\rho(\mathbf{B})$, while if $\rho(\mathbf{B}) = 0$, it converges superlinearly.

PROOF. We have, as in the proof of Theorem 9.1.1,

$$\|\mathbf{x}_n - \mathbf{x}_\star\|^{1/n} = \|\mathbf{V}\mathbf{\Lambda}^n\mathbf{V}^{-1}(\mathbf{x}_0 - \mathbf{x}_\star)\|^{1/n} \le \|\mathbf{V}\|^{1/n}\|\mathbf{\Lambda}^n\mathbf{w}_0\|^{1/n}\,,$$

where $\mathbf{w}_0 :\equiv \mathbf{V}^{-1}(\mathbf{x}_0 - \mathbf{x}_\star)$. Since $\|\mathbf{V}\|^{1/n} \to 1$, it all depends on $\mathbf{\Lambda}^n$, whose diagonal blocks are eigenvalues or Jordan blocks $\mathbf{J}_k$. The powers of the latter were considered in (9.5). If we write, with $\rho :\equiv \rho(\mathbf{B})$,

$$\mathbf{J}_k^n = \rho^n \sum_{i=0}^{m_k-1} \binom{n}{i} \lambda_k^{-i} \left(\frac{\lambda_k}{\rho}\right)^n \mathbf{S}^i\,, \tag{9.7}$$

it is evident that only dominant eigenvalues (*i.e.,* eigenvalues of maximum absolute value) count, because for the others the nth power of the fraction λ_k/ρ tends to zero. If $|\lambda_k| = \rho$, then the dominant term in the sum (9.7) is

$$\binom{n}{m_k - 1} \lambda_k^{n-m_k+1}\mathbf{S}^{m_k-1}\,.$$

There may be several such terms. But, in any case, $\|\mathbf{\Lambda}^n\mathbf{w}_0\|^{1/n}$ behaves asymptotically at worst like the nth root of the absolute value of such a term, namely like $|\lambda_k| = \rho$. So this is asymptotically a bound for $\|\mathbf{x}_n - \mathbf{x}_\star\|^{1/n}$. To see that this bound is sharp we just have to choose $\mathbf{x}_0$ so that $\mathbf{w}_0$ is an eigenvector of $\mathbf{\Lambda}$ for a dominant eigenvalue λ_k. □

The applicability of fixed point or Jacobi iteration in practice is quite limited since, if $\rho(\mathbf{B}) < 1$ holds at all, then typically only with $\rho(\mathbf{B})$ nearly 1, so that convergence is very slow. But we need a good approximate solution in $n \ll N$ steps.

Since we cannot compute the nth **error (vector)** [*Fehler(vektor)*]

$$\boxed{\mathbf{d}_n :\equiv \mathbf{x}_n - \mathbf{x}_\star\,,} \tag{9.8}$$

for checking the convergence of an iteration, *i.e.,* for checking the “quality” of the approximate solution (or, iterate) $\mathbf{x}_n$ we use the nth **residual (vector)** [*Residuum, Residuenvektor*]

$$\boxed{\mathbf{r}_n :\equiv \mathbf{b} - \mathbf{A}\mathbf{x}_n\,.} \tag{9.9}$$

Note that

$$\mathbf{r}_n = -\mathbf{A}(\mathbf{x}_n - \mathbf{x}_\star) = -\mathbf{A}\mathbf{d}_n\,. \tag{9.10}$$

For the linear fixed point iteration we have

$$\mathbf{r}_n = \mathbf{b} - \mathbf{A}\mathbf{x}_n = \mathbf{B}\mathbf{x}_n + \mathbf{b} - \mathbf{x}_n = \mathbf{x}_{n+1} - \mathbf{x}_n\,,$$

so we can rewrite it as

$$\boxed{\mathbf{x}_{n+1} := \mathbf{x}_n + \mathbf{r}_n\,,} \tag{9.11}$$

and, by multiplying it by $-\mathbf{A}$, we obtain a recursion for the residual,

$$\boxed{\mathbf{r}_{n+1} := \mathbf{r}_n - \mathbf{A}\mathbf{r}_n = \mathbf{B}\mathbf{r}_n\,.} \tag{9.12}$$

We see that we can compute the residual $\mathbf{r}_n$ either according to the definition (9.9) or by using the recursion (9.12). In either case, we need one matrix-vector multiplication. Once, $\mathbf{r}_n$ is known, the new iterate $\mathbf{x}_{n+1}$

is obtained without any matrix-vector multiplication from (9.11). Mathematically both ways of computing $\mathbf{r}_n$ are equivalent, but roundoff errors may cause the results to differ.

Assignment (9.11) is a typical update formula for the iterate: the new approximation of the solution is obtained by adding a correction, here $\mathbf{r}_n$, to the old one.

From (9.12) it follows by induction that

$$\mathbf{r}_n = p_n(\mathbf{A})\mathbf{r}_0 \in \mathsf{span}\ \{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \dots, \mathbf{A}^n\mathbf{r}_0\}\ , \tag{9.13}$$

where p_n is a polynomial of exact degree n, actually $p_n(\zeta) = (1-\zeta)^n$. Moreover, from (9.11) we conclude that

$$\mathbf{x}_n = \mathbf{x}_0 + \mathbf{r}_0 + \cdots + \mathbf{r}_{n-1} \tag{9.14a}$$
$$= \mathbf{x}_0 + q_{n-1}(\mathbf{A})\mathbf{r}_0 \tag{9.14b}$$
$$\in \mathbf{x}_0 + \mathsf{span}\ \left\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \dots, \mathbf{A}^{n-1}\mathbf{r}_0\right\} \tag{9.14c}$$

with a polynomial q_{n-1} of exact degree $n-1$. We note that here $\mathbf{x}_0 + \mathsf{span}\ \left\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \dots, \mathbf{A}^{n-1}\mathbf{r}_0\right\}$ is an affine space, *i.e.,* a linear subspace shifted by the translation $\mathbf{x}_0$.

Building up $\mathbf{x}_n = \mathbf{x}_0 + q_{n-1}(\mathbf{A})$ and $\mathbf{r}_n = p_n(\mathbf{A})$ requires in particular a total of $n+1$ matrix-vector multiplications and this is the main work of the whole iteration process (unless $\mathbf{A}$ is extremely sparse). With roughly the same work we can construct any other vector $\mathbf{x}_n$ in the same affine space and its corresponding residual. We may hope that by making a better choice in this space we will find a better approximate solution with a smaller residual.

9.2 Iterations Based on Matrix Splittings

We introduced the Jacobi iteration as fixed point iteration (9.2) with a matrix $\mathbf{B}$ whose diagonal elements are zero. If we start from an arbitrary nonsingular system $\mathbf{A}\mathbf{x} = \mathbf{b}$ and let $\mathbf{D}$ be the diagonal matrix with the diagonal of $\mathbf{A}$, we thus replace the system by

$$\boxed{\mathbf{x} = \widehat{\mathbf{B}}\mathbf{x} + \widehat{\mathbf{b}}} \quad \text{with} \quad \boxed{\widehat{\mathbf{B}} :\equiv \mathbf{I} - \mathbf{D}^{-1}\mathbf{A}\,, \quad \widehat{\mathbf{b}} :\equiv \mathbf{D}^{-1}\mathbf{b}} \tag{9.15}$$

and apply the fixed point iteration $\mathbf{x}_{n+1} := \widehat{\mathbf{B}}\mathbf{x}_n + \widehat{\mathbf{b}}$.

Written in terms of the components of $\mathbf{x}_n :\equiv \begin{pmatrix} x_1^{(n)} & \dots & x_N^{(n)} \end{pmatrix}^{\mathsf{T}}$ one step — or, as it is often called, one **sweep** — of Jacobi iteration becomes

$$x_j^{(n+1)} := \frac{1}{a_{jj}} \left(b_j - \sum_{k=1}^{j-1} a_{jk}\, x_k^{(n)} - \sum_{k=j+1}^{N} a_{jk}\, x_k^{(n)} \right)\ , \quad j = 1, \dots, N\,. \tag{9.16}$$

It seems that Gauss was the one who discovered that the convergence is usually improved if we replace in the first sum of (9.16) the old values $x_n^{(k)}$ by the already computed new values $x_{n+1}^{(k)}$:

$$x_j^{(n+1)} := \frac{1}{a_{jj}} \left(b_j - \sum_{k=1}^{j-1} a_{jk}\, x_k^{(n+1)} - \sum_{k=j+1}^{N} a_{jk}\, x_k^{(n)} \right)\ , \quad j = 1, \dots, N\,. \tag{9.17}$$

This is called **Gauss-Seidel method** [*Gauss-Seidel-Verfahren oder Einzelschrittverfahren*]. To recast it in matrix notation we write $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{D} - \mathbf{E} - \mathbf{F}\,, \tag{9.18}$$

where $\mathbf{E}$ and $\mathbf{F}$ are strictly lower and upper triangular, respectively. Then (9.17) becomes

$$\mathbf{D}\mathbf{x}_{n+1} := \mathbf{E}\mathbf{x}_{n+1} + \mathbf{F}\mathbf{x}_n + \mathbf{b}$$

or

$$\boxed{(\mathbf{D}-\mathbf{E})\mathbf{x}_{n+1} :\!= \mathbf{F}\mathbf{x}_n + \mathbf{b}\,,} \tag{9.19}$$

which can be brought into the form of fixed point iteration with

$$\boxed{\widehat{\mathbf{B}} :\equiv (\mathbf{D}-\mathbf{E})^{-1}\mathbf{F}\,, \qquad \widehat{\mathbf{b}} :\equiv (\mathbf{D}-\mathbf{E})^{-1}\mathbf{b}\,.} \tag{9.20}$$

Of course, we never intend to compute the lower triangular matrix $(\mathbf{D}-\mathbf{E})^{-1}$, but implement this recursion as the sweep (9.17). But the fixed point representation tells us that for a convergence analysis we have to determine the spectral radius of $\widehat{\mathbf{B}}$.

In the notation (9.18) Jacobi iteration takes the form

$$\boxed{\mathbf{x}_{n+1} :\!= \mathbf{D}^{-1}(\mathbf{E}+\mathbf{F})\mathbf{x}_n + \mathbf{D}^{-1}\mathbf{b} = \mathbf{x}_n + \mathbf{D}^{-1}\mathbf{r}_n} \tag{9.21}$$

and has the iteration matrix

$$\boxed{\widehat{\mathbf{B}} :\equiv \mathbf{D}^{-1}(\mathbf{E}+\mathbf{F})\,.} \tag{9.22}$$

A classical idea is to apply here **relaxation** [*Relaxation*]: we multiply the correction $\mathbf{D}^{-1}\mathbf{r}_n$ in the Jacobi iteration (9.21) by a **relaxation factor** [*Relaxationsfaktor*] ω, which typically satisfies $0 < \omega < 1$:

$$\boxed{\mathbf{x}_{n+1} :\!= \mathbf{x}_n + \mathbf{D}^{-1}\mathbf{r}_n\omega\,.} \tag{9.23}$$

Naturally this method is called the **damped Jacobi iteration** [*gedämpftes Gesamtschritt-Verfahren*]. One might also call it the **Jacobi underrelaxation (JUR) method**, but it has also been referred to as **Jacobi overrelaxation (JOR) method** or as **stationary Richardson iteration**. (In general, Richardson iteration is non-stationary, that is, $\omega :\equiv \omega_n$ depends on n.)

Note that

$$\begin{aligned}
\mathbf{x}_{n+1} &= \mathbf{x}_n + \mathbf{D}^{-1}\left[\mathbf{b} - \mathbf{A}\mathbf{x}_n\right]\omega \\
&= \mathbf{x}_n(1-\omega) + \mathbf{D}^{-1}\left[(\mathbf{E}+\mathbf{F})\mathbf{x}_n + \mathbf{b}\right]\omega \\
&= \mathbf{x}_n(1-\omega) + \mathbf{x}_{n+1}^{\text{Jac}}\omega\,.
\end{aligned} \tag{9.24}$$

So the new iterate $\mathbf{x}_{n+1}$ is the weighted mean of the old iterate $\mathbf{x}_n$ and one step of Jacobi, (9.21), starting from $\mathbf{x}_n$.

The iteration matrix is now

$$\boxed{\widehat{\mathbf{B}} :\equiv (1-\omega)\mathbf{I} + \omega\mathbf{D}^{-1}(\mathbf{E}+\mathbf{F}) = \mathbf{I} - \omega\mathbf{D}^{-1}\mathbf{A}\,.} \tag{9.25}$$

So, if the spectrum of $\mathbf{D}^{-1}\mathbf{A}$ lies, *e.g.,* on the interval $[\alpha,\beta]$ of the positive real axis, the one of $\widehat{\mathbf{B}}$ lies on $[1-\omega\beta, 1-\omega\alpha]$. Therefore, if $\omega > 0$ is chosen such that $\omega\beta < 2$, we will have convergence (according to Theorem 9.1.1). Moreover, ω could be chosen easily such that $\max\{|1-\omega\beta|, |1-\omega\alpha|\}$ is minimal, and thus ω is optimal.

It was [You50] who realized in his dissertation that the idea of relaxation is particularly effective in connection with the Gauss-Seidel method: we take *componentwise* a weighted mean of the old iterate $x_n^{(j)}$ and a step of Gauss-Seidel for that component, as given by (9.17):

$$\begin{aligned}
x_j^{(n+1)} &:= x_j^{(n)}(1-\omega) + x_j^{(n+1,\mathrm{GS})}\omega \qquad (9.26)\\
&= x_j^{(n)}(1-\omega) + \frac{\omega}{a_{jj}}\left(b_j - \sum_{k=1}^{j-1} a_{jk}\,x_k^{(n+1)} - \sum_{k=j+1}^{N} a_{jk}\,x_k^{(n)}\right)\,, \\
&\quad j = 1,\ldots,N\,.
\end{aligned}$$

This translates into

$$\boxed{(\mathbf{D}-\omega\mathbf{E})\mathbf{x}_{n+1} :\!= [(1-\omega)\mathbf{D} + \omega\mathbf{F}]\mathbf{x}_n + \omega\mathbf{b}} \tag{9.27}$$

or the fixed point iteration with

$$\boxed{\widehat{\mathbf{B}} :\equiv \left(\tfrac{1}{\omega}\mathbf{D} - \mathbf{E}\right)^{-1}\left[\left(\tfrac{1}{\omega} - 1\right)\mathbf{D} + \mathbf{F}\right], \qquad \widehat{\mathbf{b}} :\equiv \left(\tfrac{1}{\omega}\mathbf{D} - \mathbf{E}\right)^{-1}\mathbf{b}} \tag{9.28}$$

and is called **successive overrelaxation (SOR) method** [*Verfahren der sukzessiven Überrelaxation*]. It can be shown that for fixed ω in the interval $0 < \omega < 2$ the method converges for any system with Hpd matrix $\mathbf{A}$. (For a proof, see, e.g., p. 56 of [SRS68].)

There remains the question on how to choose ω so that convergence is fastest. [You50] derived a beautiful theory leading to the optimal ω for matrices that have the so-called **Property A** and are **consistently ordered** [*konsistent geordnet*]. The class of matrices with Property A includes many examples obtained by discretizing partial differential equations with the finite difference method. The property is (by definition) invariant under simultaneous column and row permutations of the matrix, but for optimal convergence we need rows and columns so-called consistently ordered.

Although we do not exactly define Property A and consistent ordering here, we give a statement of Young's convergence result. For partial proofs under slightly varying assumptions see, *e.g.,* pages 59–65 and 208–211 of [SRS68] (for $\mathbf{A}$ spd), pages 112–116 of [Saa96] (discussion of consistent ordering, but no proof of optimal ω), and pages 149–154 of [Gre97] (no discussion of consistent ordering).

THEOREM 9.2.1. *For a consistently ordered matrix $\mathbf{A}$ with Property A and real eigenvalues of $\mathbf{D}^{-1}\mathbf{A}$, the optimal relaxation factor ω_{opt} of the SOR method is*

$$\boxed{\omega_{\text{opt}} :\equiv \frac{2}{1 + \sqrt{1 - \lambda_{\max}^2}}\,,} \tag{9.29}$$

where, in terms of notation (9.18), *$\lambda_{\max}$ is the largest eigenvalue of the matrix $\mathbf{D}^{-1}(\mathbf{E} + \mathbf{F})$, which is the iteration matrix of the Jacobi method. The spectral radius of the optimal SOR iteration matrix $\widehat{\mathbf{B}}_{\text{opt}}$ is then*

$$\boxed{\rho(\widehat{\mathbf{B}}_{\text{opt}}) = \omega_{\text{opt}} - 1\,,} \tag{9.30}$$

An interesting aspect is that, under the assumptions of the theorem, by SOR with optimal ω, the real eigenvalues of $\mathbf{D}^{-1}\mathbf{A}$ are mapped onto the circle with radius $\rho(\widehat{\mathbf{B}}_{\text{opt}})$. So all the eigenvalues of $\widehat{\mathbf{B}}_{\text{opt}}$, although in general complex, have the same absolute value.

A further related method is the **symmetric SOR (SSOR) method**, where each step is a double step consisting of an SOR step followed by a backward SOR step; in short form:

$$\boxed{\begin{aligned} (\mathbf{D} - \omega\mathbf{E})\mathbf{x}_{n+\frac{1}{2}} &:= [(1-\omega)\mathbf{D} + \omega\mathbf{F}]\mathbf{x}_n + \omega\mathbf{b}\,, \\ (\mathbf{D} - \omega\mathbf{F})\mathbf{x}_{n+1} &:= [(1-\omega)\mathbf{D} + \omega\mathbf{E}]\mathbf{x}_{n+\frac{1}{2}} + \omega\mathbf{b}\,. \end{aligned}} \tag{9.31}$$

The convergence analysis is even more complicated than for SOR.

There are also block versions of all these schemes.

Except for SSOR all the methods discussed in this section can be viewed as applications of the general principle of improving fixed point iteration by **matrix splitting**: we write

$$\boxed{\mathbf{A} = \mathbf{M} - \mathbf{N}\,,} \tag{9.32}$$

where $\mathbf{M}$ is chosen so that, for any $\mathbf{y}$, the system $\mathbf{M}\mathbf{x} = \mathbf{y}$ is easy to solve for $\mathbf{x}$. Then an iterative method can be defined by

$$\boxed{\mathbf{M}\mathbf{x}_{n+1} := \mathbf{N}\mathbf{x}_n + \mathbf{b} = \mathbf{M}\mathbf{x}_n + \mathbf{r}_n\,.} \tag{9.33}$$

In every step a linear system with the matrix $\mathbf{M}$ has to be solved. Formally, the method is equivalent with

$$\boxed{\mathbf{x}_{n+1} := \mathbf{M}^{-1}\mathbf{N}\mathbf{x}_n + \mathbf{M}^{-1}\mathbf{b} = \mathbf{x}_n + \mathbf{M}^{-1}\mathbf{r}_n\,.} \tag{9.34}$$

This is just a linear fixed point iteration with

$$\boxed{\widehat{\mathbf{B}} :\equiv \mathbf{M}^{-1}\mathbf{N}\,, \qquad \widehat{\mathbf{b}} :\equiv \mathbf{M}^{-1}\mathbf{b}\,.} \tag{9.35}$$

Therefore, the iteration converges if and only if $\rho(\mathbf{M}^{-1}\mathbf{N}) < 1$. So, the aim must be to find splittings (9.32) where this spectral radius is small. Unfortunately, in general it is difficult to conclude from the spectrum of $\mathbf{A}$ on that of $\widehat{\mathbf{B}} = \mathbf{M}^{-1}\mathbf{N}$. But there is the important class of so-called **M-matrices** [*M-Matrizen*] and corresponding so-called **regular splittings** [*reguläre Splittings*] where at least convergence can be proved.

Table 9.1 summarizes the meanings of $\mathbf{M}$ and $\mathbf{N}$ in our above treated classical methods. These methods, in particular SOR, were very popular in the 1950ies and 1960ies, but they are hardly used nowadays for the original purpose. We will come back to them in Section 10.1 on basic preconditioning techniques, however.

Table 9.1 Some matrix splittings based on $\mathbf{A} = \mathbf{D} - \mathbf{E} - \mathbf{F}$.

method	$\mathbf{M}$	$\mathbf{N}$
Jacobi	$\mathbf{D}$	$\mathbf{E} + \mathbf{F}$
Gauss-Seidel	$\mathbf{D} - \mathbf{E}$	$\mathbf{F}$
damped Jacobi	$\frac{1}{\omega}\mathbf{D}$	$\left(\frac{1}{\omega} - 1\right)\mathbf{D} + \mathbf{E} + \mathbf{F}$
SOR	$\frac{1}{\omega}\mathbf{D} - \mathbf{E}$	$\left(\frac{1}{\omega} - 1\right)\mathbf{D} + \mathbf{F}$
backward SOR	$\frac{1}{\omega}\mathbf{D} - \mathbf{F}$	$\left(\frac{1}{\omega} - 1\right)\mathbf{D} + \mathbf{E}$

9.3 Krylov Subspaces and Krylov Space Solvers

The discussion of the Jacobi method in Section 9.1 suggests to look for better approximate solutions lying in the following affine space:

$$\mathbf{x}_n \in \mathbf{x}_0 + \mathsf{span}\left\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \dots, \mathbf{A}^{n-1}\mathbf{r}_0\right\},$$

see (9.14c). We first investigate the subspace that appears in this formula.

DEFINITION. Given a nonsingular $N \times N$ matrix $\mathbf{A}$ and an N-vector $\mathbf{y} \neq \mathbf{o}$, the nth **Krylov (sub)space** $\mathcal{K}_n(\mathbf{A}, \mathbf{y})$ [*Krylov–Raum*] generated by $\mathbf{A}$ from $\mathbf{y}$ is

$$\boxed{\mathcal{K}_n :\equiv \mathcal{K}_n(\mathbf{A}, \mathbf{y}) :\equiv \mathsf{span}\,(\mathbf{y}, \mathbf{A}\mathbf{y}, \dots, \mathbf{A}^{n-1}\mathbf{y}).} \tag{9.36}$$

▲

Clearly, by this definition, whenever $\mathbf{z} \in \mathcal{K}_n(\mathbf{A}, \mathbf{y})$, there is a polynomial p of degree at most $n - 1$ such that $\mathbf{z} = p(\mathbf{A})\mathbf{y}$. In general, this polynomial may be not unique since the spanning set in (9.36) may be linearly dependent. We can say more about this in a moment.

Definition (9.36) associates with a matrix $\mathbf{A}$ and a starting vector $\mathbf{y}$ a whole nested sequence of Krylov subspaces:

$$\mathcal{K}_1 \subseteq \mathcal{K}_2 \subseteq \mathcal{K}_3 \subseteq \dots .$$

The following lemma answers the question of the equality signs.

LEMMA 9.3.1. *There is a positive integer* $\bar{\nu} :\equiv \bar{\nu}(\mathbf{y}, \mathbf{A})$ *such that*

$$\boxed{\dim\ \mathcal{K}_n(\mathbf{A}, \mathbf{y}) = \begin{cases} n & \text{if} \quad n \leq \bar{\nu}\,, \\ \bar{\nu} & \text{if} \quad n \geq \bar{\nu}\,. \end{cases}}$$

The inequalities $1 \leq \bar{\nu} \leq N$ *hold, and* $\bar{\nu} < N$ *is possible if* $N > 1$.

DEFINITION. The positive integer $\bar{\nu} :\equiv \bar{\nu}(\mathbf{y}, \mathbf{A})$ of Lemma 9.3.1 is called **grade of $\mathbf{y}$ with respect to $\mathbf{A}$** [*Grad von* $\mathbf{y}$ *bezüglich* $\mathbf{A}$]. ▲

PROOF of Lemma 9.3.1. By definition (9.36),

$$\mathcal{K}_n :\equiv \mathcal{K}_n(\mathbf{A}, \mathbf{y}) :\equiv \mathsf{span}\,(\mathbf{y}, \mathbf{A}\mathbf{y}, \dots, \mathbf{A}^{n-1}\mathbf{y})\,,$$

with some $\mathbf{y} \neq \mathbf{o}$. So, $\dim \mathcal{K}_1 = 1$, and if $\mathbf{y}, \mathbf{Ay}, \dots, \mathbf{A}^{n-1}\mathbf{y}$ are linearly independent, then $\dim \mathcal{K}_n = n$. Let $n = \bar{\nu}$ be the first n for which $\mathbf{y}, \mathbf{Ay}, \dots, \mathbf{A}^{n-1}\mathbf{y}, \mathbf{A}^{n}\mathbf{y}$ are linearly dependent. Then $\mathbf{A}^{n}\mathbf{y} = \mathbf{A}^{\bar{\nu}}\mathbf{y}$ is a linear combination of the linearly independent vectors $\mathbf{y}, \mathbf{Ay}, \dots, \mathbf{A}^{\bar{\nu}-1}\mathbf{y}$:

$$\mathbf{A}^{\bar{\nu}}\mathbf{y} = \mathbf{y}\gamma_0 + \mathbf{Ay}\gamma_1 + \cdots + \mathbf{A}^{\bar{\nu}-1}\mathbf{y}\gamma_{\bar{\nu}-1}\,. \tag{9.37}$$

Here, $\gamma_0 \neq 0$, because otherwise, after multiplication by $\mathbf{A}^{-1}$, we would have

$$\mathbf{A}^{\bar{\nu}-1}\mathbf{y} = \mathbf{y}\gamma_1 + \mathbf{Ay}\gamma_2 + \cdots + \mathbf{A}^{\bar{\nu}-2}\mathbf{y}\gamma_{\bar{\nu}-1}\,,$$

in contrast to the assumption that the vectors on the right-hand side are linearly independent.

If we consider the definition (9.36) for some $n > \bar{\nu}$, then, by using (9.37), all terms $\mathbf{A}^{k}\mathbf{y}$ with $k \geq \bar{\nu}$ in the span can be recursively replaced by sums of terms with $k < \bar{\nu}$. So, the dimension cannot be larger than $\bar{\nu}$; and of course, it cannot be smaller since $\mathcal{K}_n \supseteq \mathcal{K}_{\bar{\nu}}$ if $n \geq \bar{\nu}$.

If we choose $\mathbf{y}$ as an eigenvector of $\mathbf{A}$, then $\bar{\nu} = 1$, so $\bar{\nu}$ can be smaller than N if $N > 1$. □

We can say more and understand the structure of the Krylov subspaces better if we consider the Jordan canonical form of $\mathbf{A}$ (as in the proof if Theorem 9.1.1) and recall the notion of the minimal polynomial of $\mathbf{A}$.

So we let $\mathbf{A} = \mathbf{VJV}^{-1}$ be the Jordan decomposition of $\mathbf{A}$, *i.e.,* $\mathbf{J}$ is a block-diagonal matrix of the form

$$\mathbf{J} = \begin{pmatrix} \mathbf{J}_1 & & \\ & \ddots & \\ & & \mathbf{J}_\mu \end{pmatrix}, \quad \text{where } \mathbf{J}_k = \begin{pmatrix} \lambda_k & 1 & & \\ & \lambda_k & \ddots & \\ & & \ddots & 1 \\ & & & \lambda_k \end{pmatrix} \in \mathbb{C}^{m_k \times m_k} \tag{9.38}$$

are either bidiagonal Jordan blocks or 1×1 blocks (λ_k). Some of the eigenvalues λ_k in the different blocks may coincide, and we choose to rearrange the blocks so that a full set of Jordan blocks of maximum size m_k corresponding to all the $\bar{\mu}$ distinct eigenvalues $\lambda_1, \dots, \lambda_{\bar{\mu}}$ is at the top. So, if $\bar{\mu} < \mu$, then $\lambda_{\bar{\mu}+1}, \dots, \lambda_\mu$ all coincide with some eigenvalue among the first $\bar{\mu}$, and the block sizes m_k for $k > \bar{\mu}$ are at most as large as the corresponding ones for the same eigenvalue with $k \leq \bar{\mu}$. Then the **minimal polynomial** [*Minimalpolynom*] $\widehat{\chi}_{\mathbf{A}}$ of $\mathbf{A}$ is defined by

$$\boxed{\widehat{\chi}_{\mathbf{A}}(t) :\equiv \prod_{k=1}^{\bar{\mu}} (t - \lambda_k)^{m_k}\,.} \tag{9.39}$$

Note, that the minimal polynomial is a divisor of the characteristic polynomial

$$\boxed{\chi_{\mathbf{A}}(t) :\equiv \prod_{k=1}^{\mu} (t - \lambda_k)^{m_k}} \tag{9.40}$$

(where the product runs now over all μ diagonal blocks $\mathbf{J}_k$ of the Jordan decomposition (9.38)). Its degree $\partial\widehat{\chi}_{\mathbf{A}}$ is

$$M :\equiv \partial\widehat{\chi}_{\mathbf{A}} = \sum_{k=1}^{\bar{\mu}} m_k \leq \sum_{k=1}^{\mu} m_k = \partial\psi = N\,.$$

We claim that if we insert $t := \mathbf{A}$ into the minimal polynomial (so that it becomes a matrix polynomial), we get the zero matrix.

THEOREM 9.3.2. *If $\widehat{\chi}_{\mathbf{A}}$ denotes the minimal polynomial of $\mathbf{A}$, then*

$$\widehat{\chi}_{\mathbf{A}}(\mathbf{A}) = \prod_{k=1}^{\bar{\mu}} (\mathbf{A} - \lambda_k \mathbf{I})^{m_k} = \mathbf{O}\,. \tag{9.41}$$

PROOF. We have

$$\widehat{\chi}_{\mathbf{A}}(\mathbf{A}) = \mathbf{V}\,\widehat{\chi}_{\mathbf{A}}(\mathbf{J})\,\mathbf{V}^{-1} = \mathbf{V}\left(\prod_{k=1}^{\bar{\mu}} (\mathbf{J} - \lambda_k \mathbf{I})^{m_k}\right)\mathbf{V}^{-1}\,,$$

and here all the factors of the product, and hence the product itself, is block diagonal. Consider, the block $(\mathbf{J}_k - \lambda_k \mathbf{I})^{m_k}$ of the kth factor. In the notation of (9.5) it equals $\mathbf{S}^{m_k}$, where, in this factor, $\mathbf{S}$ is the $m_k \times m_k$ matrix with ones on the upper bidiagonal and zeros elsewhere. So, $\mathbf{S}^{m_k} = \mathbf{O}$. Hence, in the product in question, there is for each k a factor where the kth block is zero. Consequently, the whole product is a zero matrix, and thus also $\widehat{\chi}_{\mathbf{A}}(\mathbf{J})$ and $\widehat{\chi}_{\mathbf{A}}(\mathbf{A})$. □

Since the minimal polynomial is a divisor of the characteristic polynomial, the following famous **Caley-Hamilton theorem** follows immediately.

COROLLARY 9.3.3. *If $\chi_{\mathbf{A}}$ denotes the characteristic polynomial of $\mathbf{A}$, then $\chi_{\mathbf{A}}(\mathbf{A}) = \mathbf{O}$.*

Now we are ready to prove the following result on the grade $\bar{\nu}$ that appeared in Lemma 9.3.1.

LEMMA 9.3.4. *The nonnegative integer $\bar{\nu}$ of Lemma 9.3.1 satisfies*

$$\boxed{\bar{\nu} = \min\left\{n \;\middle|\; \mathbf{A}^{-1}\mathbf{y} \in \mathcal{K}_n(\mathbf{A},\mathbf{y})\right\} \le \partial\widehat{\chi}_{\mathbf{A}},}$$

where $\partial\widehat{\chi}_{\mathbf{A}}$ denotes the degree of the minimal polynomial of $\mathbf{A}$.

PROOF. Multiplying (9.37) by $\mathbf{A}^{-1}$ we see that

$$\mathbf{A}^{-1}\mathbf{y} = \left(\mathbf{A}^{\bar{\nu}-1}\mathbf{y} - \mathbf{A}^{\bar{\nu}-2}\mathbf{y}\gamma_{\bar{\nu}-1} - \cdots - \mathbf{A}\mathbf{y}\gamma_2 - \mathbf{y}\gamma_1\right)\frac{1}{\gamma_0}, \tag{9.42}$$

so $\mathbf{A}^{-1}\mathbf{y} \in \mathcal{K}_{\bar{\nu}}(\mathbf{A},\mathbf{y})$. We cannot replace $\bar{\nu}$ by some $n < \bar{\nu}$ here, because this would lead to a contradiction to the minimality of $\bar{\nu}$ in (9.37), and hence, to the definition of $\bar{\nu}$.

It remains to show that $\bar{\nu} \le \partial\widehat{\chi}_{\mathbf{A}}$. The product formula for $\widehat{\chi}_{\mathbf{A}}(\mathbf{A}) = \mathbf{O}$ in (9.41) could be written as a linear combination of $\mathbf{I}$, $\mathbf{A}$, ..., $\mathbf{A}^M$ that is zero, but the coefficient of $\mathbf{A}^M$ is 1 (here, again, $M :\equiv \partial\widehat{\chi}_{\mathbf{A}}$). This remains valid if we post-multiply each term by any $\mathbf{y}$. So, for any $\mathbf{y}$, the Krylov subspace $\mathcal{K}_{M+1}(\mathbf{A},\mathbf{y})$ has dimension at most M, and in view of Lemma 9.3.1 the same is true for $\mathcal{K}_n(\mathbf{A},\mathbf{y})$ for any n. □

Of course, the actual dimension may be smaller for some $\mathbf{y}$. In fact, it is, if in the representation of $\mathbf{y}$ in terms of the basis associated with the Jordan decomposition of $\mathbf{A}$ some of the relevant coordinates vanish.

We can conclude that as long as $n \le \bar{\nu}(\mathbf{y},\mathbf{A})$, the vectors $\mathbf{y}$, $\mathbf{A}\mathbf{y}$, ..., $\mathbf{A}^{n-1}\mathbf{y}$ in (9.36) are linearly independent, and thus the polynomial p representing some $\mathbf{z} = p(\mathbf{A})\mathbf{y} \in \mathcal{K}_n(\mathbf{A},\mathbf{y})$ is uniquely determined. In other words, as long as $n \le \bar{\nu}$, there is a natural one-to-one correspondence (actually, an isomorphism) between the linear space $\mathcal{K}_n$ and the linear space $\mathcal{P}_{n-1}$ of polynomials of degree at most $n-1$.

Unfortunately, even for $n \le \bar{\nu}$ the vectors $\mathbf{y}$, $\mathbf{A}\mathbf{y}$, ..., $\mathbf{A}^{n-1}\mathbf{y}$ form typically a very ill-conditioned basis for $\mathcal{K}_n$, since they tend to be nearly linearly dependent. In fact, one can easily show that if $\mathbf{A}$ has a unique eigenvalue of largest absolute value and of algebraic multiplicity one, and if $\mathbf{y}$ is not orthogonal to a corresponding normalized eigenvector $\mathbf{v}$, then $\mathbf{A}^k\mathbf{y}/\|\mathbf{A}^k\mathbf{y}\| \to \pm\mathbf{v}$ as $k \to \infty$. Therefore, in practice, we will never make use of this so-called **Krylov basis** [*Krylovbasis*].

In connection with the iterative solution of a linear system Lemma 9.3.4 yields a most welcome corollary.

COROLLARY 9.3.5. *Let $\mathbf{x}_\star$ be the solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$ and let $\mathbf{x}_0$ be any initial approximation of it and $\mathbf{r}_0 :\equiv \mathbf{b} - \mathbf{A}\mathbf{x}_0$ the corresponding residual. Moreover, let $\bar{\nu} :\equiv \bar{\nu}(\mathbf{r}_0,\mathbf{A})$. Then*

$$\boxed{\mathbf{x}_\star \in \mathbf{x}_0 + \mathcal{K}_{\bar{\nu}}(\mathbf{A},\mathbf{r}_0)\,.}$$

PROOF. By (9.8), (9.10), and by Lemma 9.3.4,

$$\mathbf{x}_\star - \mathbf{x}_0 = \mathbf{d}_0 = -\mathbf{A}^{-1}\mathbf{r}_0 \in \mathcal{K}_{\bar{\nu}}(\mathbf{A},\mathbf{r}_0)\,.$$

□

This corollary shows that if we choose $\mathbf{x}_n$ from the affine space $\mathbf{x}_0 + \mathcal{K}_n(\mathbf{A},\mathbf{r}_0)$ there is a chance that we find the exact solution within at most $\bar{\nu}$ steps. We say then that our method has the **finite termination property**. We will see that it is easy to deduce methods that have this property. In fact, it suffices to insure that the residuals $\mathbf{r}_n$ are linearly independent as long as they are nonzero. Of course, once $\mathbf{r}_n = \mathbf{o}$ for some n, the linear system is solved.

However, in practice the finite termination property is nearly irrelevant, since $\bar{\nu}$ is normally much larger than the maximum number of iterations we are willing to execute. We rather want an approximation with sufficiently small residual quickly.

The corollary and the analogy to the relation (9.14c) now motivate the following definition.

DEFINITION. A **(typical) Krylov space method for solving a linear system** [*Krylov-Raum-Methode*] $\mathbf{A}\mathbf{x} = \mathbf{b}$ or, briefly, a **Krylov space solver** [2], is an iterative method starting from some initial approximation $\mathbf{x}_0$ and the corresponding residual $\mathbf{r}_0 :\equiv \mathbf{b} - \mathbf{A}\mathbf{x}_0$ and generating for all, or at least most n, iterates $\mathbf{x}_n$ such that

$$\boxed{\mathbf{x}_n - \mathbf{x}_0 = q_{n-1}(\mathbf{A})\mathbf{r}_0 \in \mathcal{K}_n(\mathbf{A}, \mathbf{r}_0)} \tag{9.43}$$

with a polynomial q_{n-1} of exact degree $n-1$. ▲

We first note that (9.43) implies that

$$\mathbf{d}_n - \mathbf{d}_0 = q_{n-1}(\mathbf{A})\mathbf{r}_0 \in \mathcal{K}_n(\mathbf{A}, \mathbf{r}_0)\,, \tag{9.44}$$

$$\mathbf{r}_n - \mathbf{r}_0 = -\mathbf{A}q_{n-1}(\mathbf{A})\mathbf{r}_0 \in \mathbf{A}\mathcal{K}_n(\mathbf{A}, \mathbf{r}_0)\,. \tag{9.45}$$

¿From the second equation we find a result that shows that these methods generalize the Jacobi iteration; *cf.* (9.13).

LEMMA 9.3.6. *The residuals of a Krylov space solver satisfy*

$$\boxed{\mathbf{r}_n = p_n(\mathbf{A})\mathbf{r}_0 \in \mathbf{r}_0 + \mathbf{A}\mathcal{K}_n(\mathbf{A}, \mathbf{r}_0) \subseteq \mathcal{K}_{n+1}(\mathbf{A}, \mathbf{r}_0)\,,} \tag{9.46}$$

where p_n is a polynomial of degree n, which is related to the polynomial q_{n-1} of (9.43) *by*

$$\boxed{p_n(\zeta) = 1 - \zeta q_{n-1}(\zeta)\,.} \tag{9.47}$$

In particular,

$$\boxed{p_n(0) = 1\,.} \tag{9.48}$$

DEFINITION. The polynomials $p_n \in \mathcal{P}_n$ in (9.46) are the **residual polynomials** [*Residualpolynome*] of the Krylov space solver. We refer to the condition (9.48) as the **consistency condition** [*Konsistenz-Bedingung*] for these polynomials. ▲

As we will see, for some Krylov space solvers there may exist exceptional situations, where for some n the iterate $\mathbf{x}_n$ and the residual $\mathbf{r}_n$ are not defined. There are also **atypical Krylov space methods** [*atypische Krylov-Raum-Methode*] where the approximation space for $\mathbf{x}_n - \mathbf{x}_0$ is still a Krylov space, but one that differs from $\mathcal{K}_n(\mathbf{A}, \mathbf{r}_0)$.

Krylov space methods are a very important class of numerical methods. With respect to the “influence on the development and practice of science and engineering in the 20th century”, they are considered as one of the ten most important classes [DS00, van00].

Krylov space methods have been the most important topic of sparse matrix analysis through the whole second part of the last century, although there exist other approaches for solving sparse linear systems that do not fit into this class. Moreover, the Krylov space approach is also applicable to eigenvalue problems.

As we will see there are various ways to derive suitable Krylov space solvers. Depending on the way they were used and the time period, various names have been given to the class of methods called Krylov space solvers here: **gradient methods** [Rut59], **semi-iterative methods** [Var62, You71], **polynomial acceleration methods**, **polynomial preconditioners** [BBC$^+$94], **Krylov subspace iterations** [van00].

In view of $\mathbf{r}_n = -\mathbf{A}\mathbf{d}_n$ (see (9.10) and (9.44)–(9.45)) Lemma 9.3.6 implies an analogous result on the error vectors.

[2] Many authors use the term **Krylov subspace method** instead, but, of course, any subspace of a linear space is itself a linear space. We certainly want to avoid the German “Krylov-Unterraum-Methode”.

LEMMA 9.3.7. *The error vectors of a Krylov space solver satisfy*

$$\mathbf{d}_n = p_n(\mathbf{A})\mathbf{d}_0 \in \mathcal{K}_n(\mathbf{A}, \mathbf{d}_0)\,, \tag{9.49}$$

where p_n is the nth residual polynomial.

Note, however, that the Krylov space $\mathcal{K}_n(\mathbf{A}, \mathbf{d}_0)$ that appears here, is different from the one we normally consider, $\mathcal{K}_n(\mathbf{A}, \mathbf{r}_0)$.

9.4 Chebyshev Iteration*

As we have mentioned beforehand, the simplest way — though not always the best way — to check the convergence of a Krylov space solver is to evaluate a norm of the residual vector. A natural approach to designing a Krylov space solver is therefore to try to minimize a residual norm. The representation (9.46), $\mathbf{r}_n = p_n(\mathbf{A})\mathbf{r}_0$, of the residual and the spectral decomposition $\mathbf{AU} = \mathbf{U\Lambda}$ of the matrix $\mathbf{A}$ yield $\mathbf{r}_n = \mathbf{U}p_n(\mathbf{\Lambda})\mathbf{U}^{-1}\mathbf{r}_0$. Therefore, in the 2-norm of the vectors and the induced spectral norm for the matrices,

$$\boxed{\|\mathbf{r}_n\|/\|\mathbf{r}_0\| \le \kappa(\mathbf{U})\|p_n(\mathbf{\Lambda})\|\,,} \tag{9.50}$$

where $\kappa(\mathbf{U}) :\equiv \|\mathbf{U}\|\ \|\mathbf{U}^{-1}\|$ is the (spectral) condition number of $\mathbf{U}$. If $\mathbf{\Lambda}$ is diagonal, $\mathbf{\Lambda} \equiv:$ diag $\{\lambda_1, \dots, \lambda_N\}$,

$$\|p_n(\mathbf{\Lambda})\| = \max_{i=1,\dots,N} |p_n(\lambda_i)|\,, \tag{9.51}$$

so the problem of finding a good Krylov space solver for a particular $\mathbf{A}$ can be reduced to the approximation problem

$$\boxed{\max_{i=1,\dots,N} |p_n(\lambda_i)| = \min! \quad \text{subject to} \quad p_n \in \mathcal{P}_n, \quad p_n(0) = 1\,.} \tag{9.52}$$

There are some problems with this approach, however. First, in general we cannot assume that the eigenvalues of $\mathbf{A}$ are known; their computation is normally much more costly than solving a linear system with the same matrix. Second, the condition number $\kappa(\mathbf{U})$ may be very large, and the inequality in (9.50) may be far from sharp. Third, the approximation problem (9.52) is very difficult to solve if there are complex eigenvalues.

But let us assume here that $\mathbf{A}$ is real symmetric or Hermitian and positive definite (*i.e.,* spd or Hpd), so that $\mathbf{\Lambda}$ is diagonal, the eigenvalues are positive, and $\mathbf{U}$ is unitary and thus $\kappa(\mathbf{U}) = 1$. There is still the difficulty that the individual eigenvalues are not known. Any norm of $\mathbf{A}$ yields an upper bound for the eigenvalues, and often a positive lower bound can be found somehow. So, assume

$$\lambda_i \in \mathcal{I} :\equiv [\alpha - \delta, \alpha + \delta] \qquad (i = 1, \dots, N) \tag{9.53}$$

with $\alpha > 0$ and $0 < \delta < \alpha$. Then, the following holds:

$$\boxed{\|\mathbf{r}_n\|/\|\mathbf{r}_0\| \le \|p_n(\mathbf{\Lambda})\| = \max_{i=1,\dots,N} |p_n(\lambda_i)| \le \max_{\tau\in\mathcal{I}} |p_n(\tau)|\,.} \tag{9.54}$$

It suggests to replace the approximation problem (9.52) by

$$\boxed{\max_{\tau\in\mathcal{I}} |p_n(\tau)| = \min! \quad \text{subject to} \quad p_n \in \mathcal{P}_n, \quad p_n(0) = 1\,.} \tag{9.55}$$

We claim that this real polynomial approximation problem can be solved analytically and that the optimal polynomial is just a shifted and scaled versions of the classical **Chebyshev polynomial** [*Tschebyscheff–Polynom*] of degree n, which on $\mathbb{R}$ is defined by

$$T_n(\xi) :\equiv \begin{cases} \cos(n \arccos(\xi)) & \text{if} \quad |\xi| \le 1\,, \\ \left((\text{sign}\,(\xi)\right)^n \cosh(n \,\text{arcosh}(|\xi|)) & \text{if} \quad |\xi| \ge 1\,, \end{cases}$$

and satisfies the three-term recursion

$$\boxed{T_{n+1}(\xi) := 2\xi T_n(\xi) - T_{n-1}(\xi) \qquad (n > 1)} \tag{9.56}$$

with initial values $T_0(\xi) := 1, T_1(\xi) := \xi$. The recursion is just a translation of the identity $2cos\phi \cos n\phi = \cos(n+1)\phi + \cos(n-1)\phi$ under the substitution $\xi = \cos\phi$. Clearly, $T_n(\xi) = \cos(n \arccos(\xi))$ oscillates on the interval $[-1, 1]$ between its minima and maxima ± 1, of which there are a total of $n + 1$, including two at the endpoints ± 1; see Figure 9.1 for two clippings of the graph of T_{11}.

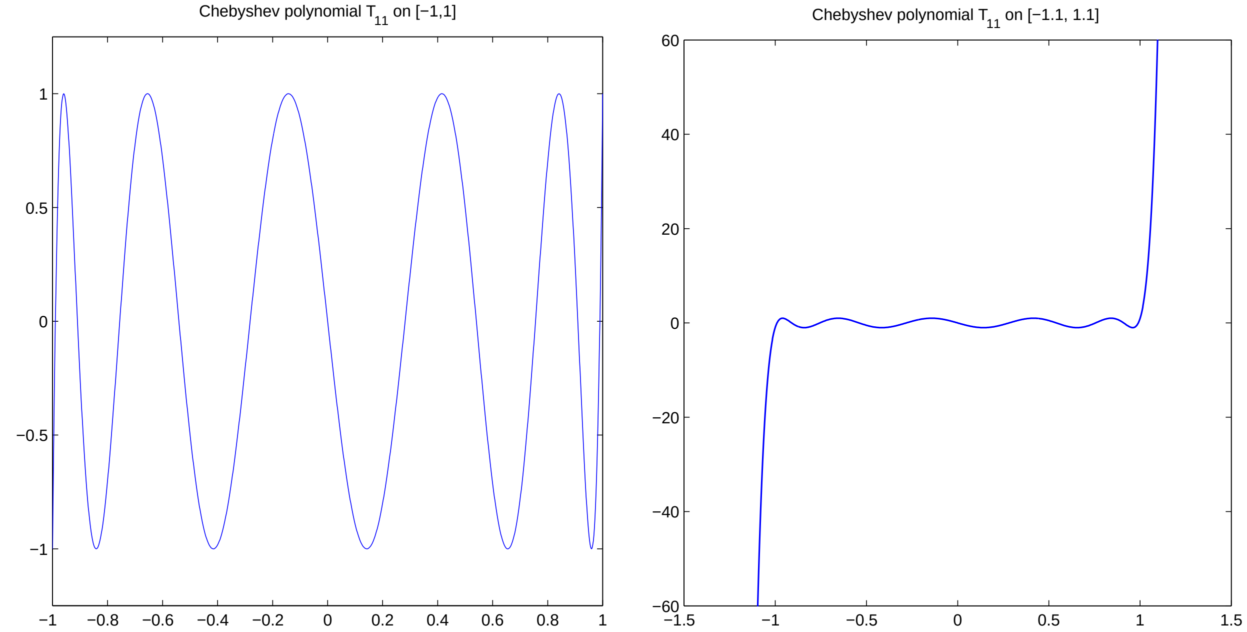


Figure 9.1: The Chebyshev polynomial T_{11} equioscillating on the interval $[-1, 1]$ (at left), and its steep increase of the absolute value outside the interval $[-1, 1]$ (at right).

We capitalize upon this behavior by using the additional substitution $\tau \mapsto \xi := (\tau - \alpha)/\delta$, which maps the given interval $\mathcal{I}$ onto $[-1, 1]$ and by scaling the function so that $p(0) = 1$; see Figure 9.2.

THEOREM 9.4.1. *The optimal solution of the approximation problem* (9.55) *with* $\mathcal{I} :\equiv [\alpha - \delta, \alpha + \delta]$ *is the* ***shifted and scaled Chebyshev polynomial***

$$\boxed{p_n(\tau) = \frac{T_n\left(\frac{\tau-\alpha}{\delta}\right)}{T_n\left(-\frac{\alpha}{\delta}\right)}}\,. \tag{9.57}$$

PROOF. Problem (9.55) is a variation of a classical approximation problem where p_n is required to be monic (*i.e.,* to have leading coefficient 1) instead of having constant coefficient 1 and the interval is $[-1, 1]$. For that problem, the solution is T_n, and the proof is a special case of the one for the equioscillation theorem from the theory of real polynomial uniform (or Chebyshev) approximation.

The polynomial p_n has on $\mathcal{I}$ the maximum and minimum values $\pm 1/T_n\left(-\frac{\alpha}{\delta}\right)$. The maximum is taken at $\lceil \frac{1}{2}(n+1)\rceil$ points and the minimum at interlacing $\lceil \frac{1}{2}n\rceil$ points. Assume there is a polynomial $s \in \mathcal{P}_n$ with $s(0) = 1$, which yields a smaller maximum in (9.55). Then the difference $p_n - s \in \mathcal{P}_n$ will have alternating sign at the $n + 1$ maxima and minima of p_n, so it will have n zeros between these points. Moreover, it has another zero at $\tau = 0$ where $p_n(0) = s(0) = 1$. That makes a total of $n + 1$ zeros, which contradicts to the limit of n zeros for a nonzero polynomial of degree n. □

The recursions (9.56) for the Chebyshev polynomials lead to recursions for the residual polynomials p_n and, thus, for the residuals $\mathbf{r}_n$ and the corresponding iterates $\mathbf{x}_n$. The resulting Krylov space solver

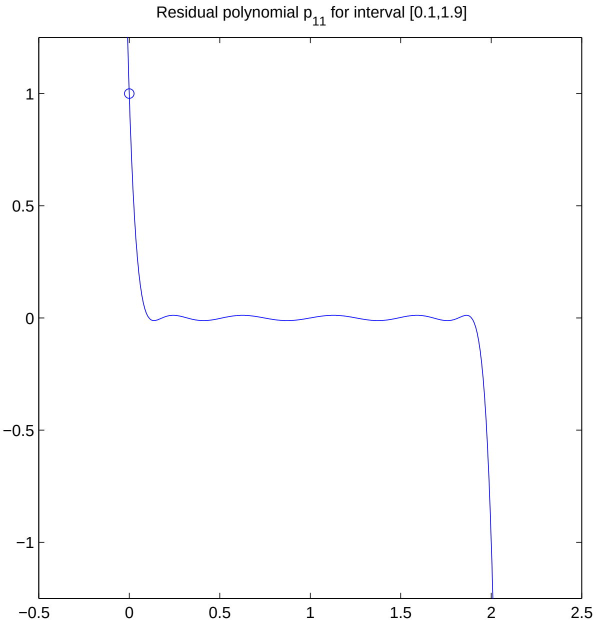


Figure 9.2: The residual polynomial p_{11} of the Chebyshev iteration for the interval $[0.1, 1.9]$ — a shifted and scaled version of T_{11}.

is nowadays called **Chebyshev iteration** [*Tschebyscheff-Iteration*]. In earlier times it was often said to be the **Chebyshev semi-iterative method**. The method was, *e.g.,* investigated by [Rut59], [GV61], and [Var62].

Chebyshev iteration is optimal for the set of matrices $\mathbf{A}$ whose spectrum is confined to the interval $\mathcal{I} :\equiv [\alpha - \delta, \alpha + \delta]$. Of course, for an individual such matrix there are still faster methods. The method can be seen to be asymptotically optimal (for $n \to \infty$) also for the set of matrices $\mathbf{A}$ whose spectrum is confined to an elliptical domain $\mathcal{E}$ with foci $\alpha - \delta$ and $\alpha + \delta$ that does not contain the origin. But, in general, it is not optimal in this case, see [FF90] and [FF91]: exceptions exists even with $\alpha, \delta \in \mathbb{R}$. Moreover, in the case where $\mathbf{A}$ is not real symmetric or Hermitian, the factor $\kappa(\mathbf{U})$ in (9.50) will normally not be 1.

Although our derivation is limited to the case of eigenvalues in a real interval $\mathcal{I}$, we formulate the method here so that the case of complex eigenvalues is included.

Algorithm 9.1 (Chebyshev Iteration)**.** .
For solving $\mathbf{Ax} = \mathbf{b}$ *choose* $\mathbf{x}_0 \in \mathbb{E}^N$ *and let* $\mathbf{r}_0 := \mathbf{b} - \mathbf{Ax}_0$*. Set* $\mathbf{r}_{-1} := \mathbf{x}_{-1} := \mathbf{o}$*.*
Choose the parameters α *and* δ *so that the spectrum of* $\mathbf{A}$ *lies on the interval* $\mathcal{I} :\equiv [\alpha - \delta, \alpha + \delta]$ *or on an elliptical domain* $\mathcal{E}$ *with foci* $\alpha \pm \delta$*, but so that* $0 \notin \mathcal{I}$ *or* $0 \notin \mathcal{E}$*, respectively. Then let* $\eta :\equiv -\alpha/\delta$*,*

$$\beta_{-1} := 0\,, \qquad \beta_0 := \frac{\delta}{2}\,\frac{1}{\eta} = -\frac{\delta^2}{2\alpha}\,, \qquad \gamma_0 := -\alpha\,, \tag{9.58a}$$

and compute, for $n = 0, 1, \ldots$ *until convergence,*

$$\beta_{n-1} :\equiv \frac{\delta}{2}\,\frac{T_{n-1}(\eta)}{T_n(\eta)} := \left(\frac{\delta}{2}\right)^2 \frac{1}{\gamma_{n-1}} \qquad \textit{if} \quad n \geq 2, \tag{9.58b}$$

$$\gamma_n :\equiv \frac{\delta}{2}\,\frac{T_{n+1}(\eta)}{T_n(\eta)} := -(\alpha + \beta_{n-1}) \qquad \textit{if} \quad n \geq 1, \tag{9.58c}$$

$$\mathbf{x}_{n+1} := -\big(\mathbf{r}_n + \mathbf{x}_n\alpha + \mathbf{x}_{n-1}\beta_{n-1}\big)/\gamma_n\,, \tag{9.58d}$$

$$\mathbf{r}_{n+1} := \big(\mathbf{Ar}_n - \mathbf{r}_n\alpha - \mathbf{r}_{n-1}\beta_{n-1}\big)/\gamma_n\,. \tag{9.58e}$$

Note that in the case of real foci, $\eta :\equiv -\alpha/\delta < -1$, so that in (9.58b) and (9.58c) the formula

$T_n(\eta) = (-1)^n \cosh(n \operatorname{arcosh}(-\eta))$ should be used.

For the Chebyshev iteration we can specify a bound for the residual norm reduction and thus estimate the speed of convergence. This requires a little bit more classical analysis; additionally, complex analysis is also helpful for understanding the background of the method.

Let us again assume that $\mathbf{A}$ is Hpd and that its spectrum is known to be in $\mathcal{I} = [\alpha - \delta, \alpha + \delta]$, where $\alpha > \delta > 0$. Moreover, let us set

$$\kappa_{\mathcal{I}} :\equiv \frac{\alpha + \delta}{\alpha - \delta} . \tag{9.59}$$

Note that $\kappa_{\mathcal{I}}$ is a bound for the spectral condition number $\kappa(\mathbf{A})$, and that $\kappa_{\mathcal{I}} = \kappa(\mathbf{A})$ if $\mathbf{I}$ is chosen smallest possible, that is so that $\alpha \pm \delta$ are the smallest and the largest eigenvalues of $\mathbf{A}$. Recall that the two bounds in (9.54) hold. The one in the middle,

$$\|\mathbf{r}_n\| / \|\mathbf{r}_0\| \leq \max_{i=1,\ldots,N} |p_n(\lambda_i)| , \tag{9.60}$$

is sharp in the sense that for any n we can specify a $\mathbf{x}_0$ so that equality holds. In fact, if m is the index for which the maximum is taken in (9.60), and if $\mathbf{u}_m$ is the eigenvector corresponding to λ_m, then we just need to choose $\mathbf{x}_0 := \mathbf{x}_\star - \mathbf{u}_m$. Then $\mathbf{r}_0 = \mathbf{A}(\mathbf{x}_\star - \mathbf{x}_0) = \mathbf{u}_m \lambda_m$ and $\mathbf{r}_n = \mathbf{u}_m \lambda_m p_n(\lambda_m) = \mathbf{r}_0 p_n(\lambda_m)$.

Since $(\tau - \alpha)/\delta \in [-1, 1]$ if $\tau \in \mathcal{I}$ we conclude further that

$$\max_{i=1,\ldots,N} |p_n(\lambda_i)| = |p_n(\lambda_m)| \leq \frac{1}{\left|T_n\left(-\frac{\alpha}{\delta}\right)\right|} = \frac{1}{|T_n(\eta)|} . \tag{9.61}$$

Here, since $\eta < -1$, we have $|T_n(\eta)|^{-1} < 1$, which means that $\|\mathbf{r}_n\| < \|\mathbf{r}_0\|$ if $n > 0$. The bound in (9.61) is sharp in the sense that there are matrices with spectrum contained in $\mathcal{I}$ for which equality holds.

As is easy to verify, $\vartheta :\equiv \exp\bigl(\operatorname{arcosh}(-\eta)\bigr) > 1$ satisfies

$$\frac{1}{2}\left(\vartheta + \frac{1}{\vartheta}\right) = -\eta . \tag{9.62}$$

In terms of ϑ the value $T_n(\eta)$ can be written as

$$T_n(\eta) = \frac{(-1)^n}{2}\left(\vartheta^n + \frac{1}{\vartheta^n}\right) . \tag{9.63}$$

Relation (9.62), which, up to the minus sign, describes the **Joukowski transformation** [Hen74], means that $\vartheta^2 + 2\eta\vartheta + 1 = 0$ and yields

$$\vartheta = -\eta \pm \sqrt{\eta^2 - 1}. \tag{9.64}$$

In terms of $\kappa_{\mathcal{I}}$ we have from (9.59)

$$\eta = -\frac{\kappa_{\mathcal{I}} + 1}{\kappa_{\mathcal{I}} - 1} , \tag{9.65}$$

and, after some manipulation, we find the two reciprocal solutions

$$\vartheta = \frac{\sqrt{\kappa_{\mathcal{I}}} + 1}{\sqrt{\kappa_{\mathcal{I}}} - 1} \qquad \text{or} \qquad \vartheta = \frac{\sqrt{\kappa_{\mathcal{I}}} - 1}{\sqrt{\kappa_{\mathcal{I}}} + 1} , \tag{9.66}$$

which both yield

$$|T_n(\eta)| = \frac{1}{2}\left[\left(\frac{\sqrt{\kappa_{\mathcal{I}}} + 1}{\sqrt{\kappa_{\mathcal{I}}} - 1}\right)^n + \left(\frac{\sqrt{\kappa_{\mathcal{I}}} - 1}{\sqrt{\kappa_{\mathcal{I}}} + 1}\right)^n\right] . \tag{9.67}$$

Actually, only the first solution, the one with $\vartheta > 1$ is consistent with our definition of ϑ based on the principal branch of arcosh. In summary, (9.60), (9.61), and (9.67) yield the following estimate.

THEOREM 9.4.2. *The residual norm reduction of the Chebyshev iteration, when applied to an Hpd system whose condition number is bounded by $\kappa_{\mathcal{I}}$, is bounded according to*

$$\frac{\|\mathbf{r}_n\|}{\|\mathbf{r}_0\|} \leq 2\left[\left(\frac{\sqrt{\kappa_{\mathcal{I}}} + 1}{\sqrt{\kappa_{\mathcal{I}}} - 1}\right)^n + \left(\frac{\sqrt{\kappa_{\mathcal{I}}} - 1}{\sqrt{\kappa_{\mathcal{I}}} + 1}\right)^n\right]^{-1} \leq 2\left(\frac{\sqrt{\kappa_{\mathcal{I}}} - 1}{\sqrt{\kappa_{\mathcal{I}}} + 1}\right)^n . \tag{9.68}$$

The first bound is sharp in the sense that there are matrices $\mathbf{A}$ with spectrum in $\mathcal{I}$ and suitable initial vectors $\mathbf{x}_0$ such that the bound is attained.

The theorem means that, in general, the residuals in the Chebyshev iteration converge linearly. The **asymptotic (root-)convergence factor** is bounded according to

$$\left(\frac{\|\mathbf{r}_n\|}{\|\mathbf{r}_0\|}\right)^{1/n} \leq \frac{\sqrt{\kappa_{\mathcal{I}}}-1}{\sqrt{\kappa_{\mathcal{I}}}+1} = \frac{1}{\vartheta} = \mathrm{e}^{-\operatorname{arcosh}(-\eta)} = |\eta| - \sqrt{\eta^2-1}\,. \tag{9.69}$$

Using some of the formulas given above, it is easy to show that the coefficients β_n and γ_n of the Chebyshev iteration converge as $n \to \infty$, that is $\beta_n \to \beta$ and $\gamma_n \to \gamma$. In fact, from (9.63) we have

$$\frac{T_{n-1}(\eta)}{T_n(\eta)} = -\,\frac{\vartheta^{n-1}+\vartheta^{-(n-1)}}{\vartheta^n+\vartheta^{-n}} \to -\,\frac{1}{\vartheta} \qquad \text{as} \quad n \to \infty\,.$$

Therefore, by (9.58b) and (9.58c), we have

$$\beta_{n-1} \to \beta :\equiv -\,\frac{\delta}{2\vartheta} = -\,\frac{\delta}{2}\left(|\eta| - \sqrt{\eta^2-1}\right)\,, \tag{9.70}$$

$$\gamma_n \to \gamma :\equiv -\,\frac{\delta\vartheta}{2} = -\,\frac{\delta}{2}\left(|\eta| + \sqrt{\eta^2-1}\right)\,, \tag{9.71}$$

and thus

$$\beta+\gamma = -\frac{\delta}{2}\big(\vartheta+\vartheta^{-1}\big) = \delta\eta = -\alpha\,, \tag{9.72}$$

as required by the limit of (9.58c).

Redefining in the recursions (9.58d) and (9.58e) $\gamma_0 := -\alpha$ and $\beta_{n-1} := \beta$, $\gamma_n := \gamma$ if $n > 0$, we obtain another Krylov space solver, which is also asymptotically optimal for the same interval and the same set of confocal ellipses. It is called **second-order Richardson iteration**. In contrast to the Chebyshev iteration it uses a stationary three-term recursion for $\mathbf{r}_n$, that is, the coefficients do not depend on n. Actually, the recursion involves again four terms, but the underlying recursion for the residual polynomials has only three terms if we write $\tau p_n(\tau) - \alpha p_n(\tau)$ as $(\tau-\alpha)p_n(\tau)$.

Like for SOR, the main disadvantage of these two methods is that some knowledge of the spectrum of $\mathbf{A}$ is required, in particular a lower bound for the distance of the smallest eigenvalue from the origin.

9.5 Preconditioning

When applied to large real-world problems Krylov space solvers often converge very slowly — if at all. In practice, Krylov space solvers are therefore nearly always applied with **preconditioning** [*Vorkonditionierung, Präkonditionierung*]. The basic idea behind it is to replace the given linear system $\mathbf{Ax} = \mathbf{b}$ by an equivalent one whose matrix is more suitable for a treatment by the chosen Krylov space method. In particular, it is normally expected to have much better condition. Ideal are matrices whose eigenvalues are clustered around one point except for a few outliers, and such that the cluster is well separated from the origin. Other properties, like the degree of nonnormality, also play a role, since highly nonnormal matrices often cause a delay of the convergence. Minor perturbations of such matrices can cause the spectrum to change much. Note that again, this new matrix needs not be available explicitly.

There are several ways of preconditioning. In the simplest case, called **left preconditioning** [*linke Vorkonditionierung*], the system is just multiplied from the left by some matrix $\mathbf{C}$ that is in some sense an approximation of the inverse of $\mathbf{A}$:

$$\boxed{\underbrace{\mathbf{CA}}_{\widehat{\mathbf{A}}}\,\mathbf{x} = \underbrace{\mathbf{Cb}}_{\widehat{\mathbf{b}}}\,.} \tag{9.73}$$

$\mathbf{C}$ is then called a **left preconditioner** [*linker Vorkonditionierer*] or, more appropriately, an **approximate inverse** [*approximative Inverse*] applied on the left-hand side. Of course, $\mathbf{C}$ should be sparse or specially structured, so that the matrix-vector product $\mathbf{Cy}$ can be calculated quickly for any $\mathbf{y}$.

Often, given is not $\mathbf{C}$ but its inverse $\mathbf{M} :\equiv \mathbf{C}^{-1}$, which is also called **left preconditioner**. In this case, we need to be able to solve $\mathbf{Mz} = \mathbf{y}$ quickly for any $\mathbf{y}$.

An alterative is to substitute $\mathbf{x}$ by $\widehat{\mathbf{x}} :\equiv \mathbf{C}^{-1}\mathbf{x}$, so that $\mathbf{A}\mathbf{x} = \mathbf{b}$ is replaced by

$$\boxed{\underbrace{\mathbf{A}\mathbf{C}}_{\widehat{\mathbf{A}}}\;\underbrace{\mathbf{C}^{-1}\mathbf{x}}_{\widehat{\mathbf{x}}} = \mathbf{b}\,.} \tag{9.74}$$

This is **right preconditioning** [*rechte Vorkonditionierung*]. Here, logically, $\mathbf{C}$ is called a **right preconditioner** [*rechter Vorkonditionierer*] or, an approximate inverse applied on the right-hand side. Again, we may have not $\mathbf{C}$ but its inverse $\mathbf{M}$ available. Note that in the situation of (9.74) we do not need $\mathbf{C}^{-1}$ because we will first obtain $\widehat{\mathbf{x}}$ and so have to compute $\mathbf{x} = \mathbf{C}\widehat{\mathbf{x}}$.

Sometimes it is most appropriate to combine these approaches, and then this is called **split preconditioning** [*gesplittete Vorkonditionierung*].

In general, the effect of preconditioning on the formulation and convergence of a Krylov space solver can be understood as the replacement of the system $\mathbf{A}\mathbf{x} = \mathbf{b}$ by one of the form

$$\boxed{\underbrace{\mathbf{C}_L\mathbf{A}\mathbf{C}_R}_{\widehat{\mathbf{A}}}\;\underbrace{\mathbf{C}_R^{-1}\mathbf{x}}_{\widehat{\mathbf{x}}} = \underbrace{\mathbf{C}_L\mathbf{b}}_{\widehat{\mathbf{b}}}\,,} \tag{9.75}$$

where we allow either $\mathbf{C}_L$ or $\mathbf{C}_R$ to be the identity, or by one of the form

$$\boxed{\underbrace{\mathbf{M}_L^{-1}\mathbf{A}\mathbf{M}_R^{-1}}_{\widehat{\mathbf{A}}}\;\underbrace{\mathbf{M}_R\mathbf{x}}_{\widehat{\mathbf{x}}} = \underbrace{\mathbf{M}_L^{-1}\mathbf{b}}_{\widehat{\mathbf{b}}}\,,} \tag{9.76}$$

where now at most either $\mathbf{M}_L$ or $\mathbf{M}_R$ is the identity. Then the product $\mathbf{C} :\equiv \mathbf{C}_L\,\mathbf{C}_R$ or the product $\mathbf{M} :\equiv \mathbf{M}_R\,\mathbf{M}_L$ is the **split preconditioner** [*gesplitteter Vorkonditionierer*].

The split preconditioning is particularly useful if $\mathbf{A}$ is real symmetric (or Hermitian) and we choose as the right preconditioner the transpose (or, in the complex case, the Hermitian transpose) of the left one, so that $\widehat{\mathbf{A}}$ is still symmetric. In particular, if $\mathbf{M}_R$ and $\mathbf{C}_L$ are lower triangular matrices $\mathbf{L}$ and $\mathbf{K}$, respectively, we have

$$\boxed{\mathbf{M} = \mathbf{L}\,\mathbf{L}^\star\,,} \qquad \boxed{\mathbf{C} = \mathbf{K}\,\mathbf{K}^\star\,,} \tag{9.77}$$

and these can be viewed as the **Cholesky decompositions** [*Cholesky-Zerlegungen*] of $\mathbf{M}$ and $\mathbf{C}$, respectively.

Left preconditioning effects the residuals: it involves the **preconditioned residual vectors**

$$\boxed{\widehat{\mathbf{r}}_n :\equiv \mathbf{C}_L\,\mathbf{r}_n = \mathbf{M}_L^{-1}\,\mathbf{r}_n = \mathbf{M}_L^{-1}\,(\mathbf{b}_n - \mathbf{A}\mathbf{x}_n)\,.} \tag{9.78}$$

On the other hand, right preconditioning effects the error vectors: it involves the **preconditioned error vectors**

$$\boxed{\widehat{\mathbf{x}}_n - \widehat{\mathbf{x}_\star} :\equiv \mathbf{M}_R\,(\mathbf{x}_n - \mathbf{x}_\star) = \mathbf{M}_R\,\mathbf{d}_n = \mathbf{C}_R^{-1}\,(\mathbf{x}_n - \mathbf{x}_\star)\,.} \tag{9.79}$$

Of course, the difficult part in preconditioning is to find the preconditioner, be it $\mathbf{C}_L$, $\mathbf{C}_R$, $\mathbf{M}_L$, $\mathbf{M}_R$, $\mathbf{L}$, or $\mathbf{K}$. Yet another question is how to efficiently combine the Krylov space solver with the preconditioner. Of course, one could just replace $\mathbf{A}$, $\mathbf{b}$, and $\mathbf{x}$ by $\widehat{\mathbf{A}}$, $\widehat{\mathbf{b}}$, and $\widehat{\mathbf{x}}$, but there are sometimes more efficient ways to build a preconditioner into a Krylov space solver. We will treat such cases in Section 9.6.7, where, in various ways, preconditioning is built into conjugate gradient algorithms.

9.6 The Conjugate Gradient Method

9.6.1 Energy Norm Minimization

In many areas of science and technology stable states are characterized by minimum energy. Discretization then leads in the first approximation to the minimization of a quadratic function in several variables,

$$\boxed{\Psi(\mathbf{x}) :\equiv \tfrac{1}{2}\,\mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x} - \mathbf{b}^\mathsf{T}\mathbf{x} + \gamma} \tag{9.80}$$

with an spd matrix $\mathbf{A}$. (We assume real data in Sections 9.6.1–9.6.6.) Such a function Ψ is well known to be convex since its second derivative is the matrix $\mathbf{A}$. So it has a unique minimum, which can be found by setting the first derivative, the gradient, equal to $\mathbf{o}$. The gradient can be seen to be

$$\boxed{\nabla\Psi(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b} = -\mathbf{r}\,,} \tag{9.81}$$

where $\mathbf{r}$ is the residual corresponding to $\mathbf{x}$. Hence,

$$\boxed{\mathbf{x} \text{ minimizer of } \Psi \quad\Longleftrightarrow\quad \nabla\Psi(\mathbf{x}) = \mathbf{o} \quad\Longleftrightarrow\quad \mathbf{A}\mathbf{x} = \mathbf{b}\,.} \tag{9.82}$$

As before, we let $\mathbf{x}_\star$ be the minimizer, *i.e.,* the solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$, and $\mathbf{d} :\equiv \mathbf{x} - \mathbf{x}_\star$ be the error of $\mathbf{x}$. If we define the $\mathbf{A}$-norm as usual by

$$\|\mathbf{y}\|_\mathbf{A} :\equiv \sqrt{\mathbf{y}^\mathsf{T}\mathbf{A}\mathbf{y}}\,, \tag{9.83}$$

it is easily seen that

$$\boxed{\|\mathbf{d}\|_\mathbf{A}^2 = \|\mathbf{x} - \mathbf{x}_\star\|_\mathbf{A}^2 = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_{\mathbf{A}^{-1}}^2 = \|\mathbf{r}\|_{\mathbf{A}^{-1}}^2 = 2\,\Psi(\mathbf{x})} \tag{9.84}$$

if in (9.80)

$$\gamma :\equiv \tfrac{1}{2}\,\mathbf{b}^\mathsf{T}\mathbf{A}^{-1}\mathbf{b}\,. \tag{9.85}$$

Of course, the value of the constant γ has no influence on the solution, so we can assume from now on that we have made this choice; thus, in other words, we are minimizing the $\mathbf{A}$-norm of the error, which is often referred to as the **energy norm** [*Energienorm*].

In summary: If $\mathbf{A}$ is spd, to minimize the quadratic function Ψ means to minimize the energy norm of the error vector of the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$.

Since $\mathbf{A}$ is spd, the level curves $\Psi(\mathbf{x}) = \mathsf{const}$ are ellipses if $N = 2$ and ellipsoids if $N = 3$.

9.6.2 Steepest Descent

The interpretation of the solution of a linear system as the minimizer of convex quadratic function Ψ suggests to find this minimizer by moving down the surface representing Ψ in the direction of steepest descent given by minus the gradient.

However, to follow approximately the so defined curve would require to make many small steps and to evaluate the gradient at every step. We prefer to follow a piecewise straight line with rather long pieces, that is, to make a long step whenever the gradient and thus the direction of steepest descent has been computed. So, in the nth step, we proceed from $(\mathbf{x}_n, \Psi(\mathbf{x}_n))$ on a straight line in the opposite direction of the gradient $\nabla\Psi(\mathbf{x})$, that is, in the direction of $\mathbf{r}_n$. A natural choice is to go to the point with the minimum value of Ψ on that line. In general, finding the minimum value of a functional on a line is called **line search**, but here, since Ψ is quadratic, the length of the step is easy to compute. On the line

$$\omega \mapsto \mathbf{x}_n + \mathbf{r}_n\omega \tag{9.86}$$

the minimum

$$\boxed{\mathbf{x}_{n+1} := \mathbf{x}_n + \mathbf{r}_n\omega_n} \tag{9.87}$$

of $\|\mathbf{x}_{n+1} - \mathbf{x}_\star\|_\mathbf{A}^2 = \|\mathbf{r}_{n+1}\|_{\mathbf{A}^{-1}}^2$ is readily found: $\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{r}_n\omega_n$ implies that $\mathbf{r}_{n+1} = \mathbf{r}_n - \mathbf{A}\mathbf{r}_n\omega_n$, so that

$$\begin{aligned}
\|\mathbf{r}_{n+1}\|_{\mathbf{A}^{-1}}^2 &= \|\mathbf{r}_n - \mathbf{A}\mathbf{r}_n\omega_n\|_{\mathbf{A}^{-1}}^2 \\
&= \|\mathbf{r}_n\|_{\mathbf{A}^{-1}}^2 - 2\,\langle\mathbf{r}_n, \mathbf{A}\mathbf{r}_n\rangle_{\mathbf{A}^{-1}}\;\omega_n + \|\mathbf{A}\mathbf{r}_n\|_{\mathbf{A}^{-1}}^2\,\omega_n^2 \\
&= \|\mathbf{r}_n\|_{\mathbf{A}^{-1}}^2 - 2\,\langle\mathbf{r}_n, \mathbf{r}_n\rangle\;\omega_n + \langle\mathbf{r}_n, \mathbf{A}\mathbf{r}_n\rangle\;\omega_n^2\,.
\end{aligned}$$

As a function of ω_n, this expression has a vanishing derivative (and is thus minimal) when

$$\boxed{\omega_n :\equiv \frac{\langle\mathbf{r}_n, \mathbf{r}_n\rangle}{\langle\mathbf{r}_n, \mathbf{A}\mathbf{r}_n\rangle}\,.} \tag{9.88}$$

This is the **method of steepest descent** [*Methode des steilsten Abstiegs*]. Comparing its formulas (9.87) and (9.88) with (9.11) we see that it differs from the Jacobi iteration only in the (locally optimal) choice of the step length.

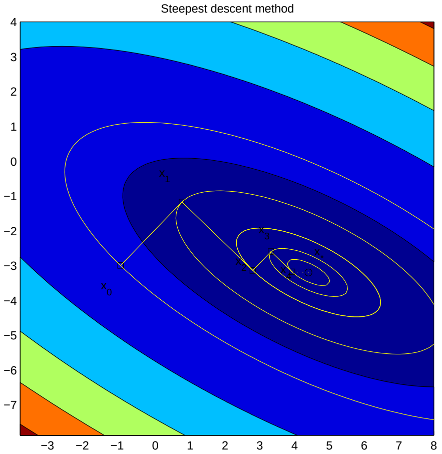


Figure 9.3: The steepest descent method for $N = 2$.

However, this method may converge very slowly even if N is only 2. This is easily seen if we assume that $N = 2$ and that the two (positive) eigenvalues of $\mathbf{A}$ differ very much in size. Then the level curves of Ψ are concentric ellipses with a large axis ratio of λ_1/λ_2, and the search directions are orthogonal to these ellipses. Obviously, there are situations, where it takes many steps to come close to the center of the ellipses; see Figure 9.3. It is also clear that, in general, this method does not converge in at most N steps.

9.6.3 Conjugate Direction Methods

Figure 9.3 manifests where the slow convergence of the steepest descent method comes from: while the direction of steepest descent is optimal on an infinitesimally small scale, it may be far from optimal when we take long steps. For an optimal step in the two-dimensional situation depicted we would have to choose as the second direction one that leads directly to the center of the ellipse. In elementary geometry we have learned that this second direction must be **conjugate** [*konjugiert*] to the first one, which at the minimum $\mathbf{x}_1 := \mathbf{x}_0 + \mathbf{v}_0\omega_0$ is tangential to the ellipse; in other words, we need $\mathbf{v}_1^\mathsf{T}\mathbf{A}\mathbf{v}_0 = 0$, see Figure 9.4.

How does this generalize to N dimensions? We choose nonzero **search directions** [*Suchrichtungen*] or **direction vectors** [*Richtungsvektoren*] $\mathbf{v}_n$ $(n = 0, 1, 2, \dots)$ that are conjugate or $\mathbf{A}$–orthogonal to each other, that is

$$\boxed{\mathbf{v}_n^\mathsf{T}\mathbf{A}\mathbf{v}_k = 0\,, \qquad k = 0, \dots, n-1,} \tag{9.89}$$

and define

$$\boxed{\mathbf{x}_{n+1} := \mathbf{x}_n + \mathbf{v}_n\omega_n\,,} \tag{9.90}$$

so that

$$\boxed{\mathbf{r}_{n+1} = \mathbf{r}_n - \mathbf{A}\mathbf{v}_n\omega_n\,.} \tag{9.91}$$

We choose the step length ω_n again such that the $\mathbf{A}$-norm of the error (that is, the $\mathbf{A}^{-1}$-norm of the residual) is minimized on the line

$$\omega \mapsto \mathbf{x}_n + \mathbf{v}_n\omega\,. \tag{9.92}$$

For any spd matrix $\mathbf{C}$, the minimum of

$$\|\mathbf{r}_n - \mathbf{A}\mathbf{v}_n\omega\|_\mathbf{C}^2 = \|\mathbf{r}_n\|_\mathbf{C}^2 - 2\,\langle \mathbf{r}_n, \mathbf{A}\mathbf{v}_n\rangle_\mathbf{C}\;\omega + \|\mathbf{A}\mathbf{v}_n\|_\mathbf{C}^2\,\omega^2 \tag{9.93}$$

on this line is at

$$\boxed{\omega_n :\equiv \frac{\langle \mathbf{r}_n, \mathbf{A}\mathbf{v}_n \rangle_{\mathbf{C}}}{\|\mathbf{A}\mathbf{v}_n\|^2_{\mathbf{C}}}\,.} \tag{9.94}$$

Here, in order to minimize the $\mathbf{A}^{-1}$-norm of the residual, we choose $\mathbf{C} = \mathbf{A}^{-1}$ and obtain,

$$\boxed{\omega_n :\equiv \frac{\langle \mathbf{r}_n, \mathbf{v}_n \rangle}{\langle \mathbf{v}_n, \mathbf{A}\mathbf{v}_n \rangle}\,.} \tag{9.95}$$

DEFINITION. Any iterative method satisfying (9.89), (9.90), and (9.95) is called a **conjugate direction method** [*Methode der konjugierten Richtungen*]. ▲

By definition, such a method chooses the step length ω_n so that $\mathbf{x}_{n+1}$ is locally optimal on the line (9.92). But does it also yield the best

$$\mathbf{x}_{n+1} \in \mathbf{x}_0 + \mathsf{span}\,\{\mathbf{v}_0, \dots, \mathbf{v}_n\} \tag{9.96}$$

with respect to the $\mathbf{A}$–norm of the error? We will show next that due to choosing conjugate directions this is indeed true.

THEOREM 9.6.1. *For a conjugate direction method the problem of minimizing the energy norm of the error of an approximate solution of the form* (9.96) *decouples into* $n+1$ *one-dimensional minimization problems on the lines* $\omega \mapsto \mathbf{x}_k + \mathbf{v}_k\omega$, $k = 0, 1, \dots, n$.
A conjugate direction method yields after $n+1$ *steps the approximate solution of the form* (9.96) *that minimizes the energy norm* ($\mathbf{A}$*–norm*) *of the error in this affine space.*

PROOF. By (9.84),

$$\begin{aligned}
\Psi(\mathbf{x}_{n+1}) &= \tfrac{1}{2}\,\|\mathbf{x}_{n+1} - \mathbf{x}_\star\|^2_{\mathbf{A}} \\
&= \tfrac{1}{2}\,\|\mathbf{x}_n + \mathbf{v}_n\omega_n - \mathbf{x}_\star\|^2_{\mathbf{A}} \\
&= \tfrac{1}{2}\,\|(\mathbf{x}_n - \mathbf{x}_\star) + \mathbf{v}_n\omega_n\|^2_{\mathbf{A}} \\
&= \Psi(\mathbf{x}_n) + \omega_n \mathbf{v}_n^{\mathsf{T}}\mathbf{A}(\mathbf{x}_n - \mathbf{x}_\star) + \tfrac{1}{2}\,\omega_n^2 \mathbf{v}_n^{\mathsf{T}}\mathbf{A}\mathbf{v}_n \\
&= \Psi(\mathbf{x}_n) - \omega_n \mathbf{v}_n^{\mathsf{T}}\mathbf{r}_n + \tfrac{1}{2}\,\omega_n^2 \mathbf{v}_n^{\mathsf{T}}\mathbf{A}\mathbf{v}_n\,.
\end{aligned}$$

As a consequence of (9.89) we get

$$\begin{aligned}
\mathbf{v}_n^{\mathsf{T}}\mathbf{r}_n &= \mathbf{v}_n^{\mathsf{T}}\left(\mathbf{r}_0 + \mathbf{A}(\mathbf{x}_0 - \mathbf{x}_n)\right) \\
&= \mathbf{v}_n^{\mathsf{T}}\mathbf{r}_0 - \mathbf{v}_n^{\mathsf{T}}\mathbf{A}(\mathbf{v}_0\omega_0 + \cdots + \mathbf{v}_{n-1}\omega_{n-1}) \\
&= \mathbf{v}_n^{\mathsf{T}}\mathbf{r}_0\,.
\end{aligned}$$

Therefore,

$$\Psi(\mathbf{x}_{n+1}) = \Psi(\mathbf{x}_n) - \omega_n \mathbf{v}_n^{\mathsf{T}}\mathbf{r}_0 + \tfrac{1}{2}\,\omega_n^2 \mathbf{v}_n^{\mathsf{T}}\mathbf{A}\mathbf{v}_n\,. \tag{9.97}$$

Here, the last two terms on the right-hand side are independent of $\mathbf{v}_0$, ..., $\mathbf{v}_{n-1}$ and ω_0, ..., ω_{n-1}. So from (9.97) we conclude that the problem of finding the global minimum with respect to the search directions $\mathbf{v}_0, \dots, \mathbf{v}_n$ decouples into the one of minimizing with respect to $\mathbf{v}_0, \dots, \mathbf{v}_{n-1}$ (which by induction we may assume to yield $\mathbf{x}_n$) and the one-dimensional minimization (line search) with respect to $\mathbf{v}_n$. The optimal ω_n in (9.97) is given by $\omega_n = (\mathbf{v}_n^{\mathsf{T}}\mathbf{r}_0)/(\mathbf{v}_n^{\mathsf{T}}\mathbf{A}\mathbf{v}_n)$, but in view of (9.89) this yields the same as (9.95). By induction, finding the global minimum of $\Psi(\mathbf{x}_n)$ decouples further into n one-dimensional minimum problems. □

Conjugate direction methods in this general sense, as well as the special case of the conjugate gradient method treated next have been introduced by [HS52].

9.6.4 The Conjugate Gradient (CG) method

In general, conjugate direction methods are not Krylov space solvers, but if we choose the search directions in a suitable Krylov subspace, then we obtain one. ¿From the relation

$$\mathbf{x}_{n+1} = \mathbf{x}_0 + \mathbf{v}_0\omega_0 + \cdots + \mathbf{v}_n\omega_n \in \mathbf{x}_0 + \mathsf{span}\,\{\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_n\} \tag{9.98}$$

we see that we need to choose search directions $\mathbf{v}_0, \dots, \mathbf{v}_n$ (which by (9.89) are linearly independent) such that they span the Krylov space $\mathcal{K}_{n+1} = \mathcal{K}_{n+1}(\mathbf{A}, \mathbf{r}_0)$:

$$\mathsf{span}\,\{\mathbf{v}_0, \dots, \mathbf{v}_n\} = \mathcal{K}_{n+1}(\mathbf{A}, \mathbf{r}_0)\,, \qquad n = 0, 1, 2, \dots\,. \tag{9.99}$$

Actually, we need primarily that $\mathsf{span}\,\{\mathbf{v}_0, \dots, \mathbf{v}_n\} \subseteq \mathcal{K}_{n+1}$, but the conjugacy condition (9.89) and the requirement that $\mathbf{v}_n \neq 0$ imply that equality must hold.

DEFINITION. The **conjugate gradient (CG) method** [*Methode der konjugierten Gradienten*] is the conjugate direction method with the choice (9.99). ▲

Theorem 9.6.1 gives us the main result on this method:

THEOREM 9.6.2. *The conjugate gradient method yields approximate solutions $\mathbf{x}_n \in \mathbf{x}_0 + \mathcal{K}_n(\mathbf{A}, \mathbf{r}_0)$ that are optimal in the sense that they minimize the energy norm ($\mathbf{A}$–norm) of the error vector for $\mathbf{x}_n$ from this affine space, or, equivalently, it minimizes the $\mathbf{A}^{-1}$–norm of the residual.*

By the two conditions (9.89) and (9.99) the search directions $\mathbf{v}_n$ have to satisfy, they are uniquely determined up to a factor ± 1. But how can we construct them efficiently?

We note first that if $\mathbf{r}_n \in \mathcal{K}_{n+1} \backslash \mathcal{K}_n$, we may normalize $\mathbf{v}_n$ so that

$$\mathbf{v}_n - \mathbf{r}_n \in \mathcal{K}_n\,. \tag{9.100}$$

Before we can proceed we need

LEMMA 9.6.3. *A Krylov space solver based on* (9.90) *with conjugate search directions* $\mathbf{v}_n$ (*satisfying* (9.89)) *and optimal step length* (9.95) *produces mutually orthogonal residuals:*

$$\boxed{\mathbf{r}_n^{\mathsf{T}}\mathbf{r}_k = 0\,, \qquad k = 0, \dots, n-1\,.} \tag{9.101}$$

So, if $\mathbf{r}_0, \dots, \mathbf{r}_{n-1} \neq \mathbf{o}$,

$$\mathsf{span}\,\{\mathbf{r}_0, \dots, \mathbf{r}_{n-1}\} = \mathcal{K}_n(\mathbf{A}, \mathbf{r}_0) \perp \mathbf{r}_n\,. \tag{9.102}$$

PROOF. Let us insert the optimal ω_n of (9.95) into the update formula (9.91) for $\mathbf{r}_n$:

$$\begin{aligned} \mathbf{r}_{n+1} &= \mathbf{r}_n - \mathbf{A}\mathbf{v}_n\omega_n \\ &= \mathbf{r}_n - \mathbf{A}\mathbf{v}_n \frac{\langle \mathbf{r}_n, \mathbf{v}_n \rangle}{\langle \mathbf{v}_n, \mathbf{A}\mathbf{v}_n \rangle}\,. \end{aligned} \tag{9.103}$$

By induction, assuming that $\mathbf{o} \neq \mathbf{r}_k \perp \mathcal{K}_k$ for $k \leq n$, which implies that (9.102) holds, we see using (9.100) and (9.89), respectively, that

$$\langle \mathbf{r}_n, \mathbf{r}_n \rangle = \langle \mathbf{r}_n, \mathbf{v}_n \rangle\;, \qquad \langle \mathbf{r}_n, \mathbf{A}\mathbf{v}_n \rangle = \langle \mathbf{v}_n, \mathbf{A}\mathbf{v}_n \rangle\;. \tag{9.104}$$

Multiplying (9.103) from the left by $\mathbf{r}_n^{\mathsf{T}}$ we conclude that $\mathbf{r}_{n+1} \perp \mathbf{r}_n$. And multiplying it by $\mathbf{r}_k^{\mathsf{T}}$ with $k < n$ shows likewise that $\mathbf{r}_{n+1} \perp \mathbf{r}_k$. So, (9.101) and (9.102) hold with n replaced by $n+1$. □

Note that in the formula (9.95) for ω_n we can replace $\langle \mathbf{r}_n, \mathbf{v}_n \rangle$ by $\langle \mathbf{r}_n, \mathbf{r}_n \rangle$; see (9.104). So,

$$\boxed{\omega_n = \frac{\langle \mathbf{r}_n, \mathbf{r}_n \rangle}{\langle \mathbf{v}_n, \mathbf{A}\mathbf{v}_n \rangle} = \frac{\delta_n}{\delta_n'}\,,} \tag{9.105}$$

where $\delta_n :\equiv \langle \mathbf{r}_n, \mathbf{r}_n \rangle$, $\delta_n' :\equiv \langle \mathbf{v}_n, \mathbf{A}\mathbf{v}_n \rangle$.

In view of the above we can choose

$$\mathbf{v}_{n+1} := \mathbf{r}_{n+1} - \text{linear combination of} \quad \mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n \,.$$

But we may try the simpler ansatz

$$\boxed{\mathbf{v}_{n+1} := \mathbf{r}_{n+1} - \mathbf{v}_n \psi_n \,.} \tag{9.106}$$

Since $\mathbf{A}\mathbf{v}_k \in \mathcal{K}_{n+1}$ for $k < n$, we find from Lemma 9.6.3 that $(\mathbf{A}\mathbf{v}_k)^\mathsf{T} \mathbf{r}_{n+1} = 0$ for $k < n$. Therefore, by multiplying (9.106) from the left by $(\mathbf{A}\mathbf{v}_k)^\mathsf{T}$ with $k < n$ we see that $\langle \mathbf{v}_{n+1}, \mathbf{A}\mathbf{v}_k \rangle = 0 \ (k < n)$ as required. Moreover, we also get $\langle \mathbf{v}_{n+1}, \mathbf{A}\mathbf{v}_n \rangle = 0$ if we choose $\psi_n = \langle \mathbf{r}_{n+1}, \mathbf{A}\mathbf{v}_n \rangle \,/\, \langle \mathbf{v}_n, \mathbf{A}\mathbf{v}_n \rangle$. Substituting in numerator and denominator $\mathbf{A}\mathbf{v}_n = (\mathbf{r}_n - \mathbf{r}_{n+1}) \frac{1}{\omega_n}$ (from the update formula (9.100) for $\mathbf{r}_n$) and making use of $\langle \mathbf{v}_n, \mathbf{r}_n \rangle = \langle \mathbf{r}_n, \mathbf{r}_n \rangle$ we finally obtain

$$\boxed{\psi_n :\equiv \frac{\langle \mathbf{r}_{n+1}, \mathbf{A}\mathbf{v}_n \rangle}{\langle \mathbf{v}_n, \mathbf{A}\mathbf{v}_n \rangle} = - \frac{\langle \mathbf{r}_{n+1}, \mathbf{r}_{n+1} \rangle}{\langle \mathbf{r}_n, \mathbf{r}_n \rangle} = - \frac{\delta_{n+1}}{\delta_n} \,.} \tag{9.107}$$

Putting things together we now get a detailed algorithm for the CG method. We will see that there exist other algorithms using different recursions, but here we get the (standard) **Hestenes-Stiefel algorithm**, which is also called **coupled two-term version** or OMIN **version** of the method. The terminology "coupled two-term" refers to the fact that for both the residuals and the search directions two-term recursions, namely (9.91) and (9.106), are used for updating. The similar recursion for the iterates $\mathbf{x}_n$ follows from the one for the residuals. Again, these recursions actually contain three terms, but only two belong to the same sequence. The name "OMIN" refers to orthogonality and minimality and will be put in a broader context later.

Algorithm 9.2 (OMIN FORM OF THE CG METHOD). .
For solving $\mathbf{A}\mathbf{x} = \mathbf{b}$ *choose an initial approximation* $\mathbf{x}_0$*, and let* $\mathbf{v}_0 := \mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$ *and* $\delta_0 := \|\mathbf{r}_0\|^2$*. Then, for* $n = 0, 1, 2, \ldots,$ *compute*

$$\begin{aligned}
\delta'_n &:= \|\mathbf{v}_n\|^2_{\mathbf{A}} \,, && \text{(9.108a)}\\
\omega_n &:= \delta_n / \delta'_n \,, && \text{(9.108b)}\\
\mathbf{x}_{n+1} &:= \mathbf{x}_n + \mathbf{v}_n \omega_n \,, && \text{(9.108c)}\\
\mathbf{r}_{n+1} &:= \mathbf{r}_n - \mathbf{A}\mathbf{v}_n \omega_n \,, && \text{(9.108d)}\\
\delta_{n+1} &:= \|\mathbf{r}_{n+1}\|^2, && \text{(9.108e)}\\
\psi_n &:= -\delta_{n+1} / \delta_n \,, && \text{(9.108f)}\\
\mathbf{v}_{n+1} &:= \mathbf{r}_{n+1} - \mathbf{v}_n \psi_n \,. && \text{(9.108g)}
\end{aligned}$$

If $\|\mathbf{r}_{n+1}\| \leq \mathrm{tol}$, *the algorithm terminates and* $\mathbf{x}_{n+1}$ *is a sufficiently accurate approximation of the solution.*

In MATLAB: `x = pcg(A,b)` , but there are many options:

```
[X,FLAG,RELRES,ITER,RESVEC] = PCG(A,B,TOL,MAXIT,M1,M2,X0)
```

These options include: multiple right-hand sides, tolerance, maximum number of iterations, preconditioning, stopping flag, residual norm, iteration number, residual history.

Since the CG method produces optimal approximations from the affine spaces $\mathbf{x}_0 + \mathcal{K}_n$ it is clear that it finds the solution in the minimum number of steps, and from Corollary 9.3.5 we know that $\bar{\nu}(\mathbf{r}_0, \mathbf{A})$ steps are needed:

THEOREM 9.6.4. *The conjugate gradient method yields (in exact arithmetic) the solution of* $\mathbf{A}\mathbf{x} = \mathbf{b}$ *in the minimum number* $\bar{\nu}(\mathbf{r}_0, \mathbf{A})$ *of steps.*

However, it is important to point out that this is a theoretical result, which does not hold in practice due to rounding errors. These often have a very strong effect on the method. Nevertheless, in most cases the method still converges, and in fact delivers very small residuals in far less than $\bar{\nu}(\mathbf{r}_0, \mathbf{A})$ steps. We stop when some measure of the error is small enough, and $\|\mathbf{r}_{n+1}\|$ is a simple such measure. However, since the CG method minimizes the energy norm within each Krylov subspace, it might be more appropriate to check the size of $\|\mathbf{d}_{n+1}\|_\mathbf{A} = \|\mathbf{r}_{n+1}\|_{\mathbf{A}^{-1}}$. [HS52] also showed how this can be done efficiently.

Next we want to illustrate how the recursions Krylov space solvers are based on can be described by matrix equations. We note that the recursions (9.108c), (9.108d), and (9.108g) mean that

$$\begin{aligned} \mathbf{v}_n &= -\ (\mathbf{x}_n - \mathbf{x}_{n+1})\,\tfrac{1}{\omega_n}\,, \\ \mathbf{A}\mathbf{v}_n &= (\mathbf{r}_n - \mathbf{r}_{n+1})\,\tfrac{1}{\omega_n}\,, \\ \mathbf{r}_{n+1} &= \mathbf{v}_{n+1} + \mathbf{v}_n \psi_n\,. \end{aligned}$$

If we let

$$\mathbf{R}_m :\equiv \begin{pmatrix} \mathbf{r}_0 & \mathbf{r}_1 & \cdots & \mathbf{r}_{m-1} \end{pmatrix}, \tag{9.109a}$$

$$\mathbf{V}_m :\equiv \begin{pmatrix} \mathbf{v}_0 & \mathbf{v}_1 & \cdots & \mathbf{v}_{m-1} \end{pmatrix}, \tag{9.109b}$$

$$\mathbf{X}_m :\equiv \begin{pmatrix} \mathbf{x}_0 & \mathbf{x}_1 & \cdots & \mathbf{x}_{m-1} \end{pmatrix}, \tag{9.109c}$$

and assume $m > n$, we can write them as

$$\mathbf{v}_n = -\underbrace{\begin{pmatrix} \mathbf{x}_0 & \cdots & \mathbf{x}_{n-1} & \mathbf{x}_n & \mathbf{x}_{n+1} & \mathbf{x}_{n+2} & \ldots & \mathbf{x}_m \end{pmatrix}}_{\mathbf{X}_{m+1}} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \omega_n^{-1} \\ -\omega_n^{-1} \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

$$\mathbf{A}\mathbf{v}_n = \underbrace{\begin{pmatrix} \mathbf{r}_0 & \cdots & \mathbf{r}_{n-1} & \mathbf{r}_n & \mathbf{r}_{n+1} & \mathbf{r}_{n+2} & \ldots & \mathbf{r}_m \end{pmatrix}}_{\mathbf{R}_{m+1}} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \omega_n^{-1} \\ -\omega_n^{-1} \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

and

$$\mathbf{r}_{n+1} = \underbrace{\begin{pmatrix} \mathbf{v}_0 & \cdots & \mathbf{v}_{n-1} & \mathbf{v}_n & \mathbf{v}_{n+1} & \mathbf{v}_{n+2} & \ldots & \mathbf{v}_m \end{pmatrix}}_{\mathbf{V}_{m+1}} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \psi_n \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

Let us further introduce the matrices

$$\mathbf{U}_m :\equiv \begin{pmatrix} 1 & \psi_0 & & & \\ & 1 & \psi_1 & & \\ & & 1 & \ddots & \\ & & & \ddots & \psi_{m-2} \\ & & & & 1 \end{pmatrix}, \tag{9.110}$$

$$\underline{\mathbf{L}}_m^\circ :\equiv \begin{pmatrix} \omega_0^{-1} & & & & \\ -\omega_0^{-1} & \omega_1^{-1} & & & \\ & -\omega_1^{-1} & \omega_2^{-1} & & \\ & & \ddots & \ddots & \\ & & & -\omega_{m-2}^{-1} & \omega_{m-1}^{-1} \\ & & & & -\omega_{m-1}^{-1} \end{pmatrix}, \tag{9.111}$$

$$\underline{\mathbf{E}}_m :\equiv \begin{pmatrix} 1 & & & & \\ -1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & \ddots & \\ & & & -1 & 1 \\ & & & & -1 \end{pmatrix}, \tag{9.112}$$

and

$$\mathbf{D}_{\omega;\, m} :\equiv \mathsf{diag}\,\{\omega_0, \omega_1, \ldots, \omega_{m-1}\}\,, \tag{9.113}$$

so that

$$\underline{\mathbf{L}}_m^\circ = \underline{\mathbf{E}}_m\, \mathbf{D}_{\omega;\, m}^{-1}\,. \tag{9.114}$$

$\underline{\mathbf{L}}_m^\circ$ and $\underline{\mathbf{E}}_m$ are "extended" lower bidiagonal matrices of the size $(m+1)\times m$ whose column sums vanish[3].

In terms of all these matrices the coupled two-term CG recursions (9.108d) for $n = -1, \ldots, m-2$ and (9.108g) for $n = 0, \ldots, m-1$ can then be summarized as

$$\boxed{\mathbf{R}_m = \mathbf{V}_m \mathbf{U}_m\,, \qquad \mathbf{A}\mathbf{V}_m = \mathbf{R}_{m+1}\underline{\mathbf{L}}_m^\circ \quad (m \le \bar{\nu})\,,} \tag{9.115}$$

and the additional recursion (9.108c) for the iterates turns into

$$\boxed{\mathbf{V}_m = -\,\mathbf{X}_{m+1}\underline{\mathbf{L}}_m^\circ \quad (m \le \bar{\nu})\,.} \tag{9.116}$$

Eliminating the direction vectors contained in $\mathbf{V}_m$ yields

$$\boxed{\mathbf{A}\mathbf{R}_m = \mathbf{R}_{m+1}\underline{\mathbf{T}}_m^\circ\,, \qquad \mathbf{R}_m = -\,\mathbf{X}_{m+1}\underline{\mathbf{T}}_m^\circ \quad (m \le \bar{\nu})\,,} \tag{9.117}$$

where

$$\boxed{\underline{\mathbf{T}}_m^\circ :\equiv \underline{\mathbf{L}}_m^\circ \mathbf{U}_m} \tag{9.118}$$

is an 'extended' tridiagonal matrix:

$$\underline{\mathbf{T}}_m^\circ \equiv: \left(\begin{array}{c} \mathbf{T}_m^\circ \\ \hline \gamma_{m-1}\mathbf{l}_m^\mathsf{T} \end{array}\right) \equiv: \begin{pmatrix} \alpha_0 & \beta_0 & & & \\ \gamma_0 & \alpha_1 & \beta_1 & & \\ & \gamma_1 & \alpha_2 & \ddots & \\ & & \ddots & \ddots & \beta_{m-2} \\ & & & \gamma_{m-2} & \alpha_{m-1} \\ & & & & \gamma_{m-1} \end{pmatrix}, \tag{9.119}$$

where $\mathbf{l}_m^\mathsf{T}$ is the last row of the unit matrix of size m.

The characteristic property of a matrix $\mathbf{Z}$ with column sums 0 is that $\mathbf{e}^\mathsf{T}\mathbf{Z} = \mathbf{o}^\mathsf{T}$ for $\mathbf{e} :\equiv \begin{pmatrix} 1 & 1 & \ldots 1 \end{pmatrix}^\mathsf{T}$ of appropriate size. So, $\mathbf{e}^\mathsf{T}\underline{\mathbf{L}}_m^\circ = \mathbf{o}^\mathsf{T}$, and therefore also

$$\mathbf{e}^\mathsf{T}\underline{\mathbf{T}}_m^\circ = \mathbf{e}^\mathsf{T}\underline{\mathbf{L}}_m^\circ \mathbf{U}_m = \mathbf{o}^\mathsf{T}\,. \tag{9.120}$$

Thus, $\underline{\mathbf{T}}_m^\circ$ has column sums zero too: if we let $\beta_{-1} :\equiv 0$, then

$$\boxed{\gamma_n = -\alpha_n - \beta_{n-1}\,, \qquad n = 0, 1, \ldots, m-1 \le \bar{\nu} - 1\,.} \tag{9.121}$$

[3] By underlining $\mathbf{E}_m$ we want to indicate that we augment this matrix by an additional row. We suggest reading $\underline{\mathbf{E}}_m$ as "E sub m extended". The same notation will be used on other occasions.

According to (9.117) the CG residuals and iterates satisfy three-term recursions:

$$\mathbf{r}_{n+1}\gamma_n = (\mathbf{A}\mathbf{r}_n - \mathbf{r}_n\alpha_n - \mathbf{r}_{n-1}\beta_{n-1})\,, \tag{9.122a}$$

$$\mathbf{x}_{n+1}\gamma_n = -(\mathbf{r}_n + \mathbf{x}_n\alpha_n + \mathbf{x}_{n-1}\beta_{n-1})\,, \tag{9.122b}$$

but what remains to find are formulas for α_n and β_{n-1}. In Lemma 9.6.3 we have seen that the residuals are orthogonal to each other, and, when multiplying (9.122a) from the left by $\mathbf{r}_n^{\mathsf{T}}$ and $\mathbf{r}_{n-1}^{\mathsf{T}}$ this property gives

$$\alpha_n = \frac{\mathbf{r}_n^{\mathsf{T}}\mathbf{A}\mathbf{r}_n}{\mathbf{r}_n^{\mathsf{T}}\mathbf{r}_n}\,, \qquad \beta_{n-1} = \frac{\mathbf{r}_{n-1}^{\mathsf{T}}\mathbf{A}\mathbf{r}_n}{\mathbf{r}_{n-1}^{\mathsf{T}}\mathbf{r}_{n-1}}\,.$$

Here, since $\mathbf{A}$ is symmetric and since the three-term recursion and the orthogonality property of Lemma 9.6.3 hold,

$$\begin{aligned}\mathbf{r}_{n-1}^{\mathsf{T}}\mathbf{A}\mathbf{r}_n &= (\mathbf{A}\mathbf{r}_{n-1})^{\mathsf{T}}\mathbf{r}_n \\ &= (\mathbf{r}_n\gamma_{n-1} + \mathbf{r}_{n-1}\alpha_{n-1} + \mathbf{r}_{n-2}\beta_{n-2})^{\mathsf{T}}\mathbf{r}_n \\ &= \mathbf{r}_n^{\mathsf{T}}\mathbf{r}_n\gamma_{n-1}\,,\end{aligned}$$

so, we only need to compute two inner products, $\langle \mathbf{r}_n, \mathbf{A}\mathbf{r}_n\rangle$ and $\langle \mathbf{r}_n, \mathbf{r}_n\rangle$ per step since $\langle \mathbf{r}_{n-1}, \mathbf{r}_{n-1}\rangle$ and γ_{n-1} will be known from the previous step, while γ_n is computed from (9.121).

Summarizing everything we obtain a new algorithm for the CG method, which is called **three-term version** or ORES **version** of the method, the latter because it makes direct use of the orthogonality of the residuals.

Algorithm 9.3 (ORES FORM OF THE CG METHOD)**. .**
For solving $\mathbf{A}\mathbf{x} = \mathbf{b}$ *with an Hpd matrix* $\mathbf{A}$*, choose an initial approximation* $\mathbf{x}_0$*, and let* $\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$*,* $\beta_{-1} := 0$*, and* $\delta_0 := \|\mathbf{r}_0\|^2$*. Then, for* $n = 0, 1, \ldots,$ *compute*

$$\alpha_n := \|\mathbf{r}_n\|_{\mathbf{A}}^2 \,/\, \delta_n\,, \tag{9.123a}$$

$$\beta_{n-1} := \gamma_{n-1}\delta_n/\delta_{n-1} \qquad (\textit{if} \quad n > 0), \tag{9.123b}$$

$$\gamma_n := -\alpha_n - \beta_{n-1}\,, \tag{9.123c}$$

$$\mathbf{x}_{n+1} := -(\mathbf{r}_n + \mathbf{x}_n\alpha_n + \mathbf{x}_{n-1}\beta_{n-1})/\gamma_n\,, \tag{9.123d}$$

$$\mathbf{r}_{n+1} := (\mathbf{A}\mathbf{r}_n - \mathbf{r}_n\alpha_n - \mathbf{r}_{n-1}\beta_{n-1})/\gamma_n\,, \tag{9.123e}$$

$$\delta_{n+1} := \|\mathbf{r}_{n+1}\|^2\,. \tag{9.123f}$$

If $\|\mathbf{r}_{n+1}\| \le \mathrm{tol}$*, the algorithm terminates and* $\mathbf{x}_{n+1}$ *is a sufficiently accurate approximation of the solution.*

Of course, $\mathbf{A}\mathbf{r}_n$, which is needed in (9.123a) and (9.123e), is computed only once and stored in a temporary vector.

There is yet another version of the CG method. Instead of $\mathbf{V}_m$ one can eliminate $\mathbf{R}_m$ in (9.115) to obtain

$$\boxed{\mathbf{A}\mathbf{V}_m = \mathbf{V}_{m+1}\underline{\mathbf{T}}'_m \quad (m \le \bar{\nu})\,,} \tag{9.124}$$

where

$$\boxed{\underline{\mathbf{T}}'_m :\equiv \mathbf{U}_{m+1}\underline{\mathbf{L}}^{\circ}_m} \tag{9.125}$$

is again an ‘extended’ tridiagonal matrix. Here, (9.124) describes a 3-term recursion for the direction vectors:

$$\mathbf{v}_{n+1}\gamma'_n = \mathbf{A}\mathbf{v}_n - \mathbf{v}_n\alpha'_n - \mathbf{v}_{n-1}\beta'_{n-1}\,, \tag{9.126}$$

where we are free to choose $\gamma'_n \neq 0$, while α'_n and β'_{n-1} have to be determined such that

$$\mathbf{v}_n^{\mathsf{T}}\mathbf{A}\mathbf{v}_{n+1} = 0\,, \qquad \mathbf{v}_{n-1}^{\mathsf{T}}\mathbf{A}\mathbf{v}_{n+1} = 0\,. \tag{9.127}$$

It is easy to verify that the search direction are then all conjugate to each other, as required by (9.89). In general, $\underline{\mathbf{T}}'_m$ does not have column sums zero.

To update the residuals and iterates we can use (9.108d) and (9.108c) from the OMIN version. Altogether we obtain yet another algorithm, which is called ODIR version of the CG method, as it makes explicit usage of the $\mathbf{A}$–orthogonality of the search directions.

Algorithm 9.4 (ODIR FORM OF THE CG METHOD). .
For solving $\mathbf{A}\mathbf{x} = \mathbf{b}$*, choose an initial approximation* $\mathbf{x}_0$*, and let* $\mathbf{v}_0 := \mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$, $\beta'_{-1} := 0$.
Then, for $n = 0, 1, \dots,$ *compute*

$$\begin{aligned}
\delta'_n &:= \|\mathbf{v}_n\|^2_{\mathbf{A}}\,, && \text{(9.128a)}\\
\omega'_n &:= \langle \mathbf{v}_n, \mathbf{r}_n \rangle / \delta'_n\,, && \text{(9.128b)}\\
\mathbf{x}_{n+1} &:= \mathbf{x}_n + \mathbf{v}_n \omega'_n\,, && \text{(9.128c)}\\
\mathbf{r}_{n+1} &:= \mathbf{r}_n - \mathbf{A}\mathbf{v}_n \omega'_n\,, && \text{(9.128d)}\\
\alpha'_n &:= \|\mathbf{A}\mathbf{v}_n\|^2 / \delta'_n\,, && \text{(9.128e)}\\
\beta'_{n-1} &:= \gamma'_{n-1}\delta'_n/\delta'_{n-1} \qquad (\textit{if} \quad n > 0), && \text{(9.128f)}\\
\mathbf{v}_{n+1} &:= (\mathbf{A}\mathbf{v}_n - \mathbf{v}_n\alpha'_n - \mathbf{v}_{n-1}\beta'_{n-1})/\gamma'_n\,, && \text{(9.128g)}
\end{aligned}$$

where $\gamma'_n \neq 0$ *can be chosen* (*e.g., to normalize* $\mathbf{v}_{n+1}$).
If $\|\mathbf{r}_{n+1}\| \leq \mathrm{tol}$*, the algorithm terminates and* $\mathbf{x}_{n+1}$ *is a sufficiently accurate approximation of the solution.*

REMARK. Both in (9.123e) and (9.128g) it is preferable to use the modified Gram-Schmidt algorithm, which also means to modify the formulas (9.123a) for α_n and (9.128e) for α'_n. But, since there are only two terms to subtract, the difference will be marginal. ▼

REMARK. Each step in the sequence

$$\underline{\mathbf{T}}^{\circ}_m \quad \rightsquigarrow \quad (\,\underline{\mathbf{L}}^{\circ}_m,\ \mathbf{U}_m\,) \quad \rightsquigarrow \quad \underline{\mathbf{T}}'_m \rightsquigarrow \quad (\,\underline{\mathbf{L}}'_m,\ \mathbf{U}'_m\,) \tag{9.129}$$

is essentially half a step of Rutihauser's quotient-difference (qd) algorithm [Rut57], except that here the matrices are not square. ▼

9.6.5 The Conjugate Residual (CR) Method

We can replace the aim of minimizing the energy norm considered in Section 9.6.1 by the aim of minimizing the 2-norm of the residual. Let us still assume that $\mathbf{A}$ is symmetric positive definite (spd), so that this is the same as minimizing the $\mathbf{A}^2$–norm of the error:

$$\boxed{\|\mathbf{x} - \mathbf{x}_\star\|^2_{\mathbf{A}^2} = \|\mathbf{d}\|^2_{\mathbf{A}^2} = \|\mathbf{r}\|^2 = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2 = 2\,\widehat{\Psi}(\mathbf{x})} \tag{9.130}$$

if

$$\boxed{\widehat{\Psi}(\mathbf{x}) :\equiv \tfrac{1}{2}\,\mathbf{x}^{\mathsf{T}}\mathbf{A}^2\mathbf{x} - \mathbf{b}^{\mathsf{T}}\mathbf{A}\mathbf{x} + \tfrac{1}{2}\,\mathbf{b}^{\mathsf{T}}\mathbf{b}\,.} \tag{9.131}$$

Now we have

$$\boxed{\nabla\widehat{\Psi}(\mathbf{x}) = \mathbf{A}^2\mathbf{x} - \mathbf{A}\mathbf{b} = -\mathbf{A}\mathbf{r}} \tag{9.132}$$

and

$$\boxed{\mathbf{x} \text{ minimizer of } \widehat{\Psi} \quad \Longleftrightarrow \quad \nabla\widehat{\Psi}(\mathbf{x}) = \mathbf{o} \quad \Longleftrightarrow \quad \mathbf{A}\mathbf{r} = \mathbf{o}\,.} \tag{9.133}$$

It should be no surprise that we can develop a variation of the conjugate gradient method adapted to this error norm by replacing in the derivations of Sections 9.6.3 and 9.6.4 the occurrences of the $\mathbf{A}$–inner product by those of the $\mathbf{A}^2$–inner product, and the occurrences of the standard inner product by those of the $\mathbf{A}$–inner product. In particular, the search directions are chosen from a growing sequence of Krylov subspaces as in (9.99), but are now $\mathbf{A}^2$–orthogonal:

$$\boxed{\mathbf{v}_n^{\mathsf{T}}\mathbf{A}^2\mathbf{v}_k = 0\,, \qquad k = 0, \dots, n-1\,.} \tag{9.134}$$

The new iterates are again found by line search,

$$\boxed{\mathbf{x}_{n+1} :\equiv \mathbf{x}_n + \mathbf{v}_n\omega_n \,,} \tag{9.135}$$

where now the step length ω_n is chosen such that the 2-norm of the residual is minimized on the line $\omega \mapsto \mathbf{x}_n + \mathbf{v}_n\omega$. In (9.94) this requires that $\mathbf{C} = \mathbf{I}$, hence

$$\boxed{\omega_n :\equiv \frac{\langle \mathbf{r}_n, \mathbf{A}\mathbf{v}_n\rangle}{\|\mathbf{A}\mathbf{v}_n\|^2} \,.} \tag{9.136}$$

DEFINITION. The Krylov space solver satisfying (9.99), (9.134), (9.135), and (9.136) is called the **conjugate residual (CR) method** [*Methode der konjugierten Residuen*]. ▲

The CR method comes again in three basic version, OMIN, ORES, and ODIR, analogous to those for CG. A straightforward adaptation leads to two matrix-vector products (MVs) per step, but it is possible to replace one by a recursion. (This causes additional roundoff error propagation, however.) Again some of the formulas for the coefficients can be simplified.

Below is an OMIN version that requires only one MV per iteration, namely for computing $\mathbf{A}\mathbf{r}_{n+1}$, which is needed in (9.137e) and is then also used in (9.137h). The algorithm requires to store the actual $\mathbf{w}_n :\equiv \mathbf{A}\mathbf{v}_n$, but this is only one extra vector in addition to the three that are needed in the OMIN version: $\mathbf{x}_n$, $\mathbf{r}_n$, and $\mathbf{v}_n$. The ORES version also needs four: $\mathbf{x}_n$, $\mathbf{x}_{n-1}$, $\mathbf{r}_n$, and $\mathbf{r}_{n-1}$.

Note that $\mathbf{w}_n = \mathbf{o}$ implies $n = \bar{\nu}$, hence, $\delta'_n :\equiv \|\mathbf{w}_n\|^2$ cannot vanish before the Krylov space is exhausted and we have found the solution of the linear system.

Algorithm 9.5 (OMIN FORM OF THE CR METHOD). .
For solving $\mathbf{A}\mathbf{x} = \mathbf{b}$ *choose an initial approximation* $\mathbf{x}_0$, *and let* $\mathbf{v}_0 := \mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$, $\mathbf{w}_0 := \mathbf{A}\mathbf{v}_0$, *and* $\delta_0 := \|\mathbf{r}_0\|^2_{\mathbf{A}}$. *Then, for* $n = 0, 1, 2, \ldots$, *compute*

$$\delta'_n := \|\mathbf{w}_n\|^2 \,, \tag{9.137a}$$

$$\omega_n := \delta_n/\delta'_n \,, \tag{9.137b}$$

$$\mathbf{x}_{n+1} := \mathbf{x}_n + \mathbf{v}_n\omega_n \,, \tag{9.137c}$$

$$\mathbf{r}_{n+1} := \mathbf{r}_n - \mathbf{w}_n\omega_n \,, \tag{9.137d}$$

$$\delta_{n+1} := \|\mathbf{r}_{n+1}\|^2_{\mathbf{A}} , \tag{9.137e}$$

$$\psi_n := -\delta_{n+1}/\delta_n \,, \tag{9.137f}$$

$$\mathbf{v}_{n+1} := \mathbf{r}_{n+1} - \mathbf{v}_n\psi_n \,, \tag{9.137g}$$

$$\mathbf{w}_{n+1} := \mathbf{A}\mathbf{r}_{n+1} - \mathbf{w}_n\psi_n \,. \tag{9.137h}$$

If $\|\mathbf{r}_{n+1}\| \leq \mathrm{tol}$, *the algorithm terminates and* $\mathbf{x}_{n+1}$ *is a sufficiently accurate approximation of the solution.*

The conjugate residual method was introduces by [Sti55], who gave formulas for an ORES version, but for one that differs from the analogue of our ORES version of CG. Stiefel used update formulas for the differences of two successive approximations and two successive residuals, respectively.

As in the various forms of the CG method, here there are again several theoretically equivalent ways to compute the quantities δ_n and δ'_n as inner products.

An interesting aspect of the ODIR version is that it also applies to indefinite symmetric (or Hermitian) matrices. In fact, $\mathbf{A}^2$ is still spd (or Hpd), and thus the quantities $\|\mathbf{v}_n\|_{\mathbf{A}^2}$ that are analogous to δ'_n in Algorithm 9.4 cannot vanish as long as $\mathbf{v}_n \neq \mathbf{o}$.

Nevertheless, the ODIR version of the CR method is no longer used much, since the mathematically equivalent MINRES algorithm has been said to be more stable. Only recently, it was pointed out by [SvM00] that with MINRES the attainable accuracy of the approximate solutions may be rather low for ill-conditioned matrices.

We will later treat a generalization of the CR method to nonsymmetric and non-Hermitian matrices.

9.6.6 A Bound for the Convergence

For the moment, let us assume that $\mathbf{A}$ is symmetric and consider the $\mathbf{C}$-norm of the error vector $\mathbf{d}_n$, where $\mathbf{C} = \mathbf{A}^\ell$ is a positive power of $\mathbf{A}$. If $\mathbf{A}$ is indefinite, we need an even power to make $\mathbf{C}$ Hpd. Then, since $\mathbf{A} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^{-1}$ with $\mathbf{U}$ orthogonal and $\boldsymbol{\Lambda}$ diagonal, and since, for any Krylov space solver, $\mathbf{d}_n = p_n(\mathbf{A})\mathbf{d}_0$ according to (9.49), we have

$$\begin{aligned}\|\mathbf{d}_n\|_{\mathbf{C}} &= \|\mathbf{A}^{\ell/2} p_n(\mathbf{A})\mathbf{d}_0\| = \|\mathbf{U} p_n(\boldsymbol{\Lambda})\mathbf{U}^{-1}\mathbf{A}^{\ell/2}\mathbf{d}_0\| \\ &\leq \|p_n(\boldsymbol{\Lambda})\| \, \|\mathbf{A}^{\ell/2}\mathbf{d}_0\| = \|p_n(\boldsymbol{\Lambda})\| \, \|\mathbf{d}_0\|_{\mathbf{C}} \\ &= \max_{i=1,\dots,N} |p_n(\lambda_i)| \, \|\mathbf{d}_0\|_{\mathbf{C}} \,. \end{aligned} \tag{9.138}$$

Further, we assume that the spectrum of $\mathbf{A}$ lies on the interval $\mathcal{I} = [\alpha - \delta, \alpha + \delta]$ of the positive real axis, and we consider the Chebyshev iteration for that interval. We denote its nth error vector and residual polynomial by $\mathbf{d}_n^{\text{Cheb}}$ and p_n^{Cheb}, respectively, while $\mathbf{d}_n$ is assumed to be the error for a Krylov space solver that is optimal in the $\|.\|_{\mathbf{C}}$–norm of the error when applied to $\mathbf{A}\mathbf{x} = \mathbf{b}$ with a matrix whose spectrum lies on $\mathcal{I}$. Then we have

$$\frac{\|\mathbf{d}_n\|_{\mathbf{C}}}{\|\mathbf{d}_0\|_{\mathbf{C}}} \leq \frac{\|\mathbf{d}_n^{\text{Cheb}}\|_{\mathbf{C}}}{\|\mathbf{d}_0\|_{\mathbf{C}}} \leq \max_{i=1,\dots,N} |p_n^{\text{Cheb}}(\lambda_i)| \leq \min_{\substack{p \in \mathcal{P}_n \\ p(0)=1}} \max_{\tau \in \mathcal{I}} |p(\tau)| \,. \tag{9.139}$$

Note that in this estimate the dependence of the residual polynomial on the initial residual and on the norm used is lost: we estimate the error of any optimal method by the error of the Chebyshev method, and in the estimate of the latter, the norm used has no effect on the bound for the error reduction.

In particular, the estimate is applicable for the CG method, which is optimal in the $\mathbf{A}$–norm of the error, and for the CR method, which is optimal in the $\mathbf{A}^2$–norm of the error, that is, in the 2–norm of the residual.

The bound on the right-hand side of (9.139) was estimated in the derivation of Theorem 9.4.2, and we can conclude that the bounds obtained in that theorem hold here too.

THEOREM 9.6.5. *The residual norm reduction of the* CR *method and the energy norm reduction of the* CG *method, when applied to an spd system whose condition number is bounded by $\kappa_{\mathcal{I}}$, have the bounds given in* (9.68) *for the residual norm reduction of the Chebyshev iteration.*

9.6.7 Preconditioned CG Algorithms

The conjugate gradient method is a typical example where preconditioning must be done so that the symmetry of $\mathbf{A}$ is not lost. So it seems to be a case for split preconditioning with $\mathbf{C}_L = \mathbf{C}_R^{\mathsf{T}}$ or $\mathbf{M}_L = \mathbf{M}_R^{\mathsf{T}}$, respectively. In principle, we could then apply any CG algorithm with $\mathbf{A}$, $\mathbf{b}$, and $\mathbf{x}$ replaced by $\widehat{\mathbf{A}}$, $\widehat{\mathbf{b}}$, and $\widehat{\mathbf{x}}$ to get a **preconditioned CG algorithm**, or, as often referred to, a **PCG algorithm**. However, we will see that there are more refined ways to incorporate the preconditioning into the algorithms. There are even ways to use one-sided preconditioning destroying the symmetry of $\mathbf{A}$ if we combine it with a variation of CG using an alternative inner product.

We start with the details of the integration of symmetrically split preconditioning. For simplicity, we want to write the preconditioner as in (9.77): $\mathbf{M} = \mathbf{L}\mathbf{L}^{\mathsf{T}}$ or $\mathbf{C} = \mathbf{K}\mathbf{K}^{\mathsf{T}}$, even if $\mathbf{L}$ or $\mathbf{K}$, respectively, is not lower triangular. Although the former notation is more common, we prefer the latter, because we can avoid the usage of $\mathbf{L}^{-1}$ in the formulation of the preconditioned algorithm. If $\mathbf{L}$ is known instead of $\mathbf{K}$, we will just have to solve a linear system — assumed to be easy due to the structure of $\mathbf{L}$ — whenever a matrix-vector product with $\mathbf{K}$ or $\mathbf{K}^{\mathsf{T}}$ comes up.

If $\mathbf{A}$ is spd and $\mathbf{K}$ is nonsingular, then $\widehat{\mathbf{A}} :\equiv \mathbf{K}\mathbf{A}\mathbf{K}^{\mathsf{T}}$ is spd too, so we can replace the spd system $\mathbf{A}\mathbf{x} = \mathbf{b}$ by the spd system

$$\boxed{\underbrace{\mathbf{K}\mathbf{A}\mathbf{K}^{\mathsf{T}}}_{\widehat{\mathbf{A}}} \, \widehat{\mathbf{x}} = \underbrace{\mathbf{K}\mathbf{b}}_{\widehat{\mathbf{b}}} \,, \qquad \mathbf{x} = \mathbf{K}^{\mathsf{T}}\widehat{\mathbf{x}} \,.} \tag{9.140}$$

In the following OMIN form of CG with split preconditioning $\widehat{\mathbf{x}}_n$ does not appear. The iterates $\mathbf{x}_n$ are computed directly.

Algorithm 9.6 (PCG–OMIN WITH SPLIT PRECONDITIONING)**.** *For solving* $\mathbf{Ax} = \mathbf{b}$ *with spd* $\mathbf{A}$ *in the split preconditioned form* (9.140) *choose an initial approximation* $\mathbf{x}_0$*, and let* $\mathbf{r}_0 := \mathbf{b} - \mathbf{Ax}_0$, $\widehat{\mathbf{r}}_0 := \mathbf{Kr}_0$, $\mathbf{v}_0 := \mathbf{K}^{\mathsf{T}}\widehat{\mathbf{r}}_0$*, and* $\delta_0 := \|\widehat{\mathbf{r}}_0\|^2$.
Then, for $n = 0, 1, 2, \ldots,$ *compute*

$$\begin{aligned}
\delta'_n &:= \|\mathbf{v}_n\|^2_{\mathbf{A}}\,, && (9.141a)\\
\omega_n &:= \delta_n/\delta'_n\,, && (9.141b)\\
\mathbf{x}_{n+1} &:= \mathbf{x}_n + \mathbf{v}_n\omega_n\,, && (9.141c)\\
\widehat{\mathbf{r}}_{n+1} &:= \widehat{\mathbf{r}}_n - \mathbf{KAv}_n\omega_n\,, && (9.141d)\\
\delta_{n+1} &:= \|\widehat{\mathbf{r}}_{n+1}\|^2\,, && (9.141e)\\
\psi_n &:= -\delta_{n+1}/\delta_n\,, && (9.141f)\\
\mathbf{v}_{n+1} &:= \mathbf{K}^{\mathsf{T}}\widehat{\mathbf{r}}_{n+1} - \mathbf{v}_n\psi_n\,. && (9.141g)
\end{aligned}$$

If $\delta_{n+1} \leq \mathrm{tol}$*, the algorithm terminates and* $\mathbf{x}_{n+1}$ *is a sufficiently accurate approximation of the solution.*

The algorithm makes use of the fact that for the preconditioned direction vectors $\widehat{\mathbf{v}}_n :\equiv \mathbf{K}^{-\mathsf{T}}\mathbf{v}_n$, which do not appear, we have

$$\|\widehat{\mathbf{v}}_n\|^2_{\widehat{\mathbf{A}}} = \left\langle \widehat{\mathbf{v}}_n, \widehat{\mathbf{A}}\widehat{\mathbf{v}}_n \right\rangle = \left\langle \mathbf{K}^{-\mathsf{T}}\mathbf{v}_n, \widehat{\mathbf{A}}\mathbf{K}^{-\mathsf{T}}\mathbf{v}_n \right\rangle = \langle \mathbf{v}_n, \mathbf{A}\mathbf{v}_n \rangle = \|\mathbf{v}_n\|^2_{\mathbf{A}}\,.$$

Let us now turn to a widely used alternative based on preconditioning $\mathbf{Ax} = \mathbf{b}$ on the left with the inverse $\mathbf{M}^{-1}$ of an spd matrix $\mathbf{M}$ that is in some sense an approximation of $\mathbf{A}$. So, we solve (as in (9.73), but in different notation),

$$\boxed{\underbrace{\mathbf{M}^{-1}\mathbf{A}}_{\widehat{\mathbf{A}}}\ \mathbf{x} = \underbrace{\mathbf{M}^{-1}\mathbf{b}}_{\widehat{\mathbf{b}}}\ .} \tag{9.142}$$

The CG method can be defined for any inner product space, in particular for $\mathbb{R}^N$ with the inner product induced by $\mathbf{M}$, the so-called $\mathbf{M}$**–inner product** [$\mathbf{M}$–*Skalarprodukt*],

$$\boxed{\langle \mathbf{x}, \mathbf{y} \rangle_{\mathbf{M}} :\equiv \langle \mathbf{x}, \mathbf{My} \rangle = \langle \mathbf{Mx}, \mathbf{y} \rangle\ .}$$

The matrix $\widehat{\mathbf{A}} :\equiv \mathbf{M}^{-1}\mathbf{A}$ in (9.142) is $\mathbf{M}$**–self-adjoint** [$\mathbf{M}$–*selbst-adjungiert*], that is, it is self-adjoint (or, symmetric) with respect to the $\mathbf{M}$–inner product:

$$\left\langle \mathbf{M}^{-1}\mathbf{Ax}, \mathbf{y} \right\rangle_{\mathbf{M}} = \langle \mathbf{Ax}, \mathbf{y} \rangle = \langle \mathbf{x}, \mathbf{Ay} \rangle = \left\langle \mathbf{x}, \mathbf{M}^{-1}\mathbf{Ay} \right\rangle_{\mathbf{M}}\ . \tag{9.143}$$

Noting that

$$\langle \widehat{\mathbf{r}}_n, \widehat{\mathbf{r}}_n \rangle_{\mathbf{M}} = \langle \mathbf{r}_n, \widehat{\mathbf{r}}_n \rangle\ , \qquad \left\langle \mathbf{v}_n, \mathbf{M}^{-1}\mathbf{A}\mathbf{v}_n \right\rangle_{\mathbf{M}} = \langle \mathbf{v}_n, \mathbf{A}\mathbf{v}_n \rangle$$

we obtain the following version of a PCG algorithm suggested by [Mv77]:

Algorithm 9.7 (PCG–OMIN WITH $\mathbf{M}$–INNER PRODUCT). *For solving* $\mathbf{Ax} = \mathbf{b}$ *with spd* $\mathbf{A}$ *and an spd left preconditioner* $\mathbf{M}$ *as in* (9.142) *choose an initial approximation* $\mathbf{x}_0$*, and let* $\mathbf{r}_0 := \mathbf{b} - \mathbf{Ax}_0$, $\mathbf{v}_0 := \widehat{\mathbf{r}}_0 := \mathbf{M}^{-1}\mathbf{r}_0$*, and* $\delta_0 := \langle \mathbf{r}_0, \widehat{\mathbf{r}}_0 \rangle$.
Then, for $n = 0, 1, 2, \dots$*, compute*

$$\begin{aligned}
\delta'_n &:= \|\mathbf{v}_n\|^2_{\mathbf{A}} \,, & \text{(9.144a)}\\
\omega_n &:= \delta_n / \delta'_n \,, & \text{(9.144b)}\\
\mathbf{x}_{n+1} &:= \mathbf{x}_n + \mathbf{v}_n \omega_n \,, & \text{(9.144c)}\\
\mathbf{r}_{n+1} &:= \mathbf{r}_n - \mathbf{A}\mathbf{v}_n \omega_n \,, & \text{(9.144d)}\\
\widehat{\mathbf{r}}_{n+1} &:= \mathbf{M}^{-1}\mathbf{r}_{n+1} \,, & \text{(9.144e)}\\
\delta_{n+1} &:= \langle \mathbf{r}_{n+1}, \widehat{\mathbf{r}}_{n+1} \rangle \,, & \text{(9.144f)}\\
\psi_n &:= -\delta_{n+1} / \delta_n \,, & \text{(9.144g)}\\
\mathbf{v}_{n+1} &:= \widehat{\mathbf{r}}_{n+1} - \mathbf{v}_n \psi_n \,. & \text{(9.144h)}
\end{aligned}$$

If $\delta_{n+1} \leq \text{tol}$ *(or, if* $\|\mathbf{r}_{n+1}\|^2 \leq \text{tol}$*), the algorithm terminates and* $\mathbf{x}_{n+1}$ *is a sufficiently accurate approximation of the solution.*

9.6.8 CG and CR for Complex Systems★

The CG and CR methods can also be applied to linear systems $\mathbf{Ax} = \mathbf{b}$ with complex data $\mathbf{A}$ and $\mathbf{b}$, but again, for CG the matrix must be Hermitian positive definite (Hpd). Of course, in the formulas $\mathbf{A}^{\mathsf{T}}$ has to be replaced by $\mathbf{A}^{\mathsf{H}}$, and $\langle \mathbf{x}, \mathbf{y} \rangle :\equiv \mathbf{x}^{\mathsf{T}}\mathbf{y}$ becomes[4] $\langle \mathbf{x}, \mathbf{y} \rangle :\equiv \mathbf{x}^{\mathsf{H}}\mathbf{y}$.

Some adaptations are necessary in the treatment of error functional minimization: In particular, in (9.80) $\mathbf{b}^{\mathsf{T}}\mathbf{x}$ is replaced by $\mathsf{Re}\,(\mathbf{b}^{\mathsf{H}}\mathbf{x}) = \mathsf{Re}\,\langle \mathbf{b}, \mathbf{x} \rangle$, and in (9.131) we have $\mathsf{Re}\,(\mathbf{b}^{\mathsf{H}}\mathbf{Ax}) = \mathsf{Re}\,\langle \mathbf{b}, \mathbf{Ax} \rangle$.

With these changes, virtually everything remains correct (I hope!).

In the following we normally allow that the data are complex, although in nearly all applications they are real. Recall that we let $\mathbb{E} :\equiv \mathbb{R}$ or $\mathbb{C}$ to combine the real and the complex cases, and that we denote the adjoint of $\mathbf{A}$ by $\mathbf{A}^\star$ — so this is the transpose $\mathbf{A}^{\mathsf{T}}$ if $\mathbf{A}$ is real, and the Hermitian transpose $\mathbf{A}^{\mathsf{H}}$ if $\mathbf{A}$ is complex.

9.6.9 CG for Least Squares Problems★

When parameters of a linear model are determined by measurements, the influence of measurement errors can be reduced by making many more measurements than the number of parameters. This leads to an **overdetermined linear system** $\mathbf{Ax} \approx \mathbf{b}$ of M equations in N unknowns, where $M > N$.

The classical approach to this problem, introduced by Gauss, is to determine the approximate solution for which the (Euclidean) 2-norm of the residual is minimized, or, what amounts to the same, where the square of this 2-norm is minimized:

$$\boxed{\|\mathbf{r}\|^2 = \|\mathbf{b} - \mathbf{Ax}\|^2 = \min!} \qquad (9.145)$$

Since $\|\mathbf{r}\|^2 = \sum r_i^2$, Gauss called this approximation the **least squares solution**. The geometric properties of the Euclidean space imply that the residual of the least squares solution is orthogonal to the column space of $\mathbf{A}$, so $\mathbf{A}^\star\mathbf{r} = \mathbf{o}$ or

$$\boxed{\mathbf{A}^\star\mathbf{Ax} = \mathbf{A}^\star\mathbf{b}\,.} \qquad (9.146)$$

These are the **normal equations**. If we assume that $\mathbf{A}$ has full column rank (*i.e.*, $\mathsf{rank}\,\mathbf{A} = N$), then $\mathbf{A}^\star\mathbf{A}$ is Hpd and the CG method can be applied to the normal equations (9.146). This approach is known as the CGNR **method**; the “N” and “R” refer to the normal equations and the residual, respectively. It means that the approximate solutions are chosen such that

$$\boxed{\mathbf{x}_n - \mathbf{x}_0 \in \mathcal{K}_n(\mathbf{A}^\star\mathbf{A}, \mathbf{A}^\star\mathbf{r}_0)\,,} \quad \text{where} \quad \mathbf{r}_0 :\equiv \mathbf{b} - \mathbf{Ax}_0\,. \qquad (9.147)$$

[4] Attention: many people define $\langle \mathbf{x}, \mathbf{y} \rangle :\equiv \mathbf{y}^{\mathsf{H}}\mathbf{x}$ instead.

Hence, with $\mathbf{B} :\equiv \mathbf{A}\mathbf{A}^\star$, the residuals $\mathbf{r}_n :\equiv \mathbf{b} - \mathbf{A}\mathbf{x}_n$ satisfy

$$\boxed{\mathbf{r}_n - \mathbf{r}_0 \in \mathbf{A}\,\mathcal{K}_n(\mathbf{A}^\star\mathbf{A}, \mathbf{A}^\star\mathbf{r}_0) = \mathbf{A}\mathbf{A}^\star\,\mathcal{K}_n(\mathbf{A}\mathbf{A}^\star, \mathbf{r}_0) = \mathbf{B}\mathcal{K}_n(\mathbf{B}, \mathbf{r}_0)\,.} \tag{9.148}$$

According to the CG optimality property $\mathbf{x}_n$ minimizes the $\mathbf{A}^\star\mathbf{A}$-norm of the error $\mathbf{x}_n - (\mathbf{A}^\star\mathbf{A})^{-1}\mathbf{A}^\star\mathbf{b}$ of the normal equations (9.146) subject to the condition (9.147), but the $\mathbf{A}^\star\mathbf{A}$-norm of the error is just the 2-norm of the residual. So, in each step CGNR produces a least squares solution subject to a reduced space defined by the condition (9.147). Adapting the derivation of (9.201) below we could see that the residuals $\mathbf{r}_n$ satisfy the Galerkin condition

$$\boxed{\mathbf{r}_n \perp \mathbf{A}\mathcal{K}_n(\mathbf{A}^\star\mathbf{A}, \mathbf{A}^\star\mathbf{r}_0) = \mathbf{B}\mathcal{K}_n(\mathbf{B}, \mathbf{r}_0)\,.} \tag{9.149}$$

For the residuals $\mathbf{s}_n :\equiv \mathbf{A}^\star\mathbf{r}_n$ of the normal equations we have then

$$\boxed{\mathbf{s}_n - \mathbf{s}_0 \in \mathbf{A}^\star\mathbf{A}\,\mathcal{K}_n(\mathbf{A}^\star\mathbf{A}, \mathbf{s}_0)\,, \qquad \mathbf{s}_n \perp \mathcal{K}_n(\mathbf{A}^\star\mathbf{A}, \mathbf{s}_0)\,,} \tag{9.150}$$

which reflects properties of CG applied to the normal equations.

Let us summarize some of these results:

THEOREM 9.6.6. *Let a full rank overdetermined rectangular system* $\mathbf{A}\mathbf{x} = \mathbf{b}$ *be given; so* $\mathbf{A}$ *is* $M \times N$, $\mathsf{rank}\,\mathbf{A} = N \leq M$. *Its least squares solution minimizing* (9.145) *is the unique solution of the normal equations* (9.146).
The CGNR *method consisting of applying the* CG *method to these normal equations has the following properties:*

- *The iterates* $\mathbf{x}_n$ *minimize the 2-norm of the residuals* $\mathbf{r}_n :\equiv \mathbf{b} - \mathbf{A}\mathbf{x}_n$ *of the original system subject to the condition* (9.147), *and in this sense they are least squares solution of the given system restricted further by* (9.147).
- *The residuals* $\mathbf{r}_n$ *satisfy* (9.148) *and* (9.149). *For the residuals* $\mathbf{s}_n$ *of the normal equations holds* (9.150).

On the other hand, if the given system $\mathbf{A}\mathbf{x} = \mathbf{b}$ is an **underdetermined linear system** of M equations in N unknowns, where $M < N$, then we can restrict $\mathbf{x}$ to the image of $\mathbf{A}^\star$ in order to define a unique solution if $\mathbf{A}$ has full rank M, which we assume again. We define $\mathbf{z}$ by

$$\mathbf{x} \equiv: \mathbf{A}^\star\mathbf{z} \tag{9.151}$$

and write the given system as

$$\boxed{\mathbf{A}\mathbf{A}^\star\mathbf{z} = \mathbf{b}\,,} \tag{9.152}$$

which is again referred to as **normal equations**. We claim that (9.151) restricts $\mathbf{x}$ to be orthogonal to the kernel of $\mathbf{A}$. In fact, we can write any $\mathbf{x}' \in \mathbb{E}^N$ as

$$\mathbf{x}' = \mathbf{x}^\perp + \mathbf{A}^\star\widetilde{\mathbf{z}}\,, \qquad \text{where} \quad \mathbf{x}^\perp \perp \mathsf{im}\,\mathbf{A}^\star\,, \quad \widetilde{\mathbf{z}} \in \mathbb{E}^M\,. \tag{9.153}$$

The latter means that $\langle \mathbf{x}^\perp, \mathbf{A}^\star\mathbf{w}\rangle = 0$ $(\forall \mathbf{w} \in \mathbb{E}^M)$ or $\mathbf{A}\mathbf{x}^\perp \perp \mathbb{E}^M$, which implies that $\mathbf{A}\mathbf{x}^\perp = \mathbf{o}$. In other words, $\mathbf{x}^\perp$ lies in the kernel of $\mathbf{A}$. Actually, it is a well-known fact from matrix theory that the kernel (or null space) of $\mathbf{A}$ and the image (or range) of $\mathbf{A}^\star$ are orthogonal complementary subspaces. We conclude that $\mathbf{A}\mathbf{x}' = \mathbf{A}\mathbf{A}^\star\widetilde{\mathbf{z}}$. So any solution of the given system $\mathbf{A}\mathbf{x}' = \mathbf{b}$ has the form (9.153) with $\widetilde{\mathbf{z}} = \mathbf{z}$ being a solution of (9.152) and $\mathbf{x}^\perp \in \mathsf{ker}\,\mathbf{A}$. ¿From Pythagoras' theorem we can conclude further that $\|\mathbf{x}'\|^2 = \|\mathbf{x}^\perp\|^2 + \|\mathbf{A}^\star\mathbf{z}\|^2$. Hence, $\mathbf{x} = \mathbf{A}^\star\mathbf{z}$ is the shortest solution. In other words, if $\mathbf{A}$ has full rank M $(< N)$, then under the substitution $\mathbf{x} = \mathbf{A}^\star\mathbf{z}$ the system (9.152) is equivalent with

$$\boxed{\mathbf{A}\mathbf{x} = \mathbf{b}\,, \qquad \|\mathbf{x}\| = \min!} \tag{9.154}$$

Since $\mathbf{A}\mathbf{A}^\star$ is Hpd, we can apply the CG method to the normal equations (9.152). This is the CGNE **method**, where "E" now refers to the error. It is also called **Craig's method**. If $\mathbf{z}_n$ is any approximate solution, the corresponding residual is

$$\|\mathbf{r}_n\| = \|\mathbf{b} - \mathbf{A}\mathbf{A}^\star\mathbf{z}_n\| = \|\mathbf{b} - \mathbf{A}\mathbf{x}_n\| \tag{9.155}$$

and is also the residual of the original system. So, implicitly, CGNE yields approximate solutions $\mathbf{z}_n$ of the normal equations (9.152) with

$$\boxed{\mathbf{z}_n - \mathbf{z}_0 \in \mathcal{K}_n(\mathbf{A}\mathbf{A}^\star, \mathbf{r}_0) = \mathcal{K}_n(\mathbf{B}, \mathbf{r}_0)\,,} \tag{9.156}$$

corresponding approximate solutions of the original system with

$$\boxed{\mathbf{x}_n - \mathbf{x}_0 \in \mathbf{A}^\star\mathcal{K}_n(\mathbf{A}\mathbf{A}^\star, \mathbf{r}_0) = \mathcal{K}_n(\mathbf{A}^\star\mathbf{A}, \mathbf{A}^\star\mathbf{r}_0)\,,} \tag{9.157}$$

as well as residuals that satisfy again (9.148) and, as can be seen,

$$\boxed{\mathbf{r}_n \perp \mathcal{K}_n(\mathbf{A}\mathbf{A}^\star, \mathbf{r}_0) = \mathcal{K}_n(\mathbf{B}, \mathbf{r}_0)\,.} \tag{9.158}$$

Consequently, the relevant Krylov space is the same as for CGNR, but the Galerkin condition is different.

In the nth step, the $\mathbf{A}\mathbf{A}^\star$-norm or $\mathbf{B}$-norm of the error $\mathbf{z}_n - (\mathbf{A}\mathbf{A}^\star)^{-1}\mathbf{b}$ of (9.152) is minimized subject to the condition (9.156). This is the same as minimizing the 2-norm of the error $\mathbf{x}_n - \mathbf{x}_\star$ of the original system subject to the condition (9.157).

THEOREM 9.6.7. *Let a full rank underdetermined rectangular system* $\mathbf{A}\mathbf{x} = \mathbf{b}$ *be given; so* $\mathbf{A}$ *is* $M \times N$, $\mathsf{rank}\,\mathbf{A} = M < N$. *Its least squares solution minimizing* (9.154) *is the unique solution of the normal equations* (9.152). *Any other solution is of the form* $\mathbf{x} = \mathbf{x}^\perp + \mathbf{A}^\star\mathbf{z}$, *where* $\mathbf{x}^\perp \perp \mathsf{im}\,\mathbf{A}^\star$, *which is equivalent to* $\mathbf{x}^\perp \in \mathsf{ker}\,\mathbf{A}$.
The CGNE *algorithm consisting of applying the* CG *method to these normal equations has the following properties:*

- $\mathbf{z}_n$ *minimizes the* $\mathbf{B}$*-norm of the error of the normal equations* (9.152) *subject to* (9.156), *where* $\mathbf{B} :\equiv \mathbf{A}\mathbf{A}^\star$.
- *The corresponding original unknowns* $\mathbf{x}_n = \mathbf{A}^\star\mathbf{z}_n$ *minimize the 2-norm of the errors* $\mathbf{x}_n - \mathbf{x}_\star$ *of the original system subject to the condition* (9.157).
- *The residuals* $\mathbf{r}_n$, *which are both the residuals of the original system and the normal equations, satisfy* (9.148) (*as in* CGNR) *and the Galerkin condition* (9.158).

The theorem remains correct for $M = N$ if we let $\mathbf{x}^\perp := \mathbf{o}$. The condition $\|\mathbf{x}\| = \min!$ in (9.154) can be dropped because $\mathbf{A}\mathbf{x} = \mathbf{b}$ has then a unique solution.

Let us now look at the normally used OMIN versions of the CGNR and CGNE algorithms. We have to adapt Algorithm 9.2 of Section 9.6.4 so that it is applied to the normal equations (9.146) and (9.152), respectively.

For CGNR the relevant residuals that are orthogonal to each other are the residuals $\mathbf{s}_n (= \mathbf{A}^\star\mathbf{r}_0)$ of the normal equations that satisfy (9.150). They can be used to determine δ_n. The search directions $\mathbf{v}_n$ are now $\mathbf{A}^\star\mathbf{A}$-orthogonal, and thus $\delta_n' := \|\mathbf{v}_n\|^2_{\mathbf{A}^\star\mathbf{A}} = \|\mathbf{A}\mathbf{v}_n\|^2$.

Algorithm 9.8 (OMIN FORM OF THE CGNR METHOD)**.** .
For solving the least squares problem (9.145) *via the normal equations* (9.146) *choose an initial approximation* $\mathbf{x}_0$*, and let* $\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$*,* $\mathbf{v}_0 := \mathbf{s}_0 := \mathbf{A}^\star \mathbf{r}_0$*,* $\delta_0 := \|\mathbf{s}_0\|^2$*, and* $\psi_{-1} := 0$*. Then, for* $n = 0, 1, 2, \dots$*, compute*

$$\begin{aligned} \delta'_n &:= \|\mathbf{A}\mathbf{v}_n\|^2\,, && (9.159a)\\ \omega_n &:= \delta_n/\delta'_n\,, && (9.159b)\\ \mathbf{x}_{n+1} &:= \mathbf{x}_n + \mathbf{v}_n\omega_n\,, && (9.159c)\\ \mathbf{r}_{n+1} &:= \mathbf{r}_n - \mathbf{A}\mathbf{v}_n\omega_n\,, && (9.159d)\\ \mathbf{s}_{n+1} &:= \mathbf{A}^\star\mathbf{r}_{n+1}\,, && (9.159e)\\ \delta_{n+1} &:= \|\mathbf{s}_{n+1}\|^2\,, && (9.159f)\\ \psi_n &:= -\,\delta_{n+1}/\delta_n\,, && (9.159g)\\ \mathbf{v}_{n+1} &:= \mathbf{s}_{n+1} - \mathbf{v}_n\psi_n\,. && (9.159h) \end{aligned}$$

If $\|\mathbf{r}_{n+1}\| \leq \mathrm{tol}$*, the algorithm terminates and* $\mathbf{x}_{n+1}$ *is a sufficiently accurate approximation of the solution.*

We have chosen to update in (9.159d) also $\mathbf{r}_n$, which allows us to rely on $\|\mathbf{r}_{n+1}\|$ for termination. One could instead update directly $\mathbf{s}_n$ according to

$$\mathbf{s}_{n+1} := \mathbf{s}_n - \mathbf{A}^\star\mathbf{A}\mathbf{v}_n\omega_n\,, \tag{9.160}$$

but then termination must rely on $\|\mathbf{s}_{n+1}\|$.

For CGNE the residuals $\mathbf{r}_n$ are the same for the original and the normal equations (9.152), but the iterates seem to change to $\mathbf{z}_n$. However, it turns out that instead everything can be formulated in terms of the approximate solutions $\mathbf{x}_n = \mathbf{A}^\star\mathbf{z}_n$ of the original system and the corresponding search directions.

Algorithm 9.9 (OMIN FORM OF THE CGNE METHOD)**.** .
For solving the minimum solution least squares problem (9.154) *via the normal equations* (9.152) *choose an initial approximation* $\mathbf{x}_0$*, and let* $\mathbf{v}_0 := \mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$*,* $\delta_0 := \|\mathbf{r}_0\|^2$*, and* $\psi_{-1} := 0$*.*
Then, for $n = 0, 1, 2, \dots$*, compute*

$$\begin{aligned} \delta'_n &:= \|\mathbf{v}_n\|^2\,, && (9.161a)\\ \omega_n &:= \delta_n/\delta'_n\,, && (9.161b)\\ \mathbf{x}_{n+1} &:= \mathbf{x}_n + \mathbf{v}_n\omega_n\,, && (9.161c)\\ \mathbf{r}_{n+1} &:= \mathbf{r}_n - \mathbf{A}\mathbf{v}_n\omega_n\,, && (9.161d)\\ \delta_{n+1} &:= \|\mathbf{r}_{n+1}\|^2\,, && (9.161e)\\ \psi_n &:= -\,\delta_{n+1}/\delta_n\,, && (9.161f)\\ \mathbf{v}_{n+1} &:= \mathbf{A}^\star\mathbf{r}_{n+1} - \mathbf{v}_n\psi_n\,. && (9.161g) \end{aligned}$$

If $\|\mathbf{r}_{n+1}\| \leq \mathrm{tol}$*, the algorithm terminates and* $\mathbf{x}_{n+1}$ *is a sufficiently accurate approximation of the solution.*

A potential disadvantage of using the normal equations (9.146) and (9.152) is that the condition number of $\mathbf{A}^\star\mathbf{A}$ and $\mathbf{A}\mathbf{A}^\star$ may be large compared to the one of the triangular matrix $\mathbf{R}$ in a QR decomposition of $\mathbf{A}$. In particular, if $\mathbf{A}$ is square and nonsingular,

$$\kappa_2(\mathbf{A}^\star\mathbf{A}) = \kappa_2(\mathbf{A}\mathbf{A}^\star) = (\kappa_2(\mathbf{A}))^2 = (\kappa_2(\mathbf{R}))^2\,.$$

9.7 The Symmetric Lanczos Process*

9.7.1 The Lanczos Process and Its Relation to the CG Method*

We have seen in Lemma 9.6.3 that the CG residuals are mutually orthogonal. So, as long as $\mathbf{r}_m \neq \mathbf{0}$, that is, the Krylov space is not exhausted, the set $\{\mathbf{r}_0, \mathbf{r}_1, \dots, \mathbf{r}_m\}$ forms an orthogonal basis of $\mathcal{K}_{m+1}(\mathbf{A}, \mathbf{r}_0)$.

To find an orthonormal basis $\{\mathbf{y}_0, \mathbf{y}_1, \dots, \mathbf{y}_m\}$, we have just to scale each basis vector. We also include a factor $(-1)^n$ that will be justified in a moment:

$$\mathbf{y}_n :\equiv (-1)^n \frac{\mathbf{r}_n}{\|\mathbf{r}_n\|} . \tag{9.162}$$

In principle, we can use any of our three versions of the CG method for computing residuals, but if our goal is to construct an orthogonal basis it seems most appropriate to use the ORES version, in particular the three-term recursion (9.123e),

$$\mathbf{r}_{n+1} := (\mathbf{A}\mathbf{r}_n - \mathbf{r}_n\alpha_n - \mathbf{r}_{n-1}\beta_{n-1})/\gamma_n .$$

It is easy to modify this recursion so that it produces directly orthonormal vectors. In fact, α_n and β_n were chosen to make $\mathbf{r}_{n+1}$ orthogonal to $\mathbf{r}_n$ and $\mathbf{r}_{n-1}$, respectively. So we just have to replace $\mathbf{r}$ by $\mathbf{y}$ in the formulas, that is, in (9.123a), (9.123b), and (9.123f):

$$\alpha_n := \|\mathbf{y}_n\|^2_{\mathbf{A}} \,/\, \delta_n , \qquad \beta_{n-1} := \gamma_{n-1}\delta_n/\delta_{n-1} , \qquad \delta_{n+1} := \|\mathbf{y}_{n+1}\|^2 .$$

However, $\delta_n := \|\mathbf{y}_n\|^2 = 1$ $(n = 0, 1, \dots)$ now, and therefore,

$$\beta_n = \gamma_n \qquad (n = 0, 1, \dots). \tag{9.163}$$

In contrast, formula (9.123c) for γ_n needs to be replaced: γ_n is now chosen to normalize $\mathbf{y}_{n+1}$. This determines $|\gamma_n|$, but not yet its sign or, in the complex case, its argument. While it can be shown that in ORES (with $\mathbf{A}$ spd or Hpd) we have $\gamma_n < 0$ and $\beta_n < 0$, we opt here for $\gamma_n > 0$. This entails the factor $(-1)^n$ in (9.162); see (9.183) below, where we compare the polynomials associated with the CG residuals $\mathbf{r}_n$ and those for the normalized vectors $\mathbf{y}_n$.

Altogether we obtain the following algorithm for computing an orthonormal basis $\{\mathbf{y}_0, \mathbf{y}_1, \dots, \mathbf{y}_m\}$ for the Krylov subspace $\mathcal{K}_{m+1}(\mathbf{A}, \mathbf{y}_0)$, where $m < \bar{\nu}(\mathbf{y}_0, \mathbf{A})$. We call it the **symmetric Lanczos algorithm** [*symmetrischer Lanczos-Algorithmus*]. It adapts the **Lanczos process** [*Lanczos-Prozess*] from [Lan50] to a Hermitian or (real symmetric) matrix $\mathbf{A}$, which needs not be positive definite. Note that Lanczos' method predates the conjugate gradient method by two years. For clarity we denote its recursion coefficients by α_n^{L} and β_n^{L} to distinguish them from α_n and β_n in the ORES version of the CG method.

The formulas derived above are also correct in the (complex) Hermitian case. Note, that even in that case α_n^{L} and β_n^{L} are real, the latter nonnegative.

Algorithm 9.10 (SYMMETRIC LANCZOS ALGORITHM)**.** .
Let a Hermitian (or real symmetric) matrix $\mathbf{A}$ and a vector $\mathbf{y}_0$ of norm 1 be given, and set $\beta_{-1}^{\mathrm{L}} := 0$. For constructing a nested set of orthonormal bases $\{\mathbf{y}_0, \mathbf{y}_1, \dots, \mathbf{y}_m\}$ for the nested Krylov subspaces $\mathcal{K}_{m+1}(\mathbf{A}, \mathbf{y}_0)$ $(m = 1, 2, \dots, \bar{\nu}(\mathbf{y}_0, \mathbf{A}) - 1)$ compute, for $n = 0, 1, \dots, m-1$,

$$\alpha_n^{\mathrm{L}} := \langle \mathbf{y}_n, \mathbf{A}\mathbf{y}_n \rangle , \tag{9.164a}$$

$$\widetilde{\mathbf{y}}_{n+1} := \mathbf{A}\mathbf{y}_n - \mathbf{y}_n\alpha_n^{\mathrm{L}} - \mathbf{y}_{n-1}\beta_{n-1}^{\mathrm{L}} , \tag{9.164b}$$

$$\beta_n^{\mathrm{L}} := \|\widetilde{\mathbf{y}}_{n+1}\| , \tag{9.164c}$$

$$\mathbf{y}_{n+1} := \widetilde{\mathbf{y}}_{n+1}/\beta_n^{\mathrm{L}} . \tag{9.164d}$$

When $n = m = \bar{\nu} - 1$, then $\widetilde{\mathbf{y}}_{n+1} = \mathbf{o}$.

Of course, the last statement is normally only valid in exact arithmetic.

The orthonormal vectors $\mathbf{y}_n$ are often called **Lanczos vectors**.

One step of this algorithm can be understood as one step of a Gram–Schmidt orthogonalization process: $\mathbf{A}\mathbf{y}_n$ is made orthogonal to $\mathbf{y}_n$ and $\mathbf{y}_{n-1}$, and the resulting projection onto the orthogonal complement of span $\{\mathbf{y}_n, \mathbf{y}_{n-1}\}$ is normalized. However, here this projection is automatically orthogonal to $\mathbf{y}_0$, ..., $\mathbf{y}_{n-2}$. This follows from Lemma 9.6.3. We will give another derivation of the symmetric Lanczos algorithm with a direct proof of this property later.

The recurrence given by (9.164b) and (9.164d) implies that

$$\mathbf{A}\mathbf{y}_n = \mathbf{y}_{n+1}\beta_n^{\mathrm{L}} + \mathbf{y}_n\alpha_n^{\mathrm{L}} + \mathbf{y}_{n-1}\beta_{n-1}^{\mathrm{L}} . \tag{9.165}$$

Again, for all n, these relations can be gathered into a matrix equation: as in (9.109a) and (9.119) we let

$$\mathbf{Y}_m :\equiv \begin{pmatrix} \mathbf{y}_0 & \mathbf{y}_1 & \cdots & \mathbf{y}_{m-1} \end{pmatrix} \tag{9.166}$$

and

$$\underline{\mathbf{T}}_m :\equiv \left(\begin{array}{c} \mathbf{T}_m \\ \hline \beta^{\mathsf{L}}_{m-1} \mathbf{l}^{\mathsf{T}}_m \end{array} \right) :\equiv \begin{pmatrix} \alpha^{\mathsf{L}}_0 & \beta^{\mathsf{L}}_0 & & \\ \beta^{\mathsf{L}}_0 & \alpha^{\mathsf{L}}_1 & \ddots & \\ & \beta^{\mathsf{L}}_1 & \ddots & \beta^{\mathsf{L}}_{m-2} \\ & & \ddots & \alpha^{\mathsf{L}}_{m-1} \\ & & & \beta^{\mathsf{L}}_{m-1} \end{pmatrix} . \tag{9.167}$$

Then (9.165) is for $n = 0, 1, \ldots, m-1$ expressed by

$$\boxed{\mathbf{A}\mathbf{Y}_m = \mathbf{Y}_{m+1}\underline{\mathbf{T}}_m \,.} \tag{9.168}$$

Here, $\underline{\mathbf{T}}_m$ has no longer vanishing column sums, but its principal submatrices, in particular $\mathbf{T}_m$, are now real symmetric. Since the columns of $\mathbf{Y}_{m+1}$ are orthonormal, we have

$$\mathbf{Y}^{\star}_m \mathbf{Y}_m = \mathbf{I}_m \,, \qquad \mathbf{Y}^{\star}_m \mathbf{Y}_{m+1} = \begin{pmatrix} \mathbf{I}_m \mid \mathbf{o} \end{pmatrix} , \tag{9.169}$$

so that

$$\boxed{\mathbf{Y}^{\star}_m \mathbf{A}\mathbf{Y}_m = \mathbf{T}_m \,.} \tag{9.170}$$

As we have noted at the end of Algorithm 9.10, when $n = m = \bar{\nu} - 1$, then $\widetilde{\mathbf{y}}_{n+1} \equiv \mathbf{y}_{\bar{\nu}} = \mathbf{o}$, at least in theory. So, for $m = \bar{\nu}$ we have

$$\boxed{\mathbf{A}\mathbf{Y}_{\bar{\nu}} = \mathbf{Y}_{\bar{\nu}}\mathbf{T}_{\bar{\nu}}} \qquad \text{and} \qquad \boxed{\mathbf{Y}^{\star}_{\bar{\nu}}\mathbf{A}\mathbf{Y}_{\bar{\nu}} = \mathbf{T}_{\bar{\nu}} \,.} \tag{9.171}$$

The relation at left has the following meaning:

LEMMA 9.7.1. *The Krylov subspace $\mathcal{K}_{\bar{\nu}}$, which is the column space of $\mathbf{Y}_{\bar{\nu}}$, is an invariant subspace of $\mathbf{A}$. The restriction of the linear operator $\mathbf{A}$ to this subspace has the matrix representation $\mathbf{T}_{\bar{\nu}}$ if expressed in the orthonormal basis $\{\mathbf{y}_0, \ldots, \mathbf{y}_{\bar{\nu}-1}\}$.*

In particular, in this case of termination, every eigenvalue of $\mathbf{T}_{\bar{\nu}}$ is an eigenvalue of $\mathbf{A}$ (see [Par98] for a proof).This raises the question whether before, under certain conditions, the eigenvalues of $\mathbf{T}_m$ approximate eigenvalues of $\mathbf{A}$. This is discussed in Section 9.7.2.

Here we want to further explore the connection between CG and the symmetric Lanczos algorithm. Let

$$\mathbf{D}_{\pm\rho;\, m} :\equiv \mathsf{diag}\,\{\rho_0, -\rho_1, \rho_2, \ldots, (-1)^{m-1}\rho_{m-1}\} \,, \tag{9.172}$$

where $\rho_n :\equiv \|\mathbf{r}_n\|$, so that by (9.162)

$$\mathbf{R}_m = \mathbf{Y}_m \mathbf{D}_{\pm\rho;\, m} \,. \tag{9.173}$$

By the orthogonality of the residuals $\mathbf{R}^{\star}_m \mathbf{R}_{m+1} = \begin{pmatrix} \mathbf{D}_{\pm\rho;\, m} \mid \mathbf{o} \end{pmatrix}$. Thus, the matrix identities (9.117) and (9.119) imply that

$$\mathbf{R}^{\star}_m \mathbf{A}\mathbf{R}_m = \mathbf{D}^2_{\pm\rho;\, m}\mathbf{T}^{\circ}_m \,, \tag{9.174}$$

and the comparison with (9.170) yields

$$\begin{aligned} \mathbf{T}^{\circ}_m &= \mathbf{D}^{-2}_{\pm\rho;\, m}\mathbf{R}^{\star}_m \mathbf{A}\mathbf{R}_m \\ &= \mathbf{D}^{-1}_{\pm\rho;\, m}\mathbf{Y}^{\star}_m \mathbf{A}\mathbf{Y}_m \mathbf{D}_{\pm\rho;\, m} \\ &= \mathbf{D}^{-1}_{\pm\rho;\, m}\mathbf{T}_m \mathbf{D}_{\pm\rho;\, m} \,. \end{aligned} \tag{9.175}$$

So, the two tridiagonal matrices are diagonally similar. It is an easy exercise to prove that under any such similarity the diagonal elements remain invariant, and so do the products of any two symmetrically located elements on the first codiagonals:

$$\alpha_n = \alpha^{\mathsf{L}}_n \,, \qquad \beta_n \gamma_n = (\beta^{\mathsf{L}}_n)^2 \,. \tag{9.176}$$

By (9.162), the CG residual polynomials p_n defined by $\mathbf{r}_n = p_n(\mathbf{A})\mathbf{r}_0$ and the **Lanczos polynomials** p_n^{L} defined by $\mathbf{y}_n = p_n^{\mathsf{L}}(\mathbf{A})\mathbf{y}_0 = p_n^{\mathsf{L}}(\mathbf{A})\,\mathbf{r}_0/\rho_0$ are for $n < \bar{\nu}$ related just by scaling:

$$p_n^{\mathsf{L}}(t) = (-1)^n \frac{\rho_0}{\rho_n}\, p_n(t) \quad \text{or,} \quad p_n(t) = (-1)^n \frac{\rho_n}{\rho_0}\, p_n^{\mathsf{L}}(t)\,. \tag{9.177}$$

On the other hand, we know that $p_n(0) = 1$, hence

$$p_n(t) = \frac{p_n^{\mathsf{L}}(t)}{p_n^{\mathsf{L}}(0)}\,, \qquad \text{where} \quad p_n^{\mathsf{L}}(0) = (-1)^n \frac{\rho_0}{\rho_n}\,. \tag{9.178}$$

In analogy to the recursions for $\mathbf{r}_n$ and $\mathbf{y}_n$ there are recursions for these polynomials:

$$p_{n+1}(t) = \big((t-\alpha_n)p_n(t) - \beta_{n-1}p_{n-1}(t)\big)/\gamma_n\,, \tag{9.179}$$

$$p_{n+1}^{\mathsf{L}}(t) = \big((t-\alpha_n^{\mathsf{L}})p_n^{\mathsf{L}}(t) - \beta_{n-1}^{\mathsf{L}}p_{n-1}^{\mathsf{L}}(t)\big)/\beta_n^{\mathsf{L}}\,. \tag{9.180}$$

They get started with $p_0(t) \equiv 1$, $\beta_{-1} := 0$ and $p_0^{\mathsf{L}}(t) \equiv 1$, $\beta_{-1}^{\mathsf{L}} := 0$, respectively, and they show in particular that

$$p_n(t) = \frac{t^n}{\gamma_0 \cdots \gamma_{n-1}} + \cdots + 1\,, \tag{9.181}$$

$$p_n^{\mathsf{L}}(t) = \frac{t^n}{\beta_0^{\mathsf{L}} \cdots \beta_{n-1}^{\mathsf{L}}} + \cdots + (-1)^n \frac{\rho_0}{\rho_n}\,. \tag{9.182}$$

So, using (9.176) and (9.177) we obtain

$$\boxed{\frac{p_n(t)}{p_n^{\mathsf{L}}(t)} \equiv (-1)^n \frac{\rho_n}{\rho_0} = \frac{\beta_0^{\mathsf{L}} \cdots \beta_{n-1}^{\mathsf{L}}}{\gamma_0 \cdots \gamma_{n-1}} = \frac{\beta_0 \cdots \beta_{n-1}}{\beta_0^{\mathsf{L}} \cdots \beta_{n-1}^{\mathsf{L}}}\,.} \tag{9.183}$$

Like the recursions for the residuals and for the Lanczos vectors, those for the corresponding polynomials, that is, (9.179) and (9.180), can be summarized in matrix notation. We illustrate this for the Lanczos vectors and define the row vector

$$(\mathbf{p}_m^{\mathsf{L}}(t))^{\mathsf{T}} :\equiv \begin{pmatrix} p_0^{\mathsf{L}}(t) & \ldots & p_{m-1}^{\mathsf{L}}(t) \end{pmatrix}. \tag{9.184}$$

Then (9.180) turns into

$$\boxed{t(\mathbf{p}_m^{\mathsf{L}}(t))^{\mathsf{T}} = (\mathbf{p}_{m+1}^{\mathsf{L}}(t))^{\mathsf{T}}\, \underline{\mathbf{T}}_m\,.} \tag{9.185}$$

If we partition $\underline{\mathbf{T}}_m$ as in (9.167) into $\mathbf{T}_m$ and the last row $\beta_{m-1}^{\mathsf{L}}\mathbf{l}_m^{\mathsf{T}}$, we can write this as

$$t(\mathbf{p}_m^{\mathsf{L}}(t))^{\mathsf{T}} = (\mathbf{p}_m^{\mathsf{L}}(t))^{\mathsf{T}}\mathbf{T}_m + p_m^{\mathsf{L}}(t)\beta_{m-1}^{\mathsf{L}}\mathbf{l}_m^{\mathsf{T}} \tag{9.186}$$

or

$$\boxed{(\mathbf{p}_m^{\mathsf{L}}(t))^{\mathsf{T}}\big(t\mathbf{I} - \mathbf{T}_m\big) = p_m^{\mathsf{L}}(t)\beta_{m-1}^{\mathsf{L}}\mathbf{l}_m^{\mathsf{T}}\,.} \tag{9.187}$$

Since $p_0^{\mathsf{L}}(t) \equiv 1$ is never zero, $\mathbf{p}_m^{\mathsf{L}}(t) \neq \mathbf{o}$ for all t and all m. Consequently, when t is a zero of p_m^{L}, then it is an eigenvalue of $\mathbf{T}_m$, and the corresponding left eigenvector is $\mathbf{p}_m(t)$. But since $\mathbf{T}_m$ is symmetric, $\mathbf{p}_m(t)$ is also a right eigenvector. One can show that conversely every eigenvalue of $\mathbf{T}_m$ is a zero of p_m^{L}. Of course, all the eigenvalues of $\mathbf{T}_m$ are real. If we knew that they are all simple, we could further conclude that they match all the zeros of p_m^{L}, and that p_m^{L} is up to scaling equal to the characteristic polynomial $\chi_m(t) :\equiv \det\,(t\mathbf{I} - \mathbf{T}_m)$ of $\mathbf{T}_m$, which we choose to be monic. This is indeed true and can be verified by analyzing the recurrence

$$\chi_{n+1}(t) = (t - \alpha_n^{\mathsf{L}})\chi_n(t) - (\beta_{n-1}^{\mathsf{L}})^2\chi_{n-1}(t) \tag{9.188}$$

that holds for these polynomials. The following can be shown; for a proof, see, *e.g.,* page 137 of [SRS68].

THEOREM 9.7.2. *If all off-diagonal elements β_k^{L} of the real symmetric tridiagonal matrix $\mathbf{T}_m$ are nonzero, then, for $n = 1, \ldots, m$, the characteristic polynomial χ_n of its $n \times n$ leading principal submatrix $\mathbf{T}_n$ has simple zeros $\zeta_{1,n} > \zeta_{2,n} > \cdots > \zeta_{n,n}$, which, if $n < m$, separate those of χ_{n+1}:*

$$\zeta_{k,n+1} > \zeta_{k,n} > \zeta_{k+1,n+1}\,, \qquad k = 1, \ldots, n\,. \tag{9.189}$$

As pointed out before, this leads to the following conclusion.

COROLLARY 9.7.3. *The polynomials χ_n, p_n^{L}, and p_n differ only by scaling. So, they have the same zeros, and these satisfy* (9.189)*.*

The sequences of these polynomials (up to degree $\bar{\nu}$) are examples of a so-called **Sturm sequence** [*Sturmsche Kette*] of functions. For the definition of such a sequence see, *e.g.,* page 135 of [SRS68].

Since the characteristic polynomials are monic, it follows from (9.181)–(9.182) that

$$\boxed{\chi_n(t) = \gamma_0 \cdots \gamma_{n-1} p_n(t) = \beta_0^{\mathsf{L}} \cdots \beta_{n-1}^{\mathsf{L}} p_n^{\mathsf{L}}(t)\,.} \tag{9.190}$$

Note that the inequalities in (9.189) are strict. If we allow equality signs, the assumptions can be relaxed to matrices that are not tridiagonal. The result goes then under the name of Cauchy's interlacing theorem see, *e.g.,* [Par80].

So far in this section, we have only assumed that $\mathbf{A}$ is Hermitian (or real symmetric). For applying the CG method we normally assume that $\mathbf{A}$ is Hpd (or spd). Then the matrices $\mathbf{T}_m$ are spd too, because by (9.170), for $\mathbf{z} \neq \mathbf{o}$,

$$\mathbf{z}^\star \mathbf{T}_m \mathbf{z} = \mathbf{z}^\star \mathbf{Y}_m^\star \mathbf{A} \mathbf{Y}_m \mathbf{z} = (\mathbf{Y}_m \mathbf{z})^\star \mathbf{A} (\mathbf{Y}_m \mathbf{z}) > 0\,.$$

LEMMA 9.7.4. *If $\mathbf{A}$ is Hpd (or spd), then, for $m = 1, \ldots, \bar{\nu}$, the tridiagonal matrices $\mathbf{T}_m$ generated by the Lanczos algorithm are real and spd, and thus the square part $\mathbf{T}_m^\circ$ of the matrices $\underline{\mathbf{T}}_m^\circ$ generated by the* ORES *form of the conjugate gradient method, which are diagonally similar to $\mathbf{T}_m$, have real positive eigenvalues.*

The risk when applying the CG method to an indefinite Hermitian matrix is that, even if the matrix is nonsingular, some $\mathbf{T}_m$ may be singular, so that $p_m^{\mathsf{L}}(0) = 0$. But then it is impossible to normalize p_m so that $p_m(0) = 1$; see (9.178). How could we fix this?

9.7.2 Eigenvalue Computations With the Symmetric Lanczos Process*

Comparing the basic matrix identities of the symmetric Lanczos process, namely

$$\mathbf{A}\mathbf{Y}_m = \mathbf{Y}_{m+1} \underline{\mathbf{T}}_m\,, \qquad \mathbf{Y}_m^\star \mathbf{Y}_m = \mathbf{I}_m\,, \qquad \mathbf{Y}_m^\star \mathbf{A} \mathbf{Y}_m = \mathbf{T}_m \tag{9.191}$$

with the assumptions of the Rayleigh–Ritz procedure for symmetric matrices discussed in [Par98], we see that the symmetric tridiagonal matrix $\mathbf{T}_m$ represents the restricted orthogonal projection $\mathbf{A}\big|_{\mathcal{K}_m}$ of $\mathbf{A}$ into the subspace $\mathcal{K}_m$, when this projection is expressed in terms of the basis consisting of the columns of $\mathbf{Y}_m$. In particular, $\mathbf{T}_m$ is the matrix Rayleigh quotient of $\mathbf{Y}_m$, and the Rayleigh residual matrix is just

$$\mathbf{R}(\mathbf{Y}_m;\ \mathbf{A}) = \mathbf{A}\mathbf{Y}_m - \mathbf{Y}_m \mathbf{T}_m = \mathbf{y}_m \beta_m^{\mathsf{L}}\, \mathbf{l}_m^{\mathsf{T}}\,, \tag{9.192}$$

where $\mathbf{l}_m$ is the last row of the $m \times m$ unit matrix, as before.

Therefore, the symmetric Lanczos algorithm is a tool for constructing the basis of a subspace and the corresponding matrix Rayleigh quotient $\mathbf{H} = \mathbf{T}_m$. The remaining steps of the Rayleigh–Ritz procedure are readily adapted. In addition to the error bounds from the general Rayleigh–Ritz theory there are more specific ones for eigenvalue computations based on the symmetric Lanczos algorithm; see [Par98].

The symmetric Lanczos algorithm is in fact widely used for computing eigenvalue approximations of large sparse symmetric (or Hermitian) matrices. There are, however, a number of difficulties in its implementation. Rounding errors in finite precision arithmetic cause a loss of orthogonality of the Lanczos vectors $\mathbf{y}_n$, the columns of $\mathbf{Y}_m$. This effect can be serious or even disastrous since only the orthogonality of $\mathbf{y}_{n+1}$ with respect to $\mathbf{y}_n$ and $\mathbf{y}_{n-1}$ is explicitly enforced, while the orthogonality with respect to earlier

Lanczos vectors is inherited. It was shown by [Pai71] that this roundoff error propagation is governed by certain rules, which tell us that devastating errors only occur when the last row of the Rayleigh residual matrix, $\mathbf{y}_m \beta_m^{\mathrm{L}}$, is very small. Paige's theory is also covered in [Par80]. It lead to a number of proposals to extend the explicit orthogonalization to certain other vectors (**partial reorthogonalization**, **selective reorthogonalization**). Without this extra work for reorthogonalization, $\mathbf{T}_m$ will soon contain multiple copies of extremal eigenvalues, even when these are simple eigenvalues of $\mathbf{A}$. Then extra work is needed to distinguish these **ghost eigenvalues** from the true ones; this approach has been followed by [CW85].

As mentioned at the beginning of Section 9.7.1, the symmetric Lanczos algorithm is the adaptation of a more general algorithm for nonsymmetric problems suggested by [Lan50]. That algorithm generates a pair of biorthogonal bases $\{\mathbf{y}_n\}$, $\{\widetilde{\mathbf{y}}_n\}$, for which $\langle \widetilde{\mathbf{y}}_m, \mathbf{y}_n \rangle = \delta_{m,n}$. Today, the eigenvalue approximations are extracted in a different way from these algorithms than was described by Lanczos, normally, by applying the QR algorithm to $\mathbf{T}_m$ for sufficiently large m.

Lanczos also mentioned the applicability of his approach to the problem of solving a sparse linear system of equations, and he worked this out in [Lan52]. He called it *method of minimized iterations*, but it became known as the **biconjugate gradient (BiCG) method** [*Methode der bikonjugierten Gradienten*]. However, again, an important detail of this solution technique has changed, namely how the approximations $\mathbf{x}_n$ are found. In [Fle76] (and in earlier Russian work) it was pointed out that Lanczos's approach can be applied in a form completely analogous to the CG algorithm Hestenes and Stiefel had suggested for spd systems — what we called the OMin form of CG (following [AMS90]). The natural name for this newer, standard form of BiCG is therefore BiOMin.

So, in some sense, Lanczos generalized CG before it was published by [HS52], and he seems to have developed his method independently, although he was working at the same institute as Hestenes, the Institute of Numerical Analysis (INA) of the National Bureau of Standard (NBS) on the campus of UCLA. But the Hestenes–Stiefel paper was much better worked out, both with regard to the theory and the details of the implementation of the method.

9.7.3 Solving the System in Coordinate Space*

The symmetric Lanczos process (Algorithm 9.10) produces a series of nested orthonormal bases for the Krylov subspaces $\mathcal{K}_n(\mathbf{A}, \mathbf{y}_0)$, $n = 1, 2, \ldots$. When solving $\mathbf{Ax} = \mathbf{b}$, we assume that $\mathbf{y}_0 := \mathbf{r}_0 / \|\mathbf{r}_0\|$ with $\mathbf{r}_0 = \mathbf{b} - \mathbf{Ax}$, so that $\mathcal{K}_n(\mathbf{A}, \mathbf{y}_0) = \mathcal{K}_n(\mathbf{A}, \mathbf{r}_0)$. We could try to transform the linear system into such a new basis; but if $n < \bar{\nu}$, we can represent $\mathbf{A}$ only approximately in this basis. However, it turns out that we can directly formulate the CG optimality condition in the Krylov subspace, and that we obtain a linear system for the coordinates of the optimal approximation $\mathbf{x}_n$.

To this end, we return to the error norm minimization approach of Section 9.6.1, but we improve it by taking into account the restriction (from the definition of a Krylov space solver)

$$\mathbf{x}_n - \mathbf{x}_0 \in \mathcal{K}_n :\equiv \mathcal{K}_n(\mathbf{A}, \mathbf{r}_0) = \mathsf{span}\ \{\mathbf{y}_0, \ldots, \mathbf{y}_{n-1}\}\ .$$

Here, $\begin{pmatrix} \mathbf{y}_0 & \mathbf{y}_1 & \cdots & \mathbf{y}_{n-1} \end{pmatrix} \equiv: \mathbf{Y}_n$, with $\mathbf{Y}_n^\star \mathbf{Y}_n = \mathbf{I}_n$ and $\mathbf{y}_0 := \mathbf{r}_0/\rho_0$, is now the nth Lanczos basis. We can write, with some coordinate vector $\mathbf{k}_n$,

$$\boxed{\mathbf{x}_n = \mathbf{x}_0 + \mathbf{Y}_n \mathbf{k}_n\,, \qquad \mathbf{r}_n = \mathbf{r}_0 - \mathbf{A}\mathbf{Y}_n \mathbf{k}_n\,.} \tag{9.193}$$

When $\mathbf{A}$ is spd or Hpd, minimization of half the square of the $\mathbf{A}$-norm of the nth error $\mathbf{d}_n$ subject to the restriction means minimization of the quadratic functional

$$\boxed{\Phi_n(\mathbf{k}_n) :\equiv \tfrac{1}{2}\|\mathbf{d}_n\|_{\mathbf{A}}^2 = \tfrac{1}{2}\|\mathbf{x}_n - \mathbf{x}_\star\|_{\mathbf{A}}^2 = \tfrac{1}{2}\|\mathbf{x}_0 - \mathbf{x}_\star + \mathbf{Y}_n \mathbf{k}_n\|_{\mathbf{A}}^2} \tag{9.194}$$

over $\mathbf{k}_n \in \mathbb{E}_n$, where $\mathbb{E}_n$ is the coordinate space $\mathbb{R}^n$ or $\mathbb{C}^n$. Writing the right-hand side of (9.194) as an inner product and multiplying out we obtain

$$\Phi_n(\mathbf{k}_n) = \tfrac{1}{2}\|\mathbf{x}_0 - \mathbf{x}_\star\|_{\mathbf{A}}^2 + \mathsf{Re}\ \langle \mathbf{x}_0 - \mathbf{x}_\star, \mathbf{A}\mathbf{Y}_n\mathbf{k}_n \rangle + \tfrac{1}{2}\ \langle \mathbf{Y}_n\mathbf{k}_n, \mathbf{A}\mathbf{Y}_n\mathbf{k}_n \rangle\ , \tag{9.195}$$

from which one can derive that

$$\boxed{\nabla\Phi_n(\mathbf{k}_n) = \mathbf{Y}_n^\star \mathbf{A}(\mathbf{x}_0 - \mathbf{x}_\star + \mathbf{Y}_n\mathbf{k}_n) = -\,\mathbf{Y}_n^\star \mathbf{r}_n\,.} \tag{9.196}$$

Since $\mathbf{Y}_n^\mathsf{T}\mathbf{A}\mathbf{Y}_n$ is Hpd, the quadratic function Φ_n has only one stationary point, and this is a minimum. It is characterized by the **Galerkin condition** [*Galerkin-Bedingung*]

$$\boxed{\mathbf{r}_n^\star \mathbf{Y}_n = \mathbf{o}^\mathsf{T}} \qquad \text{or} \qquad \boxed{\mathbf{r}_n \perp \mathcal{K}_n\,.} \tag{9.197}$$

This condition, which we have found before under different assumptions in Lemma 9.6.3, implies that the residuals are mutually orthogonal. We can formulate the following results:

THEOREM 9.7.5. *Let $\mathbf{A}$ be spd or Hpd. A Krylov space solver for $\mathbf{A}\mathbf{x} = \mathbf{b}$ achieves minimization in the energy norm (i.e., in the $\mathbf{A}$-norm of the error) in each step if and only if the Galerkin condition* (9.197) *holds for $n = 0, \dots, \bar{\nu}$, or, in other words, if and only if the residuals are mutually orthogonal.*

Now note that $\mathbf{r}_0 = \mathbf{Y}_n\mathbf{e}_1\rho_0$ if $\mathbf{e}_1$ denotes the first column of the $n \times n$ unit matrix. So, from $\mathbf{r}_n^\star\mathbf{Y}_n = \mathbf{o}^\mathsf{T}$, $\mathbf{Y}_n^\star\mathbf{Y}_n = \mathbf{I}_n$, and (9.193) we can conclude that

$$\mathbf{o} = \mathbf{Y}_n^\star\mathbf{r}_n = \mathbf{Y}_n^\star(\mathbf{r}_0 - \mathbf{A}\mathbf{Y}_n\mathbf{k}_n) = \mathbf{e}_1\rho_0 - \mathbf{Y}_n^\star\mathbf{A}\mathbf{Y}_n\mathbf{k}_n\,,$$

and, using (9.170), that

$$\boxed{\mathbf{T}_n\mathbf{k}_n = \mathbf{e}_1\rho_0\,.} \tag{9.198}$$

This result can summarized as follows:

THEOREM 9.7.6. *When a real symmetric or Hermitian system $\mathbf{A}\mathbf{x} = \mathbf{b}$ is treated with any Krylov space solver characterized by the Galerkin condition* (9.197) *(such as, in particular, the* CG *method) the coordinates $\mathbf{k}_n$ of $\mathbf{x}_n - \mathbf{x}_0$ with respect to the Lanczos basis are solutions of the real symmetric tridiagonal $n \times n$ system* (9.198)*, which itself results from the Galerkin condition* (9.197)*.*

This result gives us further insight on why the CG algorithms are limited to positive definite (or negative definite) matrices.

COROLLARY 9.7.7. *When the Krylov space solver addressed in Theorem 9.7.6 is applied to a real symmetric or Hermitian indefinite system, some approximations $\mathbf{x}_n$ may be undefined because the tridiagonal matrix $\mathbf{T}_n$ can be singular — in which case*

- *the nth Lanczos polynomial p_n^{L} has a zero at the origin: $p_n^{\mathrm{L}}(0) = 0$,*
- *a residual polynomial p_n satisfying the consistency condition $p_n(0) = 1$ does not exist, and*
- *the system* (9.198) *has no solution.*

But if $\mathbf{k}_n$ and $\mathbf{x}_n$ are undefined, then $\mathbf{k}_{n+1}$ and $\mathbf{x}_{n+1}$ are again defined and can be computed according to Theorem 9.7.6.

PROOF. We have seen above that p_n^{L} is up to scaling the characteristic polynomial of $\mathbf{T}_n$. Hence, $p_n^{\mathrm{L}}(0) = 0$ if and only if $\mathbf{T}_n$ is singular. Clearly, in view of (9.178), a normalized residual polynomial cannot exist if $p_n^{\mathrm{L}}(0) = 0$.

If the singular system (9.198) had a solution $\mathbf{k}_n$, it would define an approximate solution $\mathbf{x}_n$ whose residual would satisfy the Galerkin condition (9.197) and whose residual polynomial would satisfy the consistency condition, in contradiction to the above.

The fact that $\mathbf{k}_{n+1}$ is again well defined follows from the interlacing of the zeros of p_n^{L} and p_{n+1}^{L}; see Theorem 9.7.2. □

The fact that at least every other $\mathbf{x}_n$ is well defined has been used to derive CG-like algorithms that do an implicit double step when $\mathbf{T}_n$ is singular or near-singular.

The conjugate residual method can be formulated in an analogous way. For a nonsingular square matrix, also for a non-Hermitian one, minimization of half the square of the 2–norm of the residual under the restriction $\mathbf{x}_n - \mathbf{x}_0 \in \mathcal{K}_n$ means minimization of

$$\boxed{\widehat{\Phi}_n(\mathbf{k}_n) :\equiv \tfrac{1}{2}\|\mathbf{r}_n\|^2 = \tfrac{1}{2}\|\mathbf{r}_0 - \mathbf{A}\mathbf{Y}_n\mathbf{k}_n\|^2} \tag{9.199}$$

over $\mathbf{k}_n \in \mathbb{E}_n$. The gradient of $\widehat{\Phi}_n$ can be seen to be

$$\boxed{\nabla\widehat{\Phi}_n(\mathbf{k}_n) = -\,\mathbf{Y}_n^\star\mathbf{A}^\star\mathbf{r}_n\,.} \tag{9.200}$$

Since the matrix $\mathbf{Y}_n^\star \mathbf{A}^\star \mathbf{A} \mathbf{Y}_n$ of the quadratic function $\widehat{\Phi}_n$ is Hpd, the minimum is characterized by the Galerkin condition

$$\boxed{\mathbf{r}_n^\star \mathbf{A} \mathbf{Y}_n = \mathbf{o}^\mathsf{T}} \qquad \text{or} \qquad \boxed{\mathbf{r}_n \perp \mathbf{A}\mathcal{K}_n\,.} \tag{9.201}$$

It implies that the residuals are (in general only formally) $\mathbf{A}$-orthogonal. For true $\mathbf{A}$-orthogonality we need that $\mathbf{A}$ is spd or Hpd, which is usually the case when the CR method is applied, but has not been assumed in the derivation of this Galerkin condition.

THEOREM 9.7.8. *A Krylov space solver for $\mathbf{A}\mathbf{x} = \mathbf{b}$ achieves minimization in the 2-norm of the residual in each step if and only if the Galerkin condition* (9.201) *holds for $n = 0, \dots, \bar{\nu}$, or, in other words, if and only if the residuals are mutually conjugate.*

Next we use the representation $\mathbf{r}_n = \mathbf{r}_0 - \mathbf{A}\mathbf{Y}_n \mathbf{k}_n$ from (9.193) and insert

$$\mathbf{r}_0 = \mathbf{y}_0 \rho_0 = \mathbf{Y}_{n+1}\, \underline{\mathbf{e}}_1\, \rho_0\,,$$

with $\underline{\mathbf{e}}_1$ the first column of the $(n+1) \times (n+1)$ unit matrix, as well as the matrix representation (9.168),

$$\mathbf{A}\mathbf{Y}_n = \mathbf{Y}_{n+1} \underline{\mathbf{T}}_n$$

of the symmetric Lanczos process. We obtain the fundamental relation

$$\boxed{\mathbf{r}_n = \mathbf{Y}_{n+1}\left(\underline{\mathbf{e}}_1 \rho_0 - \underline{\mathbf{T}}_n \mathbf{k}_n\right).} \tag{9.202}$$

Here, in contrast to (9.193) we do not only represent $\mathbf{x}_n - \mathbf{x}_0$ and $\mathbf{r}_n - \mathbf{r}_0$ in the Lanczos basis, but directly $\mathbf{r}_n$. The corresponding coordinate vector

$$\boxed{\mathbf{q}_n :\equiv \underline{\mathbf{e}}_1 \rho_0 - \underline{\mathbf{T}}_n \mathbf{k}_n} \tag{9.203}$$

is called the **quasi-residual** [*Quasi-Residuum*].

Since $\mathbf{Y}_{n+1}$ has orthonormal columns, minimizing $\mathbf{r}_n$ is equivalent to minimizing its coordinate vector:

$$\boxed{\|\mathbf{r}_n\|^2 = \mathbf{q}_n^\star \mathbf{Y}_{n+1}^\star \mathbf{Y}_{n+1} \mathbf{q}_n = \|\mathbf{q}_n\|^2\,.} \tag{9.204}$$

In view of the definition (9.203) minimizing $\|\mathbf{q}_n\|^2$ is a $(n+1) \times n$ real least-squares problem for the unknown $\mathbf{k}_n$, whose matrix is the extended real tridiagonal $\underline{\mathbf{T}}_n$.

The above derived result can be summarized as follows:

THEOREM 9.7.9. *When a real symmetric or Hermitian system $\mathbf{A}\mathbf{x} = \mathbf{b}$ is solved by any Krylov space solver that minimizes the 2-norm of the residual (such as, in particular, the* CR *method), the coordinates $\mathbf{k}_n$ of $\mathbf{x}_n - \mathbf{x}_0$ with respect to the Lanczos basis are solutions of the real tridiagonal $(n+1) \times n$ least squares problem*

$$\boxed{\min_{\mathbf{k}_n \in \mathbb{E}^n} \|\mathbf{q}_n\|^2 = \min_{\mathbf{k}_n \in \mathbb{E}^n} \left\|\underline{\mathbf{e}}_1 \rho_0 - \underline{\mathbf{T}}_n \mathbf{k}_n\right\|^2.} \tag{9.205}$$

A least squares problem always has a solution, but this solution is only unique if the matrix has full rank. Here, this is the case:

THEOREM 9.7.10. *In the least squares problem* (9.205) *the matrix $\underline{\mathbf{T}}_n$ resulting from the symmetric Lanczos process has full rank n, and therefore the problem has a unique solution $\mathbf{k}_n$.*

PROOF. In the notation (9.167) for $\underline{\mathbf{T}}_n$ we have by construction (see Algorithm 9.10) either $\beta_k^\mathrm{L} \neq 0$, $k = 0, 1, \dots, n-1$, or $\beta_k^\mathrm{L} \neq 0$, $k = 0, 1, \dots, n-2$ and $\beta_{n-1}^\mathrm{L} = 0$. In the first case, the last n rows of $\underline{\mathbf{T}}_n$ form a nonsingular upper triangular matrix, so $\underline{\mathbf{T}}_n$ has rank n. In the second case the columns of $\mathbf{Y}_n$ span an invariant subspace of $\mathbf{A}$, and $\mathbf{T}_n$ represents $\mathbf{A}$ in this subspace; in particular any eigenvalue of $\mathbf{T}_n$ is an eigenvalue of $\mathbf{A}$, which is assumed to be nonsingular. So, $\mathbf{T}_n$ is also nonsingular, and thus $\underline{\mathbf{T}}_n$ has rank n. □

We could solve the sequence of least squares problems (9.205) by solving for each n the normal equations

$$\underline{\mathbf{T}}_n^\mathsf{T} \underline{\mathbf{T}}_n \mathbf{k}_n = \underline{\mathbf{T}}_n^\mathsf{T} \mathbf{e}_1 \rho_0 = \begin{pmatrix} \alpha_0^\mathrm{L} & \beta_0^\mathrm{L} & 0 & 0 & \dots \end{pmatrix}^\mathsf{T} \rho_0$$

or, preferably, by deriving an update procedure for this. But efficiency and accuracy command to use instead a well-known special update procedure for the QR or the LQ decomposition of $\underline{\mathbf{T}}_n$.

If $\mathbf{A}$ is real symmetric, the successive solution of (9.205) by updating in each step the LQ or QR decomposition of $\underline{\mathbf{T}}_n$ leads to the MINRES **algorithm** of [PS75]. With a minor modification MINRES is also applicable to Hermitian $\mathbf{A}$, and a straightforward generalization leads to the GMRES **algorithm** of [SS85] for non-Hermitian (or real non-symmetric) $\mathbf{A}$, one of the most widely used Krylov space solvers for this case. In this situation the tridiagonal real matrix $\underline{\mathbf{T}}_n$ is replaced by an upper Hessenberg matrix $\underline{\mathbf{H}}_n$ of the same size, which is in general complex if $\mathbf{A}$ is.

The **quasi-minimal residual (**QMR**) method** of [FN91], which is also for the non-Hermitian case but works with a tridiagonal $\underline{\mathbf{T}}_n$ again, follows the same pattern, except that it does not use an orthogonal basis, and hence only minimizes the 2–norm of the quasi-residual, not the one of the residual itself.

9.7.4 Further Topics Related to the CG Method and the Lanczos Process*

There are many more interesting topics in connection with the CG method:

- CG and CR as orthogonal projections.
- The connection to **orthogonal polynomials**.
- The connection to **Gauss quadrature**.
- A stopping criterion based on the energy norm.
- The connection to **Padé approximation** (high order rational interpolation at one point) and **continued fractions** [*Kettenbrüchen*].
- CG *in finite precision arithmetic*: the effects of roundoff (in particular, the "loss of orthogonality") on the speed of convergence.
- CG *in finite precision arithmetic*: the effects of roundoff (in particular, the "gap between the recursively updated and the true residual") on the ultimate accuracy. ($\rightsquigarrow$ OMIN may be far better than ORES or ODIR.)
- The connections between CG and CR as well as CGNE and CGNR.

The CG and CR methods, including the MINRES version of the latter, have also been an important source of ideas for creating iterative method for nonsymmetric linear systems. In the years $\approx$ 1970–1995 many such methods have been proposed.

9.8 Solving the System in Coordinate Space

In this chapter, we return to the approach outlined in Section 9.7.3, where we transformed the linear system into coordinate space by first constructing a nested basis of the nested Krylov subspaces. While we used there the symmetric Lanczos process to generate the basis, we will now allow for non-Hermitian matrices and apply the Arnoldi process. In this setting the most effective way to solve the system in coordinate space is the GMRES algorithm of [SS85]. The symmetric Lanczos process can be viewed as a special case of the Arnoldi process, and GMRES is a quite straightforward generalization of the MINRES algorithm of [PS75], which is restricted to Hermitian and real symmetric matrices, but is much more economical than GMRES. We will also encounter a few other related iterative methods.

9.8.1 The Arnoldi Process

By definition of the Krylov space $\mathcal{K}_n(\mathbf{A}, \mathbf{y})$ and by Lemma 9.3.1 the vectors $\mathbf{y}$, $\mathbf{A}\mathbf{y}$, ..., $\mathbf{A}^{n-1}\mathbf{y}$ form a basis of $\mathcal{K}_n(\mathbf{A}, \mathbf{y})$ as long as $n \leq \bar{\nu}(\mathbf{y}, \mathbf{A})$. However, as mentioned before, see page 106, this so-called **Krylov basis** is typically very ill-conditioned.

To construct a better basis, we could apply Gram-Schmidt orthogonalization to the Krylov basis. But as suggested by [Lan50] and [Arn51] it is is far better to combine the Gram-Schmidt process directly with the generation of the basis, as we did in the symmetric Lanczos process of Section 9.7.1. Here is a first version of the **Arnoldi algorithm** (or **Arnoldi process** [*Arnoldi-Prozess*]) using classical Gram-Schmidt (CGS) orthonormalization.

Algorithm 9.11 (ARNOLDI ALGORITHM BASED ON CGS). .
Let a nonsingular matrix $\mathbf{A}$ *and a nonzero vector* $\mathbf{y}$ *be given. For constructing a nested set of orthonormal bases* $\{\mathbf{y}_0, \mathbf{y}_1, \dots, \mathbf{y}_m\}$ *for the nested Krylov subspaces* $\mathcal{K}_{m+1}(\mathbf{A}, \mathbf{y})$ $(m = 0, \dots, \bar{\nu}(\mathbf{y}, \mathbf{A}) - 1)$ *we let* $\eta_0 := \|\mathbf{y}\|$, $\mathbf{y}_0 := \mathbf{y}/\eta_0$ *and compute, for* $n = 0, 1, \dots, m-1$,

$$\left.\begin{array}{rcl} \widetilde{\mathbf{y}} & := & (\mathbf{A}\mathbf{y}_n - \mathbf{y}_n\, \eta_{n,n} - \cdots - \mathbf{y}_0\, \eta_{0,n})\ , \\ \mathbf{y}_{n+1} & := & \widetilde{\mathbf{y}}/\eta_{n+1,n}\ , \end{array}\right\} \tag{9.206}$$

where the coefficients $\eta_{0,n}$, $\eta_{1,n}$, $\dots$, $\eta_{n,n}$ *are chosen to make* $\widetilde{\mathbf{y}}$ *orthogonal to* $\mathbf{y}_0$, $\mathbf{y}_1$, $\dots, \mathbf{y}_n$, *while* $\eta_{n+1,n}$ *is used to normalize* $\widetilde{\mathbf{y}}$*:*

$$\eta_{k,n} :\equiv \langle \mathbf{y}_k, \mathbf{A}\mathbf{y}_n \rangle \quad (k = 0, \dots, n), \qquad \eta_{n+1,n} :\equiv \|\widetilde{\mathbf{y}}\|\,. \tag{9.207}$$

When $n = m - 1 = \bar{\nu} - 1$, *then* $\widetilde{\mathbf{y}} = \mathbf{o}$ *and the process terminates.*

However, it is well known that the classical Gram-Schmidt process may produce a basis whose orthogonality is inaccurate due to large roundoff errors. Therefore, in practice, the modified Gram-Schmidt (MGS) algorithm should be applied instead of the classical one, as in the following second version of the Arnoldi algorithm.

Algorithm 9.12 (ARNOLDI ALGORITHM BASED ON MGS). .
Let a nonsingular matrix $\mathbf{A}$ *and a nonzero vector* $\mathbf{y}$ *be given. For constructing a nested set of orthonormal bases* $\{\mathbf{y}_0, \mathbf{y}_1, \dots, \mathbf{y}_m\}$ *for the nested Krylov subspaces* $\mathcal{K}_{m+1}(\mathbf{A}, \mathbf{y})$ $(m = 1, \dots, \bar{\nu}(\mathbf{y}, \mathbf{A}) - 1)$ *we let* $\mathbf{y}_0 := \mathbf{y}/\eta_0$ *and compute, for* $n = 0, 1, \dots, m-1$,

$$\left.\begin{array}{rclrcl} \widetilde{\mathbf{y}} & := & \mathbf{A}\mathbf{y}_n & & & \\ \widetilde{\mathbf{y}} & := & \widetilde{\mathbf{y}} - \mathbf{y}_k\, \eta_{k,n}\ , & \eta_{k,n} & :\equiv & \langle \mathbf{y}_k, \widetilde{\mathbf{y}} \rangle \\ & & & (k = n, n & - & 1, \dots, 0), \\ \mathbf{y}_{n+1} & := & \widetilde{\mathbf{y}}/\eta_{n+1,n}\ , & \eta_{n+1,n} & :\equiv & \|\widetilde{\mathbf{y}}\|\,. \end{array}\right\} \tag{9.208}$$

When $n = m - 1 = \bar{\nu} - 1$, *then* $\widetilde{\mathbf{y}} = \mathbf{o}$ *and the process terminates.*

The orthonormal basis of $\mathcal{K}_m$ $(m \leq \bar{\nu})$ that is generated here is called the **Arnoldi basis** [*Arnoldi-Basis*].

Clearly, in exact arithmetic, the Arnoldi process will terminate with $\eta_{\bar{\nu},\bar{\nu}-1} = 0$ when $n = m - 1 = \bar{\nu} - 1$ since the Krylov space generated from $\mathbf{y}$ is then exhausted. We therefore define

$$\mathbf{y}_{\bar{\nu}} :\equiv \mathbf{o}\,. \tag{9.209}$$

This is useful for theoretical considerations. In practice $\eta_{\bar{\nu},\bar{\nu}-1}$ may be far from small due to roundoff errors. But, typically, m is limited to values much smaller than $\bar{\nu}$.

An obvious disadvantage of the Arnoldi process is that the whole basis must be stored and that at each step all the vectors that have been generated before must be retrieved.

There is, as we know, an important exception: when the matrix is real symmetric or (complex) Hermitian, the long recursion of (9.206) reduces to a three-term recursion, and the Arnoldi process becomes the symmetric Lanczos process:

LEMMA 9.8.1. *If* $\mathbf{A} = \mathbf{A}^\star$*, then in* (9.206) *with* (9.207) *we have*

$$\begin{aligned} \eta_{k,n} &= 0\,, \qquad k = 0, 1, \dots, n-2, && (9.210)\\ \eta_{n-1,n} &= \overline{\eta_{n,n-1}}\,, && (9.211) \end{aligned}$$

so that if we let $\beta^{\mathrm{L}}_{-1} :\equiv 0$ *and*

$$\begin{aligned} \alpha^{\mathrm{L}}_n &:\equiv \eta_{n,n} = \langle \mathbf{y}_n, \mathbf{A}\mathbf{y}_n \rangle\,, && (9.212a)\\ \beta^{\mathrm{L}}_n &:\equiv \eta_{n+1,n} = \| \mathbf{A}\mathbf{y}_n - \mathbf{y}_n\, \alpha^{\mathrm{L}}_n - \mathbf{y}_{n-1}\, \beta^{\mathrm{L}}_{n-1} \|\,, && (9.212b) \end{aligned}$$

the recursion (9.206) *reduces to*

$$\mathbf{y}_{n+1} := \left(\mathbf{A}\mathbf{y}_n - \mathbf{y}_n\, \alpha^{\mathrm{L}}_n - \mathbf{y}_{n-1}\, \beta^{\mathrm{L}}_{n-1} \right) / \beta^{\mathrm{L}}_n\,, \qquad n = 0, 1, \dots, \bar{\nu} - 2\,, \qquad (9.213)$$

with $\alpha^{\mathrm{L}}_n \in \mathbb{R}$ $(n = 0, \dots, \bar{\nu} - 1)$ *and* $\beta^{\mathrm{L}}_n > 0$ $(n = 0, \dots, \bar{\nu} - 2)$.
In other words, the Arnoldi process reduces then to the symmetric Lanczos process of Algorithm 9.10.

PROOF. By construction we have $\mathbf{y}_n \perp \mathbf{y}_i$ for $i = 0, 1, \dots, n-1$ and $\|\mathbf{y}_j\| = 1$ for $j = 0, 1, \dots, n$. Therefore, if $k < n-1$ we can conclude from (9.206) that

$$\begin{aligned} \eta_{k,n} &= \langle \mathbf{y}_k, \mathbf{A}\mathbf{y}_n \rangle = \langle \mathbf{A}\mathbf{y}_k, \mathbf{y}_n \rangle \\ &= \langle \mathbf{y}_{k+1}\eta_{k+1,k} + \mathbf{y}_k \eta_{k,k} + \cdots + \mathbf{y}_0 \eta_{0,k}\;,\; \mathbf{y}_n \rangle \\ &= 0\,. \end{aligned}$$

Moreover,

$$\begin{aligned} &\alpha^{\mathrm{L}}_n = \eta_{n,n} = \langle \mathbf{y}_n, \mathbf{A}\mathbf{y}_n \rangle = \langle \mathbf{A}^\star \mathbf{y}_n, \mathbf{y}_n \rangle = \langle \mathbf{A}\mathbf{y}_n, \mathbf{y}_n \rangle = \overline{\langle \mathbf{y}_n, \mathbf{A}\mathbf{y}_n \rangle} = \overline{\alpha^{\mathrm{L}}_n}\,,\\ &\beta^{\mathrm{L}}_n = \eta_{n+1,n} = \| \mathbf{A}\mathbf{y}_n - \mathbf{y}_n\, \alpha^{\mathrm{L}}_n - \mathbf{y}_{n-1}\, \beta^{\mathrm{L}}_{n-1} \| > 0 \end{aligned}$$

except when $n = \bar{\nu} - 1$ where $\beta^{\mathrm{L}}_{\bar{\nu}-1} = 0$. Finally,

$$\begin{aligned} \eta_{n-1,n} &= \langle \mathbf{y}_{n-1}, \mathbf{A}\mathbf{y}_n \rangle = \langle \mathbf{A}\mathbf{y}_{n-1}, \mathbf{y}_n \rangle \\ &= \langle \mathbf{y}_n \eta_{n,n-1} + \mathbf{y}_{n-1} \eta_{n-1,n-1} + \cdots + \mathbf{y}_0 \eta_{0,n-1}\;,\; \mathbf{y}_n \rangle \\ &= \langle \mathbf{y}_n, \mathbf{y}_n \rangle\; \overline{\eta_{n,n-1}} \\ &= \overline{\eta_{n,n-1}}\,. \end{aligned}$$

□

In Chapter 9.6 we made heavy use of the fact that the recursions of the various CG algorithms as well as those of the symmetric Lanczos process can be written compactly in matrix notation. In the following we will often use analogue notation for the Arnoldi and further similar recursions. We define again the $N \times m$ matrix

$$\mathbf{Y}_m :\equiv \begin{pmatrix} \mathbf{y}_0 & \mathbf{y}_1 & \cdots & \mathbf{y}_{m-1} \end{pmatrix} \qquad (9.214)$$

and gather the coefficients of m steps of the recursion (9.206) in an **extended** Hessenberg matrix of size $(m+1) \times m$:

$$\underline{\mathbf{H}}_m :\equiv \begin{pmatrix} \eta_{0,0} & \eta_{0,1} & \cdots & \eta_{0,m-1} \\ \eta_{1,0} & \eta_{1,1} & \cdots & \eta_{1,m-1} \\ & \eta_{2,1} & \ddots & \vdots \\ & & \ddots & \eta_{m-1,m-1} \\ & & & \eta_{m,m-1} \end{pmatrix}. \qquad (9.215)$$

$\underline{\mathbf{H}}_m$ can be partitioned into

$$\underline{\mathbf{H}}_m \equiv: \left(\begin{array}{c} \mathbf{H}_m \\ \hline \eta_{m,m-1}\mathbf{l}_m^{\mathsf{T}} \end{array} \right), \qquad (9.216)$$

where $\mathbf{l}_m^\mathsf{T}$ is the last row of the $m \times m$ unit matrix $\mathbf{I}_m$ and $\mathbf{H}_m$ is square. If $m > n$ recursion (9.206) can be written as

$$\mathbf{A}\mathbf{y}_n = \underbrace{\begin{pmatrix} \mathbf{y}_0 & \cdots & \mathbf{y}_n & \mathbf{y}_{n+1} & \mathbf{y}_{n+2} & \cdots & \mathbf{y}_m \end{pmatrix}}_{= \mathbf{Y}_{m+1}} \begin{pmatrix} \eta_{0,n} \\ \vdots \\ \eta_{n,n} \\ \eta_{n+1,n} \\ 0 \\ \vdots \\ 0 \end{pmatrix} .$$

So we can summarize these recursions for $n = 0, \ldots, m-1$ as

$$\boxed{\mathbf{A}\mathbf{Y}_m = \mathbf{Y}_{m+1}\underline{\mathbf{H}}_m \, .} \tag{9.217}$$

This is often referred to as **Arnoldi relation**. In the Arnoldi process, the matrices $\mathbf{Y}_m$ have orthonormal columns, but this has not been used in the derivation of (9.217). In the symmetric Lanczos process the matrix $\underline{\mathbf{H}}_m$ reduces to the real tridiagonal matrix $\underline{\mathbf{T}}_m$ of (9.167) whose square part $\mathbf{T}_m$ is real symmetric — even if $\mathbf{A}$ is (complex) Hermitian.

By (9.217), the image of the restriction $\mathbf{A}\big|_{\mathcal{K}_m}$ to the subspace $\mathcal{K}_m$ of the linear mapping (or, operator) defined by $\mathbf{A}$ is contained in $\mathcal{K}_{m+1}$. With respect to the bases $\mathbf{y}_0, \ldots, \mathbf{y}_{m-1}$ and $\mathbf{y}_0, \ldots, \mathbf{y}_m$ in the domain and the range, respectively, this restricted linear mapping is represented by the $(m+1) \times m$ matrix $\underline{\mathbf{H}}_m$. Let $\mathbf{\Pi}_m$ denote the projection of $\mathcal{K}_{m+1}$ onto $\mathcal{K}_m$ along $\mathbf{y}_m$. If the bases are orthogonal, then so is this projection, but we need not assume this. With $\mathbf{\Pi}_m$ we can project the image $\mathbf{A}\big|_{\mathcal{K}_m}(\mathcal{K}_m)$ into $\mathcal{K}_m$ In terms of the aforementioned bases, $\mathbf{\Pi}_m$ has the simple representation $\begin{pmatrix} \mathbf{I}_m & \big| & \mathbf{o} \end{pmatrix}$, and the self-mapping of $\mathcal{K}_m$ defined by $\mathbf{\Pi}\,\mathbf{A}\big|_{\mathcal{K}_m}$ is just

$$\begin{pmatrix} \mathbf{I}_m & \big| & \mathbf{o} \end{pmatrix} \underline{\mathbf{H}}_m = \mathbf{H}_m \, . \tag{9.218}$$

The square matrix $\mathbf{H}_m$ and, likewise $\mathbf{T}_m$, are therefore often referred to as orthogonal projections of $\mathbf{A}$ into $\mathcal{K}_m$.

Once the Krylov space is exhausted, that is, once $m + 1 = \bar{\nu}$ and $\eta_{\bar{\nu},\bar{\nu}-1} = 0$, $\mathbf{y}_{\bar{\nu}} = \mathbf{o}$ (as defined in (9.209)), identity (9.217) simplifies to

$$\boxed{\mathbf{A}\mathbf{Y}_{\bar{\nu}} = \mathbf{Y}_{\bar{\nu}}\mathbf{H}_{\bar{\nu}} \, .} \tag{9.219}$$

This means that the columns of $\mathbf{Y}_{\bar{\nu}}$ span an invariant subspace of $\mathbf{A}$. In other words, $\mathcal{K}_{\bar{\nu}}$ is an invariant subspace of $\mathbf{A}$, and every eigenvalue of $\mathbf{H}_{\bar{\nu}}$ is also an eigenvalue of $\mathbf{A}$ (but, in general, not vice-versa).

The square matrix $\mathbf{A}$ has a spectrum $\sigma(\mathbf{A}) = \{\lambda\}_{k=1}^N$ consisting of N eigenvalues if the algebraic multiplicity of the eigenvalues is accounted for. If $\eta_{m,m-1}$ is small, one can expect that the spectrum of $\mathbf{H}_m$ approximates in some sense the one of $\mathbf{A}$, although $\mathbf{A}$ has many more eigenvalues than $\mathbf{H}_m$ if $N \gg m$. But in the case of non-Hermitian matrices the connection between the size of $\eta_{m,m-1}$ and the accuracy of the approximate eigenvalues is more complicated than in the Hermitian case treated in [Par98] and mentioned before in Section 9.7.1, even when the underlying basis $\{\mathbf{y}_n\}_{n=0}^{m-1}$ is orthonormal.

To generate a basis of a Krylov subspace, we can also use recursions of the form (9.206) with coefficients other than those of (9.207). In view of Lemma 9.3.1, any such recursion will guarantee that

$$\mathsf{span}\,(\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_{n-1}) = \mathsf{span}\,(\mathbf{y}, \mathbf{A}\mathbf{y}, \ldots, \mathbf{A}^{n-1}\mathbf{y}) = \mathcal{K}_n(\mathbf{A}, \mathbf{y}) \, . \tag{9.220}$$

Of course, the generated basis will no longer be orthonormal.

Let us next consider the relation between the Arnoldi bases, denoted by $\{\mathbf{y}_n\}_{n=0}^{m-1}$, and some other nested Krylov space bases denoted $\{\widehat{\mathbf{y}}_n\}_{n=0}^{m-1}$, which may be neither normalized nor orthogonal. When we express the new bases in the old ones, then, since the Krylov subspaces and their bases are nested, we have relations of the form

$$\widehat{\mathbf{y}}_k = \mathbf{y}_0\sigma_{0,k} + \mathbf{y}_1\sigma_{1,k} + \cdots + \mathbf{y}_k\sigma_{k,k} \, , \qquad k = 0, \ldots, m.$$

Therefore, the transformation matrix $\mathbf{S}_{m+1} = \begin{pmatrix} \sigma_{n,k} \end{pmatrix}_{k=0}^m$, for which $\widehat{\mathbf{Y}}_{m+1} = \mathbf{Y}_{m+1}\mathbf{S}_{m+1}$ holds, is upper triangular. Thus, $\mathbf{S}_m$ with $\widehat{\mathbf{Y}}_m = \mathbf{Y}_m\mathbf{S}_m$ is just the $m \times m$ leading principal minor of $\mathbf{S}_{m+1}$.

So, we have a nested sequence of triangular transformation matrices. When the restricted linear mapping $\mathbf{A}\big|_{\mathcal{K}_m} : \mathcal{K}_m \to \mathcal{K}_{m+1}$ is in the new basis represented by $\widehat{\underline{\mathbf{H}}}_m$, then

$$\boxed{\widehat{\underline{\mathbf{H}}}_m = \mathbf{S}_{m+1}^{-1} \underline{\mathbf{H}}_m \mathbf{S}_m \, .} \tag{9.221}$$

In fact, if $\mathbf{w} = \mathbf{Y}_m \mathbf{k} = \widehat{\mathbf{Y}}_m \widehat{\mathbf{k}}$ and $\mathbf{A}\mathbf{w} = \mathbf{Y}_{m+1}\mathbf{j} = \widehat{\mathbf{Y}}_{m+1}\widehat{\mathbf{j}}$ with $\mathbf{k} = \mathbf{S}_m \widehat{\mathbf{k}}$, $\mathbf{j} = \mathbf{S}_{m+1}\widehat{\mathbf{j}}$, $\mathbf{j} = \underline{\mathbf{H}}_m \mathbf{k}$, and $\widehat{\mathbf{j}} = \widehat{\underline{\mathbf{H}}}_m \widehat{\mathbf{k}}$, then it follows that

$$\widehat{\mathbf{j}} = \mathbf{S}_{m+1}^{-1}\mathbf{j} = \mathbf{S}_{m+1}^{-1}\underline{\mathbf{H}}_m \mathbf{k} = \mathbf{S}_{m+1}^{-1}\underline{\mathbf{H}}_m \mathbf{S}_m \widehat{\mathbf{k}} = \widehat{\underline{\mathbf{H}}}_m \widehat{\mathbf{k}}$$

if $\widehat{\underline{\mathbf{H}}}_m$ satisfies (9.221). So, the following holds:

LEMMA 9.8.2. *Any generation of a nested sequence of bases for a nested sequence of Krylov subspaces can be represented by a matrix identity of the form* (9.217) *with a nested sequence of extended upper Hessenberg matrices* $\underline{\mathbf{H}}_m$ $(m = 1, 2, \ldots, \bar{\nu})$ *with nonzero elements on the first subdiagonal, except for* $\eta_{\bar{\nu},\bar{\nu}-1} = 0$ *if* $m + 1 = \bar{\nu}$.
A basis transformation in such a nested sequence of Krylov subspaces is expressed by a nested sequence of upper triangular matrices $\mathbf{S}_m$ $(m = 1, 2, \ldots)$*. The two sets of Hessenberg matrices describing the generation of the two sets of bases are then related by* (9.221)*.*

A particularly simple special case is when we rescale a basis, that is, just change the length of the basis vectors. In this case, the matrices $\mathbf{S}_m$ are nested diagonal matrices.

Finally we mention that the recursion (9.206) may also be used for generating a sequence of vectors that has more than $\bar{\nu}$ elements. Then, for $n \geq \bar{\nu}$ these vectors will no longer be linearly independent. But also in this case, the matrix identity (9.217) remains valid. An example is Chebyshev iteration.

9.8.2 The Transformation to Coordinate Space

As we have seen in Section 9.7.3, there is the option of designing Krylov space methods that generate a basis for the space independently from computing the residuals or search directions and then specify the iteration by conditions in the coordinate space.

Given $\mathbf{A}$, $\mathbf{x}_0$, and $\mathbf{y} := \mathbf{r}_0 :\equiv \mathbf{b} - \mathbf{A}\mathbf{x}_0$, assume that the vectors $\mathbf{y}_n$, $n = 1, 2, \ldots$, are generated by the Arnoldi process or by some other application of recursion (9.206). Then (9.217) holds after n steps with $m = n$, and the columns of $\mathbf{Y}_m$ form a generating set of $\mathcal{K}_n(\mathbf{A}, \mathbf{r}_0)$. Recall that for a Krylov space method, $\mathbf{x}_n - \mathbf{x}_0 \in \mathcal{K}_n(\mathbf{A}, \mathbf{r}_0)$, so there exists a coordinate vector $\mathbf{k}_n$ such that

$$\boxed{\mathbf{x}_n = \mathbf{x}_0 + \mathbf{Y}_n \mathbf{k}_n \, , \qquad \mathbf{r}_n = \mathbf{r}_0 - \mathbf{A}\mathbf{Y}_n \mathbf{k}_n \, .} \tag{9.222}$$

In view of $\mathbf{y}_0 = \mathbf{r}_0/\rho_0$ with $\rho_0 :\equiv \|\mathbf{r}_0\|$, we find, by inserting (9.217) in the last equation and using

$$\mathbf{r}_0 = \mathbf{y}_0 \rho_0 = \mathbf{Y}_{n+1}\, \underline{\mathbf{e}}_1\, \rho_0 \, , \tag{9.223}$$

with $\underline{\mathbf{e}}_1 :\equiv \begin{pmatrix} 1 & 0 & 0 & \cdots \end{pmatrix}^{\mathsf{T}} \in \mathbb{R}^{n+1}$, that

$$\boxed{\mathbf{r}_n = \mathbf{Y}_{n+1}\left(\underline{\mathbf{e}}_1 \rho_0 - \underline{\mathbf{H}}_n \mathbf{k}_n\right) .} \tag{9.224}$$

If $\mathbf{Y}_{n+1}$ has orthonormal columns, minimizing $\mathbf{r}_n$ is equivalent to minimizing its coordinate vector, the **quasi-residual**

$$\boxed{\mathbf{q}_n :\equiv \underline{\mathbf{e}}_1 \rho_0 - \underline{\mathbf{H}}_n \mathbf{k}_n \, .} \tag{9.225}$$

The only difference to (9.202)–(9.203) is that the tridiagonal matrix $\underline{\mathbf{T}}_n$ from the Lanczos process is replaces by the Hessenberg matrix $\underline{\mathbf{H}}_n$ resulting from the Arnoldi process — if we construct an orthonormal basis — or from another recursion of the form (9.206).

If we are using the Arnoldi basis (or the Lanczos basis if $\mathbf{A}$ is Hermitian), $\|\mathbf{r}_n\| = \|\mathbf{q}_n\|$ because with respect to an orthonormal basis the coordinate map $\mathbf{r}_n \mapsto \mathbf{q}_n$ is an isometry, so we can replace the minimization of $\|\mathbf{r}_n\|$ by the minimization of $\|\mathbf{q}_n\|$. This is the basic idea behind the GMRES algorithm of [SS85] and the preceding MINRES algorithm of [PS75] for Hermitian $\mathbf{A}$. But GMRES requires to store the whole Arnoldi basis. In contrast, in MINRES the basis has not to be stored, as we will see.

9.8.3 GMRES

In this section we assume that the Krylov space is generated by the symmetric Lanczos process if $\mathbf{A}$ is Hermitian, and by the Arnoldi process otherwise. We start from the relations (9.222)–(9.225) and recall that $\|\mathbf{r}_n\| = \|\mathbf{q}_n\|$, so that the minimization of $\|\mathbf{r}_n\|^2$ can be replaced by the minimization of $\|\mathbf{q}_n\|^2$:

$$\boxed{\|\mathbf{r}_n\|^2 = \|\mathbf{q}_n\|^2 = \|\underline{\mathbf{e}}_1\rho_0 - \underline{\mathbf{H}}_n\mathbf{k}_n\|^2 = \min!} \tag{9.226}$$

The obvious advantages of this approach are that the restriction to the Krylov space is taken into account explicitly and that the unknown vector is just an n-vector instead of an N-vector. In fact, since $\underline{\mathbf{H}}_n$ is an $(n+1)\times n$ matrix, minimizing $\|\mathbf{q}_n\|^2$ is an $(n+1)\times n$ least-squares problem. Because we want to solve it for each n, we need a method that allows us to update the solution simply as the dimension increases. But since $\underline{\mathbf{H}}_n$ is upper Hessenberg or tridiagonal, there is indeed a well-known efficient way to update its QR decomposition, based on applying at each step a suitable Givens rotation; details are given below. This is how the GMRES algorithm of [SS85] operates. In the Hermitian case, $\underline{\mathbf{H}}_n$ is symmetric tridiagonal, and thus the QR and the LQ decompositions are equivalent. [PS75] chose to describe their MINRES algorithm in terms of LQ, but we will here use QR also in this case, since this variation of MINRES is essentially a special case of GMRES. (Moreover, the QMR method, which is a modification of the biconjugate gradient (BICG) method for non-Hermitian systems, uses the same technique.)

Let $\underline{\mathbf{H}}_n = \mathbf{Q}_{n+1}\underline{\mathbf{R}}_n^{\mathrm{QR}}$ be a QR decomposition of $\underline{\mathbf{H}}_n$ (with a unitary matrix $\mathbf{Q}_{n+1}$ of order $n+1$). In $\underline{\mathbf{R}}_n^{\mathrm{QR}}$ the marker QR helps us to distinguish this matrix from the matrix $\mathbf{R}_n$ of residual vectors. The last row of this upper triangular $(n+1)\times n$ matrix $\underline{\mathbf{R}}_n^{\mathrm{QR}}$ is zero. We denote its upper square $n\times n$ submatrix by $\mathbf{R}_n^{\mathrm{QR}}$ and define

$$\boxed{\underline{\mathbf{h}}_n :\equiv \left(\frac{\mathbf{h}_n}{\widetilde{\eta}_n}\right) :\equiv \mathbf{Q}_{n+1}^{\star}\underline{\mathbf{e}}_1\rho_0\,.} \tag{9.227}$$

In view of

$$\begin{aligned}
\|\underline{\mathbf{e}}_1\rho_0 - \underline{\mathbf{H}}_n\mathbf{k}_n\|^2 &= \|\mathbf{Q}_{n+1}^{\star}\underline{\mathbf{e}}_1\rho_0 - \underline{\mathbf{R}}_n^{\mathrm{QR}}\mathbf{k}_n\|^2 \\
&= \|\underline{\mathbf{h}}_n - \underline{\mathbf{R}}_n^{\mathrm{QR}}\mathbf{k}_n\|^2 \\
&= \|\mathbf{h}_n - \mathbf{R}_n^{\mathrm{QR}}\mathbf{k}_n\|^2 + |\widetilde{\eta}_n|^2
\end{aligned} \tag{9.228}$$

we see that

$$\boxed{\mathbf{k}_n = (\mathbf{R}_n^{\mathrm{QR}})^{-1}\mathbf{h}_n} \tag{9.229}$$

is the solution of our least-squares problem and that the corresponding least-squares error equals

$$\boxed{\|\underline{\mathbf{e}}_1\rho_0 - \underline{\mathbf{H}}_n\mathbf{k}_n\|^2 = |\widetilde{\eta}_n|^2\,.} \tag{9.230}$$

In fact, multiplying the least-squares problem (9.226) by the unitary matrix $\mathbf{Q}_{n+1}^{\star}$ turns it into one with the upper triangular matrix $\underline{\mathbf{R}}_n^{\mathrm{QR}}$, see (9.228), where the choice of $\mathbf{k}_n$ no longer influences the defect of the last equation, and thus the problem is solved by choosing $\mathbf{k}_n$ such that the first n equations are fulfilled.

¿From (9.226) and (9.230) we see in particular that the minimum residual norm is equal to $|\widetilde{\eta}_n|$ and can be found without computing $\mathbf{k}_n$ or the residual $\mathbf{r}_n$. We will determine the unitary matrix $\mathbf{Q}_{n+1}$ only in its factored form, namely as the product of n Givens rotations that are chosen to annihilate the subdiagonal elements of the Hessenberg (or tridiagonal) matrix:

$$\mathbf{Q}_{n+1} :\equiv \left(\begin{array}{c|c} \mathbf{Q}_n & \mathbf{o} \\ \hline \mathbf{o}^{\mathsf{T}} & 1 \end{array}\right)\mathbf{G}_n \qquad \text{with} \quad \mathbf{G}_n :\equiv \left(\begin{array}{c|cc} \mathbf{I}_{n-1} & \mathbf{o} & \mathbf{o} \\ \hline \mathbf{o}^{\mathsf{T}} & c_n & -s_n \\ \mathbf{o}^{\mathsf{T}} & \overline{s_n} & c_n \end{array}\right), \tag{9.231}$$

where $c_n \geq 0$ and s_n satisfying $c_n^2 + |s_n^2| = 1$ are chosen such that the two last coordinates of the last column of $\underline{\mathbf{H}}_n$ with respect to the orthogonal basis consisting of the columns of

$$\left(\begin{array}{c|c} \mathbf{Q}_n & \mathbf{o} \\ \hline \mathbf{o}^{\mathsf{T}} & 1 \end{array}\right)$$

that is,

$$\begin{pmatrix} \star \\ \vdots \\ \star \\ \mu_n \\ \nu_n \end{pmatrix} :\equiv \left(\begin{array}{c|c} \mathbf{Q}_n^\star & \mathbf{o} \\ \hline \mathbf{o}^\mathsf{T} & 1 \end{array}\right) \underline{\mathbf{H}}_n \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix} ,$$

are transformed by $\mathbf{G}_n$ so that the last component vanishes:

$$\mathbf{G}_n^\star \begin{pmatrix} \star \\ \vdots \\ \star \\ \mu_n \\ \nu_n \end{pmatrix} = \begin{pmatrix} \star \\ \vdots \\ \star \\ c_n\mu_n + s_n\nu_n \\ 0 \end{pmatrix} .$$

This means that

$$\boxed{\begin{array}{lll} c_n := \dfrac{|\mu_n|}{\sqrt{|\mu_n|^2 + |\nu_n|^2}} \, , & s_n := c_n \overline{\dfrac{\nu_n}{\mu_n}} \, , & \text{if } \mu_n \neq 0 \, , \\ & & \\ c_n := 0 \, , & s_n := 1 \, , & \text{if } \mu_n = 0 \, . \end{array}} \tag{9.232}$$

If $\underline{\mathbf{H}}_n$ is real, c_n and s_n are the cosine and sine of the rotation angle.

In view of (9.227) and (9.231) updating $\underline{\mathbf{h}}_{n-1}$ is simple:

$$\underline{\mathbf{h}}_n = \mathbf{G}_n^\star \left(\frac{\underline{\mathbf{h}}_{n-1}}{0}\right) = \mathbf{G}_n^\star \begin{pmatrix} \mathbf{h}_{n-1} \\ \hline \widetilde{\eta}_{n-1} \\ 0 \end{pmatrix} = \begin{pmatrix} \mathbf{h}_{n-1} \\ \hline c_n \, \widetilde{\eta}_{n-1} \\ -\overline{s_n} \, \widetilde{\eta}_{n-1} \end{pmatrix} .$$

So,

$$\boxed{\left(\frac{\mathbf{h}_n}{\widetilde{\eta}_n}\right) = \underline{\mathbf{h}}_n := \begin{pmatrix} \mathbf{h}_{n-1} \\ \hline c_n \, \widetilde{\eta}_{n-1} \\ -\overline{s_n} \, \widetilde{\eta}_{n-1} \end{pmatrix} .} \tag{9.233}$$

In particular, since $\widetilde{\eta}_1 = \|\mathbf{r}_0\|$, it follows by induction that

$$\boxed{\|\underline{\mathbf{e}}_1 \rho_0 - \underline{\mathbf{H}}_n \mathbf{k}_n\| = |\widetilde{\eta}_n| = |s_n \widetilde{\eta}_{n-1}| = |s_1 \, s_2 \cdots s_n| \, \|\mathbf{r}_0\| \, .} \tag{9.234}$$

Hence, in GMRES we can update the QR decomposition of $\underline{\mathbf{H}}_n$ along with the generation of the Arnoldi basis, where in each step a new column of $\underline{\mathbf{H}}_n$ containing the Gram-Schmidt coefficients is created. At the same time we can compute the residual norm for nearly free (without computing the residual itself). Once the norm is sufficiently small, we find the solution $\mathbf{k}_n$ of the least squares problem by solving an upper triangular system, see (9.229), and can then insert it into the first equation of (9.222) to determine the approximate solution $\mathbf{x}_n$. Note that an approximation $\mathbf{x}_n$ is only available at the end, and that to compute it, the whole Arnoldi basis has to be stored. Therefore, in practice, GMRES is normally restarted after a fixed number of steps, say m. In Box 9.1 we give the standard version of this restarted GMRES(m) algorithm based on the Arnoldi process with modified Gram-Schmidt.

In the algorithm we use as before the notation $\underline{\mathbf{H}}_m \equiv: (\eta_{k,l})_{(k,l)=(0,0)}^{(m,m-1)}$ and

$$\underline{\mathbf{h}}_n^\mathsf{T} \equiv: \begin{pmatrix} \eta_0 & \eta_1 & \ldots & \eta_{n-1} & \widetilde{\eta}_n \end{pmatrix} .$$

Recall that the first n components of $\underline{\mathbf{h}}_n$ are the same as in $\underline{\mathbf{h}}_{n-1}$; only η_{n-1} and $\widetilde{\eta}_n$ have to be computed. As initial value for $n = 0$ we can use $\underline{\mathbf{h}}_n := \begin{pmatrix} 1 \end{pmatrix}$, which means to set $\widetilde{\eta}_0 := 1$.

In our formulation of this algorithm some quantities are overwritten: $\widetilde{\mathbf{y}}_n$ in the modified Gram-Schmidt part and the elements $\eta_{k,l}$ of $\underline{\mathbf{H}}_m$ during the QR decomposition of this matrix. In fact, after n steps, we will have stored there the upper triangle of the leading $n \times n$ submatrix of the R-factor, that is, $\mathbf{R}_n^{\mathrm{QR}} = (\eta_{k,l})_{(k,l)=(0,0)}^{(n-1,n-1)}$.

Box 9.1 The restarted GMRES algorithm

Algorithm 9.13 (GMRES(m)). .
For solving $\mathbf{A}\mathbf{x} = \mathbf{b}$ *choose an initial approximation* $\mathbf{x}_0$, *and let* $\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$, $\rho_0 := \|\mathbf{r}_0\|$, $\mathbf{y}_0 := \mathbf{r}_0/\rho_0$, *and* $\widetilde{\eta}_0 := 1$.
Then, for $n = 1, \dots, m$*:*

1. *Do one step of the Arnoldi algorithm by applying the modified Gram-Schmidt process to* $\widetilde{\mathbf{y}}_n := \mathbf{A}\mathbf{y}_{n-1}$*:*

$$\begin{aligned} &\textit{for } k := 0, 1, \dots, n-1 : \\ &\quad \eta_{k,n-1} := \langle \mathbf{y}_k, \widetilde{\mathbf{y}}_n \rangle \,, \qquad \widetilde{\mathbf{y}}_n := \widetilde{\mathbf{y}}_n - \mathbf{y}_k \eta_{k,n-1} \,, \\ &\eta_{n,n-1} := \|\widetilde{\mathbf{y}}_n\| \,, \qquad \mathbf{y}_n := \widetilde{\mathbf{y}}_n / \eta_{n,n-1} \,. \end{aligned}$$

2. *If* $n > 1$, *apply the* $n-1$ *adjoint Givens rotations* $\mathbf{G}_k^\star$ $(k = 1, \dots, n-1)$ *with the parameters* c_k *and* s_k *to the new last column of* $\mathbf{H}_n$*:*

$$\begin{aligned} &\textit{for } k := 1, 2, \dots, n-1 : \\ &\begin{pmatrix} \eta_{k-1,n-1} \\ \eta_{k,n-1} \end{pmatrix} := \begin{pmatrix} c_k & s_k \\ -\overline{s_k} & c_k \end{pmatrix} \begin{pmatrix} \eta_{k-1,n-1} \\ \eta_{k,n-1} \end{pmatrix} . \end{aligned}$$

3. *Let* $\mu_n := \eta_{n-1,n-1}$, $\nu_n := \eta_{n,n-1}$ *and compute* c_n *and* s_n *of the Givens rotation* $\mathbf{G}_n$ *according to* (9.232).

4. *Apply the adjoint Givens rotation* $\mathbf{G}_n^\star$ *to update* $\underline{\mathbf{h}}_{n-1}$ *and the last two components of the modified last column of* $\underline{\mathbf{H}}_n$*:*

$$\begin{aligned} \eta_{n-1} &:= c_n \, \widetilde{\eta}_{n-1} \,, & \eta_{n-1,n-1} &:= c_n \eta_{n-1,n-1} + s_n \eta_{n,n-1} \,, \\ \widetilde{\eta}_n &:= -\overline{s_n} \, \widetilde{\eta}_{n-1} \,, & \eta_{n,n-1} &:= 0 \,. \end{aligned}$$

5. *If* $|\widetilde{\eta}_n| \leq \mathrm{tol}$ *or* $n = m$, *the iteration or the sweep terminates and the triangular system* $\mathbf{R}_n^{\mathrm{QR}} \mathbf{k}_n = \mathbf{h}_n$ *has to be solved for* $\mathbf{k}_n$, *so that* $\mathbf{x}_n := \mathbf{x}_0 + \mathbf{Y}_n \mathbf{k}_n$ *can be computed.*

If $|\widetilde{\eta}_n| \leq \mathrm{tol}$ *terminate; otherwise, set* $\mathbf{x}_0 := \mathbf{x}_m$ *and restart.*

9.8.4 MINRES

When we have a Hermitian matrix $\mathbf{A}$, the Arnoldi process reduces to the symmetric Lanczos process, see Lemma 9.8.1, which has the benefit of a three-term recursion. So for the generation of the basis there is no need to keep all the computed basis vectors available. However, to fully capitalize upon this, we also need a recursive way of computing $\mathbf{x}_n$ in order to avoid the evaluation of $\mathbf{x}_n := \mathbf{x}_0 + \mathbf{Y}_n\mathbf{k}_n$ at the end of the algorithm. In the Hermitian case such a recursion exists and is part of the MINRES algorithm of [PS75]. This also means that there is no reason for restarting MINRES.

To see how this can be done, we first recall that the n-vector $\mathbf{h}_n$ emerges from the $(n-1)$-vector $\mathbf{h}_{n-1}$ by just appending the component $\eta_n := c_n\widetilde{\eta}_{n-1}$, see (9.233). We rewrite $\mathbf{x}_n = \mathbf{x}_0 + \mathbf{Y}_n\mathbf{k}_n$ by using (9.229) as

$$\boxed{\mathbf{x}_n = \mathbf{x}_0 + \mathbf{Z}_n\mathbf{h}_n\,,} \qquad \text{where} \quad \mathbf{Z}_n :\equiv \mathbf{Y}_n(\mathbf{R}_n^{\mathrm{QR}})^{-1}$$

contains the search directions $\mathbf{z}_0, \ldots, \mathbf{z}_{n-1}$, and we conclude that

$$\boxed{\mathbf{x}_n := \mathbf{x}_{n-1} + \mathbf{z}_{n-1}c_n\widetilde{\eta}_{n-1}\,.} \tag{9.237}$$

This is true for MINRES and GMRES. However, only in MINRES $\mathbf{R}_n^{\mathrm{QR}}$ is a banded upper tridiagonal matrix with bandwidth three. Therefore the relation

$$\boxed{\mathbf{Y}_n = \mathbf{Z}_n\mathbf{R}_n^{\mathrm{QR}}} \tag{9.238}$$

can be viewed as the matrix representation of a three-term recursion for generating the vectors $\{\mathbf{z}_k\}_{k=0}^{n-1}$: if the elements in column k of $\mathbf{R}_n^{\mathrm{QR}}$ are $\widetilde{\gamma}_{k-2}$, $\widetilde{\beta}_{k-1}$, and $\widetilde{\alpha}_k$, then

$$\boxed{\mathbf{z}_k := (\mathbf{y}_k - \mathbf{z}_{k-1}\widetilde{\beta}_{k-1} - \mathbf{z}_{k-2}\widetilde{\gamma}_{k-2})/\widetilde{\alpha}_k\,.} \tag{9.239}$$

Our variant of MINRES, which as we mentioned differs from the original of [PS75] by using a QR instead of an LQ decomposition is given in Box 9.2. In the steps 2 and 4 of the MINRES loop, the last column of the tridiagonal matrix $\underline{\mathbf{T}}_n$ with the elements β_{n-2}, α_{n-1}, and β_{n-1} gets transformed into the last column of the banded upper triangular matrix $\mathbf{R}_n^{\mathrm{QR}}$ with the elements $\widetilde{\gamma}_{n-3}$, $\widetilde{\beta}_{n-2}$, and $\widetilde{\alpha}_{n-1}$ in the positions $(n-3, n-1)$, $(n-2, n-1)$, and $(n-1, n-1)$, respectively. In contrast to GMRES, storing the vectors $\mathbf{y}_n$ is not necessary here.

In general, computing or updating the residual is unnecessary in both MINRES and GMRES since its norm is equal to $|\widetilde{\eta}_n|$. So, the progress of the algorithm can be judged from this number.

Let us nevertheless note that by multiplying (9.237) by $\mathbf{A}$ we could find a recursion for the residuals; but since it would require an extra matrix-vector product, it is not of interest. There is another, cheaper way of updating the residual, which also holds for GMRES. First, inserting $\underline{\mathbf{H}}_n = \mathbf{Q}_{n+1}\underline{\mathbf{R}}_n^{\mathrm{QR}}$ and (9.229) into (9.224) and taking (9.227) into account we get

$$\begin{aligned}
\mathbf{r}_n &= \mathbf{Y}_{n+1}\left(\underline{\mathbf{e}}_1\rho_0 - \mathbf{Q}_{n+1}\underline{\mathbf{R}}_n^{\mathrm{QR}}(\mathbf{R}_n^{\mathrm{QR}})^{-1}\mathbf{h}_n\right) \\
&= \mathbf{Y}_{n+1}\left(\underline{\mathbf{e}}_1\rho_0 - \mathbf{Q}_{n+1}\left(\frac{\mathbf{h}_n}{0}\right)\right) \\
&= \mathbf{Y}_{n+1}\mathbf{Q}_{n+1}\,\mathbf{l}_{n+1}\widetilde{\eta}_n\,, \qquad\qquad (9.240)
\end{aligned}$$

where, as before, $\mathbf{l}_{n+1} :\equiv \begin{pmatrix} 0 & \ldots & 0 & 1 \end{pmatrix}^{\mathsf{T}}$ is the last column of the unit matrix of order $n+1$. Using (9.231) we see further that

$$\begin{aligned}
\mathbf{r}_n &= \begin{pmatrix} \mathbf{Y}_n & \big| & \mathbf{y}_n \end{pmatrix} \left(\begin{array}{c|c} \mathbf{Q}_n & \mathbf{o} \\ \hline \mathbf{o}^{\mathsf{T}} & 1 \end{array}\right) \mathbf{G}_n \left(\frac{\mathbf{o}}{1}\right)\widetilde{\eta}_n \\
&= -\mathbf{Y}_n\mathbf{Q}_n\mathbf{l}_n s_n\widetilde{\eta}_n + \mathbf{y}_n c_n\widetilde{\eta}_n\,.
\end{aligned}$$

Finally, using (9.240) and $\widetilde{\eta}_n = -\overline{s_n}\,\widetilde{\eta}_{n-1}$ (see (9.233)) to simplify the first term on the right-hand side, we get the recursion

$$\boxed{\mathbf{r}_n = \mathbf{r}_{n-1}|s_n|^2 + \mathbf{y}_n c_n\widetilde{\eta}_n\,.} \tag{9.241}$$

Box 9.2 A variant of the MinRes algorithm

Algorithm 9.14 (MinRes). .
For solving $\mathbf{A}\mathbf{x} = \mathbf{b}$ *with Hermitian* $\mathbf{A}$ *choose* $\mathbf{x}_0$*, and let* $\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$*,* $\rho_0 := \|\mathbf{r}_0\|$*,* $\mathbf{y}_0 := \mathbf{r}_0/\rho_0$*,* $\mathbf{z}_{-2} := \mathbf{z}_{-1} := \mathbf{o}$*, and* $\widetilde{\eta}_0 := 1$.
Then, for $n = 1, \dots, m$*:*

1. *Do one step of the symmetric Lanczos algorithm:*

$$\begin{array}{ll} \widetilde{\mathbf{y}}_n := \mathbf{A}\mathbf{y}_{n-1}\,, & \widetilde{\mathbf{y}}_n := \widetilde{\mathbf{y}}_n - \mathbf{y}_{n-2}\beta_{n-2} \quad \textit{if } n > 1, \\ \alpha_{n-1} := \langle \mathbf{y}_{n-1}, \widetilde{\mathbf{y}}_n \rangle\,, & \widetilde{\mathbf{y}}_n := \widetilde{\mathbf{y}}_n - \mathbf{y}_{n-1}\alpha_{n-1}\,, \\ \beta_{n-1} := \|\widetilde{\mathbf{y}}_n\|\,, & \mathbf{y}_n := \widetilde{\mathbf{y}}_n/\beta_{n-1}\,. \end{array}$$

2. *Let* $\widetilde{\alpha}_{n-1} := \alpha_{n-1}$*, and, if* $n > 1$*,* $\widetilde{\beta}_{n-1} := \beta_{n-1}$.
If $n > 2$*, apply* $\mathbf{G}^\star_{n-2}$ *to the new last column of* $\underline{\mathbf{T}}_n$*:*

$$\begin{pmatrix} \widetilde{\gamma}_{n-3} \\ \widetilde{\beta}_{n-2} \end{pmatrix} := \begin{pmatrix} c_{n-2} & s_{n-2} \\ -\overline{s_{n-2}} & c_{n-2} \end{pmatrix} \begin{pmatrix} 0 \\ \widetilde{\beta}_{n-2} \end{pmatrix};$$

if $n > 1$*, apply* $\mathbf{G}^\star_{n-1}$ *to the last column of* $\mathbf{G}^\star_{n-2}\underline{\mathbf{T}}_n$*:*

$$\begin{pmatrix} \widetilde{\beta}_{n-2} \\ \widetilde{\alpha}_{n-1} \end{pmatrix} := \begin{pmatrix} c_{n-1} & s_{n-1} \\ -\overline{s_{n-1}} & c_{n-1} \end{pmatrix} \begin{pmatrix} \widetilde{\beta}_{n-2} \\ \widetilde{\alpha}_{n-1} \end{pmatrix}.$$

3. *Let* $\mu_n := \widetilde{\alpha}_{n-1}\,,\ \nu_n := \beta_{n-1}$ *(no tilde!) and compute* c_n *and* s_n *of the Givens rotation* $\mathbf{G}_n$ *according to* (9.232)*.*

4. *Apply the adjoint Givens rotation* $\mathbf{G}^\star_n$ *to update* $\underline{\mathbf{h}}_{n-1}$ *and the last two components of the modified last column of* $\underline{\mathbf{T}}_n$*:*

$$\eta_{n-1} := c_n\,\widetilde{\eta}_{n-1}\,, \quad \widetilde{\eta}_n := -\overline{s_n}\,\widetilde{\eta}_{n-1}\,, \quad \widetilde{\alpha}_{n-1} := c_n\mu_n + s_n\nu_n\,.$$

5. *Compute* $\mathbf{z}_{n-1}$ *and* $\mathbf{x}_n$ *according to* (9.239) *and* (9.237)*.*

6. *If* $|\widetilde{\eta}_n| \le \mathrm{tol}$*, the algorithm terminates and* $\mathbf{x}_n$ *is a sufficiently accurate approximate solution.*

9.8.5 FOM★

Instead of minimizing the residual norm we could aim at satisfying the Galerkin condition (9.197), $\mathbf{r}_n \perp \mathcal{K}_n$ or $\mathbf{Y}_n^\star \mathbf{r}_n = \mathbf{o}$, known from the CG method, though it does in general not belong to an optimality property. In view of $\mathbf{Y}_n^\star \mathbf{Y}_{n+1} = \left(\begin{array}{c|c} \mathbf{I}_n & \mathbf{o} \end{array} \right)$ Eq. (9.224) then yields the $n \times n$ linear Hessenberg system

$$\boxed{\mathbf{H}_n \mathbf{k}_n = \mathbf{e}_1 \rho_0 \,,} \tag{9.242}$$

which has a unique solution if and only if $\mathbf{H}_n$ is nonsingular. A recursive solution by the Gauss LU decomposition without pivoting is possible if and only if all leading principle submatrices $\mathbf{H}_k$ are nonsingular. It can be shown that this condition is equivalent to the existence of the approximants $\mathbf{x}_k$ for all k. This is in particular also true if $\mathbf{A}$ is Hermitian and thus $\underline{\mathbf{H}}_n$ is tridiagonal. When $\mathbf{A}$ is even Hpd, the LU decomposition does exist and the above approach is mathematically equivalent to the CG method.

In the case of a non-Hermitian system, solving (9.242) is the basis of the **Full Orthogonalization Method** (FOM). Like GMRES it applies the Arnoldi process starting from $\mathbf{r}_0$ to generate the orthonormal Krylov subspace basis $\{\mathbf{y}_k\}$ and the matrix $\underline{\mathbf{H}}_n$. If the system (9.242) is also solved recursively (which may not be stable, however), termination can be controlled using the following result:

LEMMA 9.8.3. *The* FOM *residual vector* $\mathbf{r}_n$ *can be expressed in terms of the subdiagonal element* $\eta_{n,n-1}$ *of* $\underline{\mathbf{H}}_n$*, the solution* $\mathbf{k}_n$ *or* (9.242)*, and the latest Krylov subspace basis vector* $\mathbf{y}_n$*:*

$$\mathbf{r}_n = -\mathbf{y}_n \, \eta_{n,n-1} \, \mathbf{e}_n^\star \, \mathbf{k}_n \,, \tag{9.243}$$

$$\|\mathbf{r}_n\| = |\eta_{n,n-1}| \, |\mathbf{e}_n^\star \mathbf{k}_n| \,. \tag{9.244}$$

PROOF. By (9.224), when we separate the last basis vector $\mathbf{y}_n$, and by (9.242), $\mathbf{r}_n = \mathbf{Y}_n (\mathbf{e}_1 \rho_0 - \mathbf{H}_n \mathbf{k}_n) - \mathbf{y}_n \eta_{n,n-1} \mathbf{e}_n^\star \mathbf{k}_n = -\mathbf{y}_n \eta_{n,n-1} \mathbf{e}_n^\star \mathbf{k}_n$. □

9.8.6 SYMMLQ★

Related to MINRES and the aforementioned FOM-type realization of the CG method is yet another algorithm of [PS75] called SYMMLQ. It is also applicable to nonsingular Hermitian systems, including indefinite ones. It differs from MINRES in two fundamental aspects: the search space

$$\boxed{\mathbf{x}_n - \mathbf{x}_0 \in \mathcal{L}_n := \mathbf{A}\mathcal{K}_n(\mathbf{A}, \mathbf{r}_0)} \tag{9.245}$$

and the optimization property

$$\boxed{\|\mathbf{d}_n\| = \|\mathbf{x}_n - \mathbf{x}_\star\| = \min!} \tag{9.246}$$

So, SYMMLQ is minimizing the error in the 2-norm, but, as we mentioned before, this requires a space different from $\mathcal{K}_n(\mathbf{A}, \mathbf{r}_0)$. Additionally SYMMLQ allows one to compute with an extra recursion the CG iterates $\mathbf{x}_n^{\mathrm{CG}}$ whenever they all exist.

We construct as in MINRES with the symmetric Lanczos process an orthonormal basis $\mathbf{y}_0, \ldots, \mathbf{y}_{n-1}$ of $\mathcal{K}_n :\equiv \mathcal{K}_n(\mathbf{A}, \mathbf{r}_0)$ and write these vectors as the columns of $\mathbf{Y}_n$, so that (9.245) is matched by the representation

$$\boxed{\mathbf{x}_n - \mathbf{x}_0 = \mathbf{A}\mathbf{Y}_n \mathbf{k}_n \,,} \tag{9.247}$$

where, of course, $\mathbf{k}_n$ is different from the one in (9.222), GMRES, MINRES, or FOM. Condition (9.246) takes the form

$$\|\mathbf{A}\mathbf{Y}_n \mathbf{k}_n - (\mathbf{x}_\star - \mathbf{x}_0)\| = \min! \tag{9.248}$$

which is a least squares problem with the normal equations

$$\mathbf{Y}_n^\star \mathbf{A}^\star \mathbf{A} \mathbf{Y}_n \mathbf{k}_n = \mathbf{Y}_n^\star \mathbf{A}^\star (\mathbf{x}_\star - \mathbf{x}_0) \tag{9.249}$$

and the Galerkin condition

$$\boxed{\mathbf{d}_n = \mathbf{x}_n - \mathbf{x}_\star \perp \mathbf{A}\mathcal{K}_n(\mathbf{A}, \mathbf{r}_0) \,, \qquad \textit{i.e.,} \qquad \mathbf{r}_n \perp \mathcal{K}_n(\mathbf{A}, \mathbf{r}_0) \,.} \tag{9.250}$$

Now, $\mathbf{A}^\star = \mathbf{A}$ and $\mathbf{A}(\mathbf{x}_\star - \mathbf{x}_0) = \mathbf{r}_0$. So, since $\mathbf{y}_k^\star \mathbf{r}_0 = \mathbf{y}_k^\star \mathbf{y}_0 \rho_0 = \delta_{k,0}\rho_0,$ we can simplify the right-hand side of (9.249) to $\mathbf{e}_1\rho_0$. On the left-hand side, using the Lanczos relation $\mathbf{A}\mathbf{Y}_n = \mathbf{Y}_{n+1}\underline{\mathbf{T}}_n$ and the QR decomposition $\underline{\mathbf{T}}_n = \underline{\mathbf{Q}}_n \mathbf{R}_n^{\mathrm{QR}}$ (with an $(n+1) \times n$ matrix $\underline{\mathbf{Q}}_n$ with orthonormal columns and an $n \times n$ upper triangular $\underline{\mathbf{Q}}_n$, we get

$$\begin{aligned}\mathbf{Y}_n^\star \mathbf{A}^\star \mathbf{A}\mathbf{Y}_n &= \underline{\mathbf{T}}_n^\star \mathbf{Y}_{n+1}^\star \mathbf{Y}_{n+1}\underline{\mathbf{T}}_n = \underline{\mathbf{T}}_n^\star \underline{\mathbf{T}}_n = (\mathbf{R}_n^{\mathrm{QR}})^\star \underline{\mathbf{Q}}_n^\star \underline{\mathbf{Q}}_n \mathbf{R}_n^{\mathrm{QR}} \\ &= (\mathbf{R}_n^{\mathrm{QR}})^\star \mathbf{R}_n^{\mathrm{QR}} .\end{aligned}$$

$(\mathbf{R}_n^{\mathrm{QR}})^\star \mathbf{R}_n^{\mathrm{QR}}$ is just the Cholesky decomposition of the Hpd matrix $\underline{\mathbf{T}}_n^\star \underline{\mathbf{T}}_n$, here computed via the more stable QR decomposition of $\underline{\mathbf{T}}_n$. Altogether, (9.249) reduces to

$$\boxed{(\mathbf{R}_n^{\mathrm{QR}})^\star\, \mathbf{R}_n^{\mathrm{QR}}\, \mathbf{k}_n = \mathbf{e}_1 \rho_0 \,.} \tag{9.251}$$

Setting $\mathbf{L}_n^{\mathrm{QR}} :\equiv (\mathbf{R}_n^{\mathrm{QR}})^\star$, inserting $\mathbf{k}_n$ into (9.247), and using first $\mathbf{A}\mathbf{Y}_n = \mathbf{Y}_{n+1}\underline{\mathbf{T}}_n$ and then $\underline{\mathbf{T}}_n = \underline{\mathbf{Q}}_n \mathbf{R}_n^{\mathrm{QR}}$, we obtain further

$$\begin{aligned}\mathbf{x}_n &= \mathbf{x}_0 + \mathbf{A}\mathbf{Y}_n (\mathbf{R}_n^{\mathrm{QR}})^{-1} (\mathbf{L}_n^{\mathrm{QR}})^{-1} \mathbf{e}_1 \rho_0 \\ &= \mathbf{x}_0 + \mathbf{Y}_{n+1}\underline{\mathbf{T}}_n (\mathbf{R}_n^{\mathrm{QR}})^{-1} (\mathbf{L}_n^{\mathrm{QR}})^{-1} \mathbf{e}_1 \rho_0 \\ &= \mathbf{x}_0 + \mathbf{Y}_{n+1}\underline{\mathbf{Q}}_n (\mathbf{L}_n^{\mathrm{QR}})^{-1} \mathbf{e}_1 \rho_0 \,.\end{aligned}$$

So, if we let

$$\boxed{\mathbf{W}_n :\equiv \mathbf{Y}_{n+1}\underline{\mathbf{Q}}_n \,, \qquad \mathbf{g}_n :\equiv (\mathbf{L}_n^{\mathrm{QR}})^{-1} \mathbf{e}_1 \rho_0 \,,} \tag{9.252}$$

we finally get

$$\boxed{\mathbf{x}_n = \mathbf{x}_0 + \mathbf{W}_n \mathbf{g}_n = \mathbf{x}_{n-1} + \mathbf{w}_{n-1} g_{n-1} \,.} \tag{9.253}$$

$\mathbf{W}_n$ and the coordinate vector $\mathbf{g}_n$ are easy to update: $\mathbf{W}_n$ by appending the last column of $\mathbf{Y}_{n+1}$ and applying the Givens tranformation $\mathbf{G}_n$ to the last two columns; and $\mathbf{g}_n$ by the three-term recursion induced by $\mathbf{L}_n^{\mathrm{QR}}$, a lower triangular matrix with bandwidth three.

The resulting SYMMLQ algorithm is given in Box 9.3. Note that the first three steps and part of the forth are the same as in MINRES.

Since $\mathbf{L}_n^{\mathrm{QR}} = (\mathbf{R}_n^{\mathrm{QR}})^\star$ the recursion for the components g_k of $\mathbf{g}_n$ is the same as the recursion for the columns $\mathbf{w}_k$ of $\mathbf{W}_n$. Of course, we could exhibit this by writing the relation $\mathbf{L}_n^{\mathrm{QR}}\mathbf{g}_n = \mathbf{e}_1\rho_0$ from (9.252) as $\mathbf{g}_n^\star \mathbf{R}_n^{\mathrm{QR}} = \rho_0 \mathbf{e}_1^\star$.

Box 9.3 The SYMMLQ algorithm

Algorithm 9.15 (SYMMLQ)**.** .
For solving $\mathbf{A}\mathbf{x} = \mathbf{b}$ *with Hermitian* $\mathbf{A}$ *choose* $\mathbf{x}_0$*, and let* $\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$*,* $\rho_0 := \|\mathbf{r}_0\|$*,* $\mathbf{w}_0 := \mathbf{y}_0 := \mathbf{r}_0/\rho_0$*, and* $g_{-2} := g_{-1} := 0$.
Then, for $n = 1, \ldots, m$*:*

1. *Do one step of the symmetric Lanczos algorithm:*

$$\begin{array}{ll} \widetilde{\mathbf{y}}_n := \mathbf{A}\mathbf{y}_{n-1}\,, & \widetilde{\mathbf{y}}_n := \widetilde{\mathbf{y}}_n - \mathbf{y}_{n-2}\beta_{n-2} \quad \textit{if } n > 1, \\ \alpha_{n-1} := \langle \mathbf{y}_{n-1}, \widetilde{\mathbf{y}}_n \rangle\,, & \widetilde{\mathbf{y}}_n := \widetilde{\mathbf{y}}_n - \mathbf{y}_{n-1}\alpha_{n-1}\,, \\ \beta_{n-1} := \|\widetilde{\mathbf{y}}_n\|\,, & \mathbf{y}_n := \widetilde{\mathbf{y}}_n/\beta_{n-1}\,. \end{array}$$

2. *Let* $\widetilde{\alpha}_{n-1} := \alpha_{n-1}$*, and, if* $n > 1$*,* $\widetilde{\beta}_{n-1} := \beta_{n-1}$.
   If $n > 2$*, apply* $\mathbf{G}_{n-2}^\star$ *to the new last column of* $\underline{\mathbf{T}}_n$*:*

$$\begin{pmatrix} \widetilde{\gamma}_{n-3} \\ \widetilde{\beta}_{n-2} \end{pmatrix} := \begin{pmatrix} c_{n-2} & s_{n-2} \\ -\overline{s_{n-2}} & c_{n-2} \end{pmatrix} \begin{pmatrix} 0 \\ \widetilde{\beta}_{n-2} \end{pmatrix};$$

   if $n > 1$*, apply* $\mathbf{G}_{n-1}^\star$ *to the last column of* $\mathbf{G}_{n-2}^\star \underline{\mathbf{T}}_n$*:*

$$\begin{pmatrix} \widetilde{\beta}_{n-2} \\ \widetilde{\alpha}_{n-1} \end{pmatrix} := \begin{pmatrix} c_{n-1} & s_{n-1} \\ -\overline{s_{n-1}} & c_{n-1} \end{pmatrix} \begin{pmatrix} \widetilde{\beta}_{n-2} \\ \widetilde{\alpha}_{n-1} \end{pmatrix}.$$

3. *Let* $\mu_n := \widetilde{\alpha}_{n-1}\,,\ \nu_n := \beta_{n-1}$ *(no tilde!) and compute* c_n *and* s_n *of the Givens rotation* $\mathbf{G}_n$ *according to* (9.232)*.*

4. *Apply the adjoint Givens rotation* $\mathbf{G}_n^\star$ *to update the last two components of the modified last column of* $\underline{\mathbf{T}}_n$ *(which turns into* $\mathbf{R}_n^{\mathrm{QR}}$*), then compute the last component* g_{n-1} *of* $\mathbf{g}_n$ *by the three-term recurrence induced by* $\mathbf{L}_n^{\mathrm{QR}} = (\mathbf{R}_n^{\mathrm{QR}})^\star$.

5. *Compute* $\mathbf{x}_n$ *according to* (9.253) *and update the value of the error norm* η_n*. [Details not given.]*

6. *If* $|\eta_n| \le \mathrm{tol}$*, the algorithm terminates and* $\mathbf{x}_n$ *is a sufficiently accurate approximate solution.*

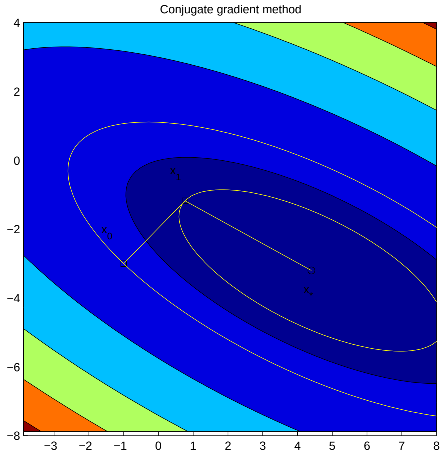


Figure 9.4: Conjugate directions — the CG method for $N = 2$.

Bibliography

[AMS90] S. F. Ashby, T. A. Manteuffel, and P. E. Saylor. A taxonomy for conjugate gradient methods. *SIAM J. Numer. Anal.*, 27:1542–1568, 1990.

[Arn51] W. E. Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quart. Appl. Math.*, 9:17–29, 1951.

[BBC$^+$94] 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, PA, 1994.

[CW85] J. K. Cullum and R. A. Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations (2 Vols.)*. Birkhäuser, Boston-Basel-Stuttgart, 1985.

[DS00] Jack Dongarra and Francis Sullivan. Guest editors' introduction to the top 10 algorithms. *Computing in Science and Engineering*, 2(1):22–23, 2000.

[FF90] B. Fischer and R. W. Freund. On the constrained Chebyshev approximation problem on ellipses. *Journal of Approximation Theory*, 62:297–315, 1990.

[FF91] B. Fischer and R. W. Freund. Chebyshev polynomials are not always optimal. *Journal of Approximation Theory*, 65:261–272, 1991.

[Fle76] R. Fletcher. Conjugate gradient methods for indefinite systems. In G. A. Watson, editor, *Numerical Analysis, Dundee, 1975*, volume 506 of *Lecture Notes in Mathematics*, pages 73–89. Springer, Berlin, 1976.

[FN91] R. W. Freund and N. M. Nachtigal. QMR: a quasi-minimal residual method for non-Hermitian linear systems. *Numer. Math.*, 60:315–339, 1991. Received Feb. 19, 1991.

[Gre97] A. Greenbaum. *Iterative Methods for Solving Linear Systems*. SIAM, Philadelphia, PA, 1997.

[GV61] G. H. Golub and R. S. Varga. Chebyshev semiiterative methods, successive overrelaxation iterative methods, and second order Richardson iterative methods. *Numer. Math.*, 3:147–168, 1961.

[Hen74] P. Henrici. *Applied and Computational Complex Analysis, Vol. 1*. Wiley, New York, 1974.

[HS52] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bureau Standards*, 49:409–435, 1952.

[Lan50] C. Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *J. Res. Nat. Bureau Standards*, 45:255–281, 1950.

[Lan52] C. Lanczos. Solution of systems of linear equations by minimized iterations. *J. Res. Nat. Bureau Standards*, 49:33–53, 1952.

[Mv77] J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear equations systems of which the coefficient matrix is a symmetric M-matrix. *Math. Comp.*, 31:148–162, 1977.

[Pai71] C. C. Paige. *The computations of eigenvalues and eigenvectors of very large sparse matrices*. PhD thesis, University of London, 1971.

[Par80] B. N. Parlett. *The Symmetric Eigenvalue Problem*. Prentice-Hall, Englewood Cliffs, N.J., 1980.

[Par98] B. N. Parlett. *The Symmetric Eigenvalue Problem*. Classics in Applied Mathematics. SIAM, 2nd edition, 1998.

[PS75] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM J. Numer. Anal.*, 12:617–629, 1975.

[Rut57] H. Rutishauser. *Der Quotienten-Differenzen-Algorithmus*. Mitt. Inst. angew. Math. ETH, Nr. 7. Birkhäuser, Basel, 1957.

[Rut59] H. Rutishauser. Theory of gradient methods. In *Refined Iterative Methods for Computation of the Solution and the Eigenvalues of Self-Adjoint Boundary Value Problems*, Mitt. Inst. angew. Math. ETH Zürich, Nr. 8, pages 24–49. Birkhäuser, Basel, 1959.

[Saa96] Youcef Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing, Boston, 1996.

[SRS68] H. R. Schwarz, H. Rutishauser, and E. Stiefel. *Numerik symmetrischer Matrizen*. Teubner, Stuttgart, 1968.

[SS85] Y. Saad and M. H. Schultz. Conjugate gradient-like algorithms for solving nonsymmetric linear systems. *Math. Comp.*, 44:417–424, 1985.

[Sti55] E. Stiefel. Relaxationsmethoden bester Strategie zur Lösung linearer Gleichungssysteme. *Comm. Math. Helv.*, 29:157–179, 1955.

[SvM00] G. L. G. Sleijpen, H. A. van der Vorst, and J. Modersitzki. Differences in the effects of rounding errors in Krylov solvers for symmetric indefinite linear systems. *SIAM J. Matrix Anal. Appl.*, 22(3):726–751, 2000.

[van00] H. A. van der Vorst. Krylov subspace iteration. *Computing in Science and Engineering*, 2:32–37, 2000.

[Var62] R. S. Varga. *Matrix Iterative Analysis*. Prentice-Hall, Englewood Cliffs, N.J., 1962. Rev. 2nd ed., Springer-Verlag, 1999.

[You50] David M. Young. *Iterative Methods for Solving Partial Difference Equations of Elliptic Type*. PhD thesis, Harvard University, 1950. http://www.cs.utexas.edu/users/young/david_young_thesis.pdf.

[You71] D. M. Young. *Iterative Solution of Large Linear Systems*. Academic Press, Orlando, 1971.

Chapter 10

Preconditioning

One can distinguish between two different aspects of the iterative solution of a linear system. The first one in the particular acceleration technique for a sequence of iterations vectors, that is a technique used to construct a new approximation for the solution, with information from previous approximations. This was covered in Chapter 9. The second aspect is the transformation of the given system to one that can be more efficiently solved by a particular iteration method. This is called *preconditioning*. A good preconditioner improves the convergence of the iterative method, sufficiently to overcome the extra cost of its construction and application. Indeed, without a preconditioner the iterative method may even fail to converge in practice.

The general (and challenging) problem of finding an efficient preconditioner is to identify a linear operator $\mathbf{P}$ with the following properties:

1. **$\mathbf{P}^{-1}$ is a good approximation of $\mathbf{A}^{-1}$ is some sense**. Although no general theory is available, we can say that P should act so that $\mathbf{P}^{-1}\mathbf{A}$ is near to being the identity matrix and its eigenvalues are clustered within a sufficiently small region of the complex plane (see for instance [Gre97]);

2. **$\mathbf{P}$ is efficient**, in the sense that the iteration method converges much faster, in terms of CPU time, for the preconditioned system. In other words, preconditioners must be selected in such a way that the cost of constructing and using them is offset by the improved convergence properties they permit to achieve;

3. **$\mathbf{P}$ or $\mathbf{P}^{-1}$ can be constructed in parallel**, to take advantage of the architecture of modern supercomputers.

The choice of $\mathbf{P}$ varies from "black-box" algebraic techniques which can be applied to general matrices to "problem dependent" preconditioners which exploit special features of a particular class of problems. Although problem dependent preconditioners can be very powerful, there is still a practical need for efficient preconditioning techniques for large classes of problems. Between these two extrema, there is a class of preconditioners which are "general-purpose" for a particular – although large – class of problems. These preconditioners are sometimes called "grey-box" preconditioners, since the user has to supply few information about the matrix and the problem to be solved.

The preconditioners that we are going to address in this course can roughly be grouped into two familes, the one of *single-level algebraic preconditioners* and the one of *multilevel (algebraic) preconditioners*, which comprise

1. **Relaxation schemes**, like Jacobi, Gauss-Seidel and symmetric Gauss-Seidel (point or block versions) [Var00]. These schemes seldomly provide satisfactory performances as stand-alone preconditioner, but can be very effective if used as smoothers in multilevel methods (like, for example, ML [SHT04]);

2. **Polynomial preconditioners**, like Neumann, Least-Squares, and Chebyshev [Saa96a].

3. **Incomplete Factorizations preconditioners**, like IC(k,ϵ), ILU(k,ϵ), ILUT(k,ϵ), ILUS(k,ϵ) and ILUC(k,ϵ) [Saa96a];

4. **Sparse Approximate Inverses**, like SPAI [GH97], AINV [BCT00] and RIF [BT03].

5. **One-level domain decomposition preconditioners of Schwarz type**, with minimal or wider overlap among the subdomains [SBG96a, QV99]. The local linear problems can be solved with exact factorizations, incomplete factorizations, or other techniques.

6. **Two-level domain decomposition preconditioners of Schwarz type**, with either geometrically or algebraically constructed coarse grid correction.

7. **(Algebraic/Geometric) Multigrid schemes**, where the grid/matrix hierarchy is generated by geometric reasonings (MG) [Bra77], by algebraic procedures (AMG) [RS87] or by smoothed aggregation (SA) [VBM98].

8. $\mathscr{H}$**-Matrices**, which, in case of symmetric positive definite matrices, allow for an efficient approximative matrix factorisation by using a geometrically motivated tree-like storage structure [Hac99, GH03, BH03, BGH05].

Remark 10.1. *Single-level preconditioners can be used as stand-alone preconditioners, on in conjunction with multilevel preconditioners. In this latter case, the single-level preconditioner is reinterpreted as a smoother for the multilevel hierarchy.*

10.1 Preconditioning based on classical matrix splittings

Recall from Section 9.2 that a matrix splitting $\mathbf{A} = \mathbf{M} - \mathbf{N}$ with appropriately chosen $\mathbf{N}$ may be used to speed up the linear fixed point iteration

$$\boxed{\mathbf{x}_{n+1} := \mathbf{B}\mathbf{x}_n + \mathbf{b}} \qquad \text{with} \qquad \boxed{\mathbf{B} :\equiv \mathbf{I} - \mathbf{A}\,,} \tag{10.1}$$

by replacing it by

$$\boxed{\mathbf{x}_{n+1} := \widehat{\mathbf{B}}\mathbf{x}_n + \widehat{\mathbf{b}}} \tag{10.2}$$

with

$$\boxed{\widehat{\mathbf{B}} :\equiv \mathbf{M}^{-1}\mathbf{N} = \mathbf{M}^{-1}(\mathbf{I} - \mathbf{A})\,, \qquad \widehat{\mathbf{b}} :\equiv \mathbf{M}^{-1}\mathbf{b}\,.} \tag{10.3}$$

While (10.1) is the straightforward fixed point iteration for the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$, the modified iteration (10.2)–(10.3) is the fixed point iteration for the left-preconditioned system $\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b}$.

So the methods based on matrix splittings that we discussed in Section 9.2 can all be understood as linear fixed point iterations for preconditioned systems. There the aim of the preconditioning was to make the spectral radius $\rho(\widehat{\mathbf{B}})$ as small as possible, since this spectral radius equals the asymptotic rate of convergence. In any case it has to be smaller than 1, since this is a necessary and sufficient condition for convergence for all $\mathbf{x}_0$. This condition also guarantees that $\widehat{\mathbf{A}} :\equiv \mathbf{M}^{-1}\mathbf{A} = \mathbf{I} - \widehat{\mathbf{B}}$ is nonsingular.

The same preconditioners $\mathbf{M}$ can also be used with other Krylov space solvers. For example, we could use the matrix $\mathbf{M}$ from a block Jacobi splitting or the one from an SSOR splitting as a left preconditioner in the Chebyshev method or in many of the Krylov space solvers we will discuss later. This simple approach to preconditioning is quite popular. Note that here the preconditioning has the effect that all eigenvalues of $\widehat{\mathbf{A}}$ lie in a circle of radius $\rho(\widehat{\mathbf{B}}) < 1$ around the point 1. We do not iterate with, say block Jacobi or SSOR, but we only use the underlying splitting to obtain a better conditioned matrix $\widehat{\mathbf{A}}$. In some sense, we combine each step of the Krylov space solver with one step of the iteration based on splitting.

The SOR splitting is not appropriate for preconditioning. In this case, as we mentioned, if $\mathbf{A}$ has Property A and is consistently ordered, and if $\mathbf{D}^{-1}\mathbf{A}$ has real eigenvalues, then the eigenvalues of $\widehat{\mathbf{B}}$ lie for the optimal ω on the circle with radius $\rho(\widehat{\mathbf{B}}) < 1$ and center 0, those of $\widehat{\mathbf{A}}$ lie on a circle with the same radius but center 1. This kind of spectrum is not very suitable for Krylov space solvers unless $\rho(\widehat{\mathbf{B}})$ is really small.

How more effective are iterative procedures when used as preconditioners? Let us consider, for example, the use of the preconditioned CG method to accelerate a linear iterative solver in the form

$$\mathbf{x}_{m+1} = P^{-1}N\mathbf{x}_m + P^{-1}\mathbf{b}$$

with P symmetric and definite. This process converges provided that

$$\|P^{-1}N\|_A = \|I - P^{-1}A\|_A = q < 1.$$

By simple computations, it is possible to show that the condition number of the matrix $P^{-1}A$ can be bounded by

$$\kappa(P^{-1}A) \leq \frac{1+q}{1-q}.$$

Therefore, the application of preconditioned CG method with preconditioner P^{-1} yields a superior convergence factor, as

$$\frac{\sqrt{\kappa(P^{-1}A)}-1}{\sqrt{\kappa(P^{-1}A)}+1} \leq \frac{\sqrt{\frac{1+q}{1-q}}-1}{\sqrt{\frac{1+q}{1-q}}+1} \leq \frac{1-\sqrt{1-q^2}}{q} \leq q \quad \forall q \in (0,1).$$

10.2 Incomplete LU and Cholesky factorizations

If we knew a Gaussian LU factorization of $\mathbf{A}$, say $\mathbf{A} = \mathbf{P}^\mathsf{T}\mathbf{L}\mathbf{U}$ with a permutation matrix $\mathbf{P}$, a lower triangular matrix $\mathbf{L}$, and an upper triangular matrix $\mathbf{U}$, we could choose

$$\mathbf{M} :\equiv \mathbf{P}^\mathsf{T}\mathbf{L}\mathbf{U} \tag{10.4}$$

as a left preconditioner, which means that $\widehat{\mathbf{A}} = \mathbf{M}^{-1}\mathbf{A} = \mathbf{I}$ would be optimally preconditioned. Application of this preconditioner would require to forward substitute with $\mathbf{U}$, back substitute with $\mathbf{L}$, and to permute components according to $\mathbf{P}$. But application of $\mathbf{M}^{-1}$ to $\mathbf{r}_0$ would yield in one step

$$\mathbf{x}_\star = \mathbf{x}_0 - \mathbf{d}_0 = \mathbf{x}_0 + \mathbf{M}^{-1}\mathbf{r}_0\,. \tag{10.5}$$

Of course, when we choose $\mathbf{x}_0 := \mathbf{o}$, so that $\mathbf{r}_0 = \mathbf{b}$, then this is essentially just the application of Gauss elimination.

As we mentioned in Chapter 8 the problem with the LU factorization is that for most large sparse matrices (except for banded ones with a very dense band) this approach is inefficient because the LU factors are much denser than $\mathbf{A}$ and thus their computation is costly and their memory requirement is large. The set of additional nonzero elements in $\mathbf{L}$ and $\mathbf{U}$ in positions of zero elements in $\mathbf{A}$ is called **fill-in**.

An often very effective alternative is to compute an approximate LU factorization that is as sparse as $\mathbf{A}$ or at least nearly as sparse: we choose sparse matrices $\mathbf{L}$ and $\mathbf{U}$ and a permutation matrix $\mathbf{P}$ so that the difference $\mathbf{L}\mathbf{U} - \mathbf{P}\mathbf{A}$ is small. The product $\mathbf{M} :\equiv \mathbf{P}^\mathsf{T}\mathbf{L}\mathbf{U} \approx \mathbf{A}$ is then called an **incomplete LU (ILU) factorization** [*unvollständige LU-Zerlegung*]. For the case of spd and Hpd matrices there are analogous products $\mathbf{M} :\equiv \mathbf{L}\mathbf{L}^\mathsf{T} \approx \mathbf{A}$ called **incomplete Cholesky (IC) factorization** [*unvollständige Cholesky-Zerlegung*].

There are many variants of ILU and IC factorizations. Often, no pivoting is used even in the unsymmetric ILU decomposition; that is, $\mathbf{P} = \mathbf{I}$. In the simplest variant we compute an LU or a Cholesky factorization, but where $\mathbf{A}$ has a zero element we replace any nonzero element of $\mathbf{L}$ or $\mathbf{U}$ by a zero. That is, zero fill-in is enforced.

The next step of sophistication is to prescribe some **pattern** [*Muster*] P of forced zeros:

$$\boxed{P \subseteq \left\{(i,j) \;\middle|\; i \neq j,\ a_{i,j} = 0,\ 1 \leq i \leq N,\ 1 \leq j \leq N\right\}\,.} \tag{10.6}$$

One version of the corresponding ILU factorization algorithm without pivoting looks as follows:

Algorithm 10.1 (ILU FACTORIZATION WITH FIXED PATTERN P)**.**

```
for $k = 1, \ldots, n-1$ do
  for $i = k+1, \ldots, n$ do
    if $(i,k) \notin P$,
      $a_{ik} := a_{ik}/a_{kk}$ ;
      for $j = k+1, \ldots, n$ do
        if $(i,j) \notin P$,
          $a_{ij} := a_{ij} - a_{ik} * a_{kj}$ ;
        endif
      endfor
    endif
  endfor
endfor
```

This simple ILU factorisation is known as ILU(0). Although effective, in some cases the accuracy of the ILU(0) may be insufficient to yield an adequate rate of convergence. More accurate factorisations will differ from ILU(0) by allowing some *fill-in*. The resulting class of methods is called ILU(f), where f is the level-of-fill. A level-of-fill is attributed to each element that is processed by Gaussian elimination, and dropping will be based on the level-of-fill. The level-of-fill should be indicative of the size of the element: the higher the level-of-fill, the smaller the elements. A simple model can be employed to justify to effectiveness of this approach is detailed in [Saa96a, Section 10.3.3].

Alternative dropping techniques can be based on the numerical size of the element to be discarded. Numerical dropping strategies generally yield more accurate factorisations with the same amount of fill-in than level-of-fill methods. The general strategy is to compute an entire row of the $\tilde{L}$ and $\tilde{U}$ matrices, and then keep only the biggest entries in a certain number. In this way, the amount of fill-in is controlled; however, the structure of the resulting matrices is undefined. These factorisations are usually referred to as ILUT, and a variant which performs pivoting is called ILUTP. Many other variants have been presented in literature; see for instance [Axe94, Saa96a, Cv97].

Of course, there is no guarantee that ILU decompositions exist, even when $\mathbf{A}$ is nonsingular, since we do not include pivoting here. But worse, the ILU decomposition may not exist even when the full LU decomposition of $\mathbf{A}$ exists. For example, even when $\mathbf{A}$ is spd, the ILU decomposition or the corresponding IC decomposition need not exist. But there are classes of matrices for which it can be shown that the ILU decomposition exists, at least in exact arithmetic.

It is easy to show that after the ILU decomposition — if it can be completed — holds

$$\boxed{\mathbf{A} = \mathbf{LU} - \mathbf{R}\,,} \tag{10.7}$$

where

$$\boxed{\begin{array}{llll} l_{ij} = 0 & \text{if} \quad i < j & \text{or} & (i,j) \in P\,, \\ u_{ij} = 0 & \text{if} \quad i < j & \text{or} & (i,j) \in P\,, \\ r_{ij} = 0 & \text{if} \quad (i,j) \notin P\,. & & \end{array}} \tag{10.8}$$

If one wants to avoid the assumption $(i,j) \in P \implies a_{i,j} = 0$ in the definition of P, then one needs the following assignment before executing the algorithm

$$\forall (i,j) \in P \text{ with } a_{ij} \neq 0 : \quad r_{ij} := -a_{ij}\,, \quad a_{ij} := 0\,. \tag{10.9}$$

But this is hardly ever required in practice.

In other versions of ILU algorithms the pattern P is not fixed in advance, but depends on the sizes of the matrix elements that are constructed: any constructed small element of $\mathbf{L}$ or $\mathbf{U}$ is deleted on the spot by comparison with a threshold (drop tolerance) T. This version is called **ILUT**. A detailed treatment of various ILU algorithms is given in [Saa96b].

MATLAB provides for example:

`[L,U,P] = luinc(A,'0')`: ILU with pivoting, no fill-in
`[L,U,P] = luinc(A,droptol)`: ILUT with pivoting
`[L,U,P] = cholinc(A,'0')`: IC with no fill-in
`[L,U,P] = cholinc(A,droptol)`: IC with drop tolerance

10.3 Polynomial preconditioning

Given $\mathbf{A}$ and $\mathbf{c}$, a Krylov space solver applied to $\mathbf{A}\mathbf{w} = \mathbf{c}$ delivers approximations $\mathbf{w}_\ell$ of $\mathbf{w}_\star :\equiv \mathbf{A}^{-1}\mathbf{c}$, and according to (9.43) we have, when choosing $\mathbf{w}_0 := \mathbf{o}$,

$$\mathbf{w}_\ell = q_{\ell-1}(\mathbf{A})\mathbf{c} \in \mathcal{K}_\ell(\mathbf{A}, \mathbf{c})\,. \tag{10.10}$$

For some Krylov space solvers, the polynomial $q_{\ell-1} \in \mathcal{P}_{\ell-1}$ depends on the right-hand side $\mathbf{c}$, but in others, like Jacobi or Chebyshev iteration, it does not. Let us assume the latter case. Then $q_{\ell-1}(\mathbf{A})$ can be viewed as an operator that maps any $\mathbf{c}$ into an approximation of $\mathbf{A}^{-1}\mathbf{c}$. In particular, if $\mathbf{c} := \mathbf{A}\mathbf{w}$ is considered as an image point of $\mathbf{A}$ (and if $\mathbf{A}$ is nonsingular, it always can be considered so), then $q_{\ell-1}(\mathbf{A})\mathbf{A}$ can be viewed as an approximation of the identity. So, $\mathbf{C} := q_{\ell-1}(\mathbf{A})$ is an approximate inverse of $\mathbf{A}$, which can be used for preconditioning.

In summary: *given any Krylov space solver whose recurrence coefficients only depend on* $\mathbf{A}$ *but not on* $\mathbf{c}$*, and given any fixed iteration number* ℓ*, if* $q_{\ell-1} \in \mathcal{P}_{\ell-1}$ *is the polynomial representing the* ℓ*th iterate* $\mathbf{w}_\ell$ *when solving* $\mathbf{A}\mathbf{w} = \mathbf{c}$*, then* $\mathbf{C} := q_{\ell-1}(\mathbf{A})$ *is an approximate inverse of* $\mathbf{A}$.

To implement this preconditioner for computing, say, $\mathbf{C}\mathbf{A}\mathbf{z}$, we first compute $\mathbf{c} := \mathbf{A}\mathbf{z}$ and then perform ℓ steps of the Krylov space solver applied to $\mathbf{A}\mathbf{w} = \mathbf{c}$; so that $\mathbf{w}_\ell = \mathbf{C}\mathbf{A}\mathbf{z}$.

As early as 1959, it was suggested by [Rut59] to use Chebyshev iteration in this way as a preconditioner (although the notion of "preconditioning" did not yet exist at that time).

10.4 Inner-outer iteration

A natural generalization of polynomial preconditioning is to allow to use any Krylov space solver, even one where $q_{\ell-1}$ depends on $\mathbf{c}$, and also to allow ℓ to vary from one application of $\mathbf{C}$ to the next; so we should write $\mathbf{C}_n$ when applying the **inner iteration** [*innere Iteration*] the nth time, that is, in the nth step of the **outer iteration** [*äussere Iteration*]. Instead of choosing ℓ in advance, we may then terminate each inner iteration when the inner residual

$$\mathbf{c}_n - \mathbf{A}\mathbf{w}_{n,\ell} = \mathbf{c}_n - \mathbf{A}\,\underbrace{q_{n,\ell_n-1}(\mathbf{A})}_{\equiv:\,\mathbf{C}_n}\,\mathbf{c}_n = \underbrace{\left(\mathbf{I} - \mathbf{A}q_{n,\ell_n-1}(\mathbf{A})\right)}_{\equiv:\,p_{n,\ell_n}(\mathbf{A})}\mathbf{c}_n$$

is considered small enough. It need not be very small. The combination of such a **flexible preconditioning** [*flexible Vorkonditionierung*] with a Krylov space solver as outer iteration is called **inner-outer iteration** and has become very fashionable recently.

10.5 Sparse Approximate Inverse Preconditioners (SPAI)*

Rather than constructing an approximation $\mathbf{P}$ of $\mathbf{A}$, an alternative is to build $\mathbf{M} = \mathbf{P}^{-1}$ which approximates $\mathbf{A}^{-1}$ directly. When $\mathbf{M}$ is available, no inversion or solve operations will be required on the preconditioning stage. Preconditioners based on this technique are called "explicit" preconditioners, while those based on the former are called "implicit."

To approximate $\mathbf{A}^{-1}$ is a difficult task since this matrix is in general dense, and moreover the construction should be done in parallel. One possible way to operate is to minimise the Frobenius norm of the residual matrix $\mathbf{I} - \mathbf{A}\mathbf{M}_F$. An important feature of this objective function is that it can be decoupled as the sum of squares of the 2-norm of the n individual column:

$$\mathbf{I} - \mathbf{A}\mathbf{M}_F^2 = \sum_{j=1}^{n} \mathbf{e}_j - \mathbf{A}\mathbf{m}_{j}{}_2^2, \tag{10.11}$$

in which $\mathbf{e}_j$ and $\mathbf{m}_j$ are the $j-$th column of the identity matrix $\mathbf{I}$ and of matrix $\mathbf{M}$, respectively. Hence, one can minimise the individual functions $\mathbf{e}_j - \mathbf{A}\mathbf{m}_{j}{}_2^2$. This approach has been proposed by Grote and Huckle [GH97]. Of course, the exact inverse will be found if no restriction is placed on $\mathbf{M}$. Usually, a sparsity pattern (that is, the set of non-zero indexes) for $\mathbf{M}$ is prescribed (or other dropping strategies as in ILU preconditioners), or minimisation is performed using an iterative method like GMRES, implemented with sparse matrix-vector and sparse vector-vector operators.

A notable disadvantage of this approach is that it is difficult to assess in advance whether or not the resulting matrix will be non-singular. An alternative is to seek a two-sided approximation, i.e. a pair of $\mathbf{L}$ and $\mathbf{U}$ respectively lower triangular and upper triangular, which attempt to minimise the objective function

$$\|\mathbf{I} - \mathbf{LAU}\|_F^2.$$

Gould and Scott [GS98] indicate that the SPAI preconditioner may be a good alternative to ILU, but it is more expensive to compute both in terms of time and storage, at least if computed sequentially. Thus, the use of SPAI preconditioner can be of interest when used for several right-hand sides.

References about SPAI methods can be found in [BMT96, BT98, Zha98, Cho00]. For a comparative study of various SPAI preconditioners we refer to [BT99].

10.6 Domain Decomposition Preconditioners

A class of preconditioners, well-suited for parallel computations, is based on the *domain decomposition* (DD) approach. The basic idea of DD methods is to decompose the computational domain Ω into M smaller parts Ω_i, $i = 1, \ldots, M$, called subdomains, such that

$$\overline{\Omega} = \bigcup_{i=1}^{M} \overline{\Omega}_i \quad \text{and} \quad \Omega_i \cap \Omega_j = \emptyset \quad \text{for} \quad i \neq j. \tag{10.12}$$

Next, the original problem can be reformulated within each subdomain Ω_i, of smaller size. This family of subproblems is coupled one to another through the values of the unknown solution at subdomain interface. This coupling is then removed at the expense of introducing an iterative process which involves, at each step, solutions on the Ω_i with additional interface conditions on $\partial\Omega_i \setminus \partial\Omega$.

Note that the term DD can embrace a large variety of numerical schemes. Each subdomain may refer to a different physical region, modelled with different physical or numerical models. This leads to the so-called heterogeneous DD; see [QV99, Chap. 8]. On the other hand, when each subdomain is modelled and discretized using the same equations and numerical model, one has a homogeneous domain decomposition.

The class of homogeneous DD methods is rather large. A common classification divides these DD methods into two groups [SBG96b, CM94, SBG96a, QV99]:

- In the first group, named after Schwarz, the computational domain is subdivided into *overlapping* subdomains (as depicted on the left of Figure 10.1), and local Dirichlet-type problems are then solved on each subdomain. The “communication” between the solutions on the different subdomains is here guaranteed by the overlapping region. This method has been proposed by H.-A. Schwarz in 1870 to prove the existence of solutions of elliptic PDEs on domains of complex shape, that is, domains for which no exact solution of the problem is available [Sch70].

- The second group uses *non-overlapping* subdomains (as shown on the right of Figure 10.1). It is thus possible to decompose the unknowns into two sets: one formed by the unknowns on the interface between subdomains, the other formed by unknowns associated with nodes internal to the subdomains. At the algebraic level, one may then compute the Schur complement (SC) matrix by “condensing” the unknowns in the second set. The system is then solved by first computing the values of the interface unknowns and then solving the independent problems for the internal unknowns.

These DD methods are usually rather inefficient when used as solvers of the linear problem; however, they can be reformulated as efficient parallel preconditioners. Note that all the state-of-the-art DD preconditioners consist of *local* and *global* components. The local part, acting at the subdomain level, captures the

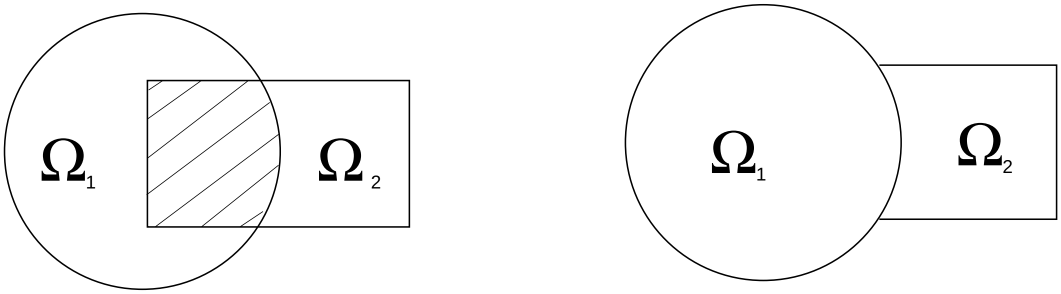


Figure 10.1: Example of overlapping (left) and non-overlapping (right) DD. Overlapping area is shaded.

strong couplings that appear between neighbouring subdomains, while the global part provide an overall – although inexpensive – communication among the subdomains. Should a preconditioner act only locally, then the iterative method that uses such a local preconditioner will have convergence rate that depends strongly on the number of the subdomains.

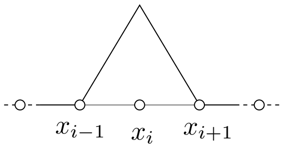
The global component is usually referred to as a "coarse space correction", since usually it is defined on a space that is coarse with respect to the fine space containing the solution. The complexity of this auxiliary problem is much lower than that of the original problem, and its role is to diffuse information among the subdomains. In an analogous manner to multigrid methods, this coarse space is used to correct the "smooth" part of the error, whereas the (local) preconditioner is used to damp the "high-frequency" part.

[1D Poisson problem] For the sake of clarity, we will illustrate the notation considering the 1D Poisson problem

$$\begin{array}{rcll} -u''(x) & = & f(x) & x \in \Omega = (0,1) \\ u(x) & = & 0 & x \in \Gamma = \partial\Omega \end{array} \tag{10.13}$$

Following the FEM principle introduced in Chapter 6, we start by discretising the computational domain Ω by subdividing the unit interval into disjoint subintervals $[ih, (i+1)h]$ of width $h = 1/N$ each. The element functions that we use to model the solution are the hat functions ψ_i centered at $x_i = ih$ and formaly defined as

$$\psi_i(x) = \begin{cases} h^{-1}(x - x_{i-1}), & x_{i-1} \le x \le x_i \\ 1 - h^{-1}(x - x_i), & x_i \le x \le x_{i+1} \\ 0, & \text{otherwise} \end{cases}$$



spanning the function space

$$V_h = \{\psi_1, \psi_2, \ldots, \psi_{N-1}\} \tag{10.14}$$

of piecewise linear functions.

Finally, we define the associated bilinear forms $a(\cdot,\cdot)$ and the linear form $b(\cdot)$ as

$$a(v,u) = \int_\Omega v'(x)u'(x)\,d\Omega \quad \text{and} \quad b(v) = \int_\Omega v(x)f(x)\,d\Omega, \tag{10.15}$$

which are used to construct the discrete representation of the variational problem

$$\mathbf{A}\mathbf{x} = \mathbf{b} \quad \text{where} \quad \mathbf{A}_{i,j} = a(\psi_i, \psi_j),\ \mathbf{b}_i = b(\psi_i). \tag{10.16}$$

Note that the homogeneous Dirichlet boundary conditions are satisfied by *not* including ψ_0 and ψ_N into the space V_h.

10.6.1 One-level Schwarz Preconditioners

The simplest Schwarz preconditioner is the one-level preconditioner, defined as follows. We divide Ω into M subdomains $\{\Omega_i\}_{i=1}^M$ such that $\overline{\Omega} = \bigcup_{i=1}^M \overline{\Omega}_i$ and with an overlap width of the order of $\delta = \xi h$, where $\xi \in \mathbb{N}^+$ and h denotes the meshwidth of the geometry discretisation. The value $\xi = 1$ corresponds to the *minimal overlap*, that is, in that case we mean that the overlap among the subdomains is of one element. From an algorithmic point of view, we can think to subdivide Ω into M non-overlapping subdomains $\hat{\Omega}_i$, and then extend each of them to Ω_i by adding all layers of elements in Ω within a distance δ from $\hat{\Omega}_i$. See Figure 10.2 for an illustration of a 1D and a 2D domain Ω partitioned into several non-overlapping regions $\hat{\Omega}_i$, on the left, and into extended subdomains, on the right.

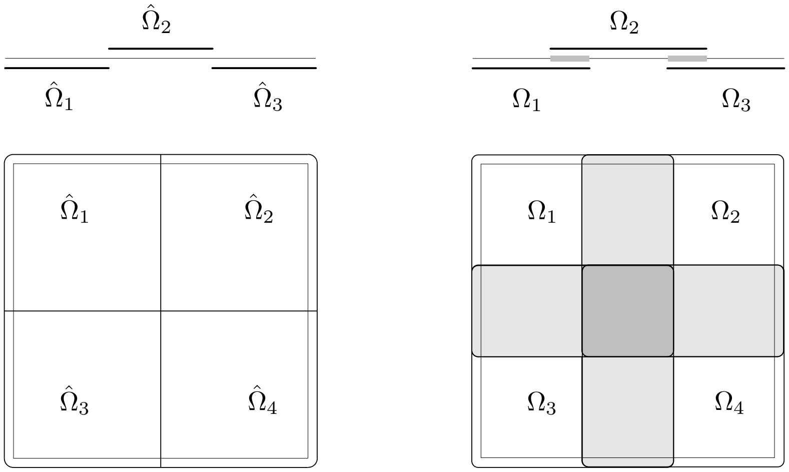


Figure 10.2: Decomposition of the domain Ω into subdomains. *(top/left)* The 1D domain is decomposed into 3 non-overlapping subdomains $\hat{\Omega}_i$. *(top/right)* By expanding the subdomains, yielding regions Ω_i, we obtain simple overlaps depicted in grey. *(bottom/left)* The 2D domain is decomposed into 4 non-overlapping subdomains $\hat{\Omega}_i$. *(bottom/right)* By expanding the subdomains, yielding regions Ω_i, we obtain simple and multiple overlaps depicted in grey.

By notation, let H be the linear size of the subdomains, $H = \max\{\mathrm{diam}(\Omega_i)\}$. We assume that all subdomains have comparable size, i.e., $C_1 H \leq \mathrm{diam}(\Omega_i) \leq C_2 H$, where C_1 and C_2 are positive constants independent of H.

Now consider the discretised problem formulation (10.14), (10.15), (10.16). V_h is the standard finite element space spanned by the basis functions associated with the nodes in $\Omega \setminus \partial\Omega$. Let $V_i \subset V_h$ be the finite element space spanned by the basis function associated to the nodes in $\Omega_i \setminus \partial\Omega_i$, and let n_i be the dimension of V_i. Moreover, let R_i be the restriction operator $V_h \to V_i$ and $\mathbf{R}_i$ the associated $n_i \times n$ matrix. $\mathbf{R}_i^{\mathrm{T}}$ is the injection map, more precisely $\mathbf{R}_i^{\mathrm{T}}$ is a $n \times n_i$ matrix whose action extends by zero a vector with nodal values in $\Omega_i \setminus \partial\Omega_i$.

[1D Poisson problem continued] After having discretised the computational domain, we form the function subspaces V_i by grouping the appropriate element functions ψ_j. To this end, we define the quantities m_k which correspond to the numbers of subintervals that belong to the different subdomains only. These quantities satisfy the relation

$$\sum_{k=1}^{M} m_k = N - (M-1)\xi \tag{10.17}$$

and should be chosen about the same size to ensure balanced subdomain sizes. We then con-

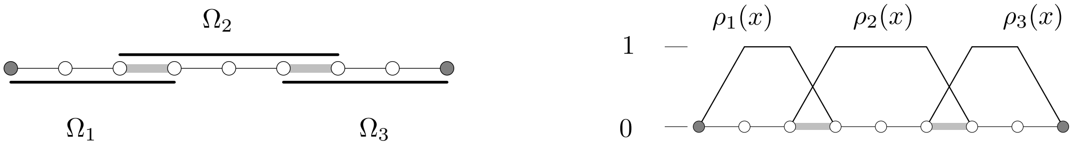


Figure 10.3: Domain decomposition on the discretised geometry. *(left)* Partitioning of the computational domain using $\xi = 1$. *(right)* Definition of the (unit partitioning) weight functions $\rho_k(x)$ which guarantee a "conservation of the solution" in the overlaps.

tinue by defining the start index s_k and end index e_k of each subdomain, i.e.

$$\left.\begin{array}{rcl} s_{k+1} & = & e_k - \xi \\ e_{k+1} & = & e_k + m_{k+1} + \xi \end{array}\right\} \quad \text{with} \quad \left\{\begin{array}{rcl} s_1 & = & 0 \\ e_1 & = & m_1 + \xi \\ e_M & = & N \end{array}\right. \tag{10.18}$$

and with their help we finally define the function subspaces

$$V_i = \{\psi_{s_i+1}, \psi_{s_i+2}, \ldots, \psi_{e_i-1}\} \quad \text{where} \quad n_i = e_i - s_i - 1. \tag{10.19}$$

Note that this choice implies

$$H_{\max} = \max_k H_k = h\Big(\left\lceil \frac{N}{M} \right\rceil + 2\xi \Big). \tag{10.20}$$

For the sake of clarity, we consider the concrete example shown in Figure 10.3 (left). The chosen decomposition leads to the values

$$\mathbf{m} = (2,2,2), \qquad \begin{array}{rcl} \mathbf{s} & = & (0,2,5) \\ \mathbf{e} & = & (3,6,8) \end{array} \qquad \text{and} \qquad \begin{array}{rcl} V_1 & = & \{\psi_1, \psi_2\} \\ V_2 & = & \{\psi_3, \psi_4, \psi_5\} \\ V_3 & = & \{\psi_6, \psi_7\} \end{array} \tag{10.21}$$

In order to define the restriction matrices $\mathbf{R}_k$, it is convenient to (at least implictly) define the weight functions $\rho_k(x)$, as shown in Figure 10.3 (right). With their help we readily obtain

$$\mathbf{R}_k(i, s_k + i) = \rho_k(x_{s_k+i}) \quad \text{for} \quad i = 1, \ldots, n_k. \tag{10.22}$$

It is now possible to define the local bilinear forms

$$a_i(\cdot,\cdot) : V_i \times V_i \to \mathbb{R}, \quad i = 1, \ldots, M$$

which are in general an approximation of $a(\cdot,\cdot)$ (yet often it coincides with a). Should $a(\cdot,\cdot)$ be coercive, then $a_i(\cdot,\cdot)$ is coercive too. This guarantees that the corresponding algebraic matrix $\mathbf{A}_i$ is non-singular. In the case of $a_i(\cdot,\cdot) \equiv a(\cdot,\cdot)$, the algebraic counterpart of $a_i(\cdot,\cdot)$ is the matrix

$$\mathbf{A}_i = \mathbf{R}_i \mathbf{A} \mathbf{R}_i^{\mathrm{T}}.$$

Let the non-singular symmetric matrix $\tilde{\mathbf{A}}_i$ be either $\mathbf{A}_i$ itself or an approximation of it. Then, the additive (or one-level) Schwarz preconditioner can be written as

$$\mathbf{P}_S^{-1} = \sum_{i=1}^{M} \underbrace{\mathbf{R}_i^{\mathrm{T}} \tilde{\mathbf{A}}_i^{-1} \mathbf{R}_i}_{\mathbf{B}_i}. \tag{10.23}$$

Remark 10.2. *Non-symmetric preconditioners can be obtained using different restriction and prolongation operators. A typical choice consists of using $\mathbf{R}_{i,0}^{\mathrm{T}}$ instead of $\mathbf{R}_i^{\mathrm{T}}$, where $\mathbf{R}_{i,0}^{\mathrm{T}}$ represents the restriction operator for the minimal overlap case. This preconditioner is called RAS [CS99, FS01].*

Preconditioner (10.23) is not scalable. Its convergence rate deteriorates as the number of subdomains increases (i.e., H decreases), as stated by the following theorem.

Theorem 10.1. *Let $\tilde{\mathbf{A}}_i = \mathbf{A}_i$. Then, there exists a positive constant C independent of H and h (but possibly dependent on the operator coefficients) such that*

$$\kappa(\mathbf{P}_S^{-1}\mathbf{A}) \leq C\frac{1}{H\delta},$$

where $\kappa(\mathbf{P}_S^{-1}\mathbf{A})$ denotes the condition number *of $\mathbf{A}$ w.r.t. the inner $\mathbf{P}_S$-product and is defined as*

$$\kappa(\mathbf{P}_S^{-1}\mathbf{A}) := \kappa(\mathbf{P}_S^{-1/2}\mathbf{A}\mathbf{P}_S^{-1/2}) = \frac{\lambda_{\max}(\mathbf{P}_S^{-1/2}\mathbf{A}\mathbf{P}_S^{-1/2})}{\lambda_{\min}(\mathbf{P}_S^{-1/2}\mathbf{A}\mathbf{P}_S^{-1/2})}.$$

Proof. See [DW90]. ■

10.6.2 Two-level Schwarz Preconditioners

The non-scalability of the one-level Schwarz preconditioner is due to the fact that information is exchanged only locally via the overlapping regions, while for elliptic problems the domain of dependence is global. The Green's function is non-zero throughout the whole domain, and some way of transmitting global information is needed to make the algorithm scalable.

In a two-level Schwarz preconditioner, a further correction term $\mathbf{B}_0$ is added to the corrections on the subdomains, obtaining

$$\mathbf{P}_C^{-1} = \mathbf{B}_0 + \mathbf{P}_S^{-1} = \sum_{i=0}^{M} \mathbf{B}_i, \tag{10.24}$$

where $\mathbf{B}_0 = \mathbf{R}_0^{\mathrm{T}}\mathbf{A}_0^{-1}\mathbf{R}_0$ is the coarse level correction. $\mathbf{A}_0$ corresponds to the solution of the original variational problem in the space V_0, which is "coarse" in the sense that it contains a limited number of degrees of freedom so to make the "exact" inversion of $\mathbf{A}_0$ computationally acceptable – if n_0 is the dimension of the coarse space, one has $n_0 \ll n$.

Remark 10.3. *To build up the stiffness matrix on the coarse grid, one has two possibilities. The first one is to assemble the coarse grid exactly in the same way as on the fine level. This can be easily implemented in simple cases, whereas it is more difficult if the coefficient exhibits rapid variation since in that case data must be scaled down to the coarse grid. The second possibility is to build up the coarse matrix using the method called the Galerkin approximation,*

$$\mathbf{A}_0 = \mathbf{R}_0\mathbf{A}\mathbf{R}_0^{\mathrm{T}}. \tag{10.25}$$

This can be performed using matrix multiplication techniques.

Since $\mathrm{rank}(\mathbf{R}_0^{\mathrm{T}}\mathbf{A}_0^{-1}\mathbf{R}_0) = n_0$, $\mathbf{B}_0$ has a large null space. Therefore, it cannot be directly used as a preconditioner: any component of the error which lies in the null space of $\mathbf{R}_0^{\mathrm{T}}\mathbf{A}_0^{-1}\mathbf{R}_0$ is never corrected. Therefore, the formulation of equation (10.24) completes $\mathbf{B}_0$. Note that $\mathbf{B}_0$ reduces only the components of the error that can be represented on the coarse space, that is, the low frequencies, whereas $\mathbf{P}_S$ corrects the high frequencies.

The spectral properties (and the parallel performances) of two-level Schwarz preconditioners will depend strongly on the definition of the coarse space V_0. There are virtually unlimited choices of the coarse grid correction that may be used. Convergence of the entire scheme will depend on the particular interpolation and coarse grid operator used. Whenever possible, the coarse space V_0 should be contained in V_h. Usually, the coarse operator represents the discretisation of the continuous problem on a very coarse mesh. This is the common case, for instance, of structured grids like the one depicted in Figure 10.4. In this case, the following result holds.

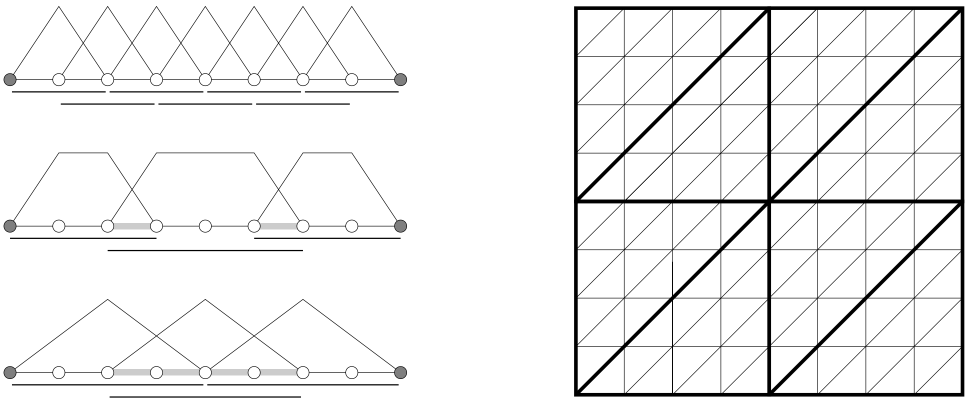


Figure 10.4: Example of coarse mesh selection. *(left)* The original mesh (top) is either coarsened following an algebraic reasoning (middle) or a geometric one (bottom). *(right)* If a structured grid is available (e.g. for a 2D domain), the elments (triangles) of the fine mesh (thin) are contained in the elements of a coarser mesh (thick), suggesting a geometric approach.

Theorem 10.2. *Consider the additive two-level overlapping Schwarz method, where the overlap is uniform of width $\mathcal{O}(\delta)$, the coarse grid space V_0 corresponds to the finite-element functions on elements of width $\mathcal{O}(H)$, and $V_0 \subset V_h$. Then*

$$\kappa(\mathbf{P}_C^{-1}\mathbf{A}) \le C\left(1+\frac{H}{\delta}\right), \tag{10.26}$$

where C does not depend on h, H and δ and, similar to Theorem 10.1, $\kappa(\mathbf{P}_C^{-1}\mathbf{A})$ denotes the condition number *w.r.t. the inner $\mathbf{P}_C$-product.*

Proof. See [SBG96a]. ■

In a more general setting, with a coarse space which is not embedded in the fine space, it is possible to prove more general (yet usually weaker) theorems. For more details we refer the reader to [CZ96, CGZ99].

> **[1D Poisson problem concluded]** The coarse grid correction $\mathbf{B}_0$ can be implemented in numerous ways, mainly depending on the coarse level restriction $\mathbf{R}_0$.
>
> An *algebraic* way to construct $\mathbf{R}_0$ using the already available restriction mappings $\mathbf{R}_i$ can be formulated as
>
> $$\mathbf{R}_0^{\mathrm{T}} = (\mathbf{R}_1^{\mathrm{T}}\mathbf{1}, \mathbf{R}_2^{\mathrm{T}}\mathbf{1}, \ldots, \mathbf{R}_M^{\mathrm{T}}\mathbf{1}), \tag{10.27}$$
>
> where $\mathbf{1}$ is the vector consisting of all ones. Note that this corresponds to defining the functions
>
> $$\psi_k^{(0)}(x) = \sum_{i=s_k}^{e_k} \rho(x_i)\psi_i(x) \quad \text{yielding} \quad V_0 = \{\psi_1^{(0)}, \psi_2^{(0)}, \ldots, \psi_M^{(0)}\}, \tag{10.28}$$
>
> as shown in Figure 10.4 (left).
>
> This approach typically suffers from a poor conditioning of the matrix $\mathbf{A}_0$ and thus an alternative set of functions $\psi_k^{(0)}$ should be chosen. Hence, let us turn our attention to a *geometric* approach. Following the FEM paradigm and assuming the availability of a coarser mesh (nested into the already used one), one could chose $\psi_k^{(0)}$ to be the set of hat functions on the coarse grid, see Figure 10.4 (left).
>
> We conclude this section by reporting a few numerical results that validate Theorems 10.1 and 10.2. In this case we have
>
> $$\kappa(\mathbf{P}_S^{-1}\mathbf{A}) \quad \le \quad C_1\frac{1}{H\delta} \approx C_1\frac{1}{\xi}NM \tag{10.29}$$
>
> $$\kappa(\mathbf{P}_C^{-1}\mathbf{A}) \quad \le \quad C_2\big(1+\frac{H}{\delta}\big) \approx C_2\frac{1}{\xi}\frac{N}{M} \tag{10.30}$$

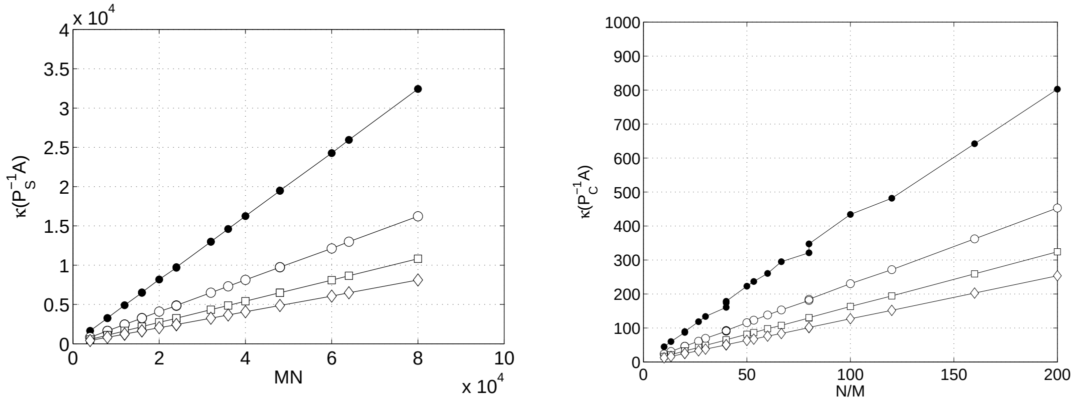


Figure 10.5: Condition number estimates for several overlaps ξ ($\bullet = 1$, $\circ = 2$, $\Box = 3$, $\Diamond = 4$) obtained by a Lanczos procedure. *(left)* Estimate for the one-level Schwarz preconditioner *(right)* Estimate for the two-level additive Schwarz preconditioner

since $H \approx M^{-1}$ and $\delta \approx \xi N^{-1}$. The condition number estimates shown in Figure 10.5 exhibit the behaviour predicted by (10.29) and (10.30), respectively.

10.7 Multigrid Preconditioners

A very important class of methods, mainly developed during the 80's, is the class of *multigrid* (MG) methods, which can be used as solvers and preconditioners for both linear and non-linear systems.

MG methods are based on the observation that the (one-level) schemes presented in Sections 10.1 to 10.5 are well suited for reducing high frequency components of the residual, while they are less effective on the low frequency part. This is why they are often referred to as *smoothers*.

[Smoothing properties of damped Jacobi] We reconsider the 1D Poisson problem described in Section 10.6. Similar to before, we discretise the computational domain Ω by using the grid $G_1 = \{j\, h_1\}_{j=0}^{N_1}$, this time subdividing the unit interval into $N \equiv N_1 = 2^L$ subintervals of width $h_1 = 1/N_1$. Moreover, we choose the hat functions ψ_i as element functions and assemble the corresponding sparse matrix $\mathbf{A} \equiv \mathbf{A}_1$ and the right-hand side $\mathbf{b} \equiv \mathbf{b}_1$ according to (10.16).

For this problem the damped Jacobi iteration represents a simple yet powerful smoother. In fact, given a damping parameter ω, one can perform ν steps of

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \omega \mathbf{D}^{-1}\mathbf{r}_k \quad \text{with} \quad \mathbf{r}_k = \mathbf{b} - \mathbf{A}\mathbf{x}_k, \tag{10.31}$$

which corresponds to performing ν steps of the iteration

$$\mathbf{r}_{k+1} = \underbrace{\left(\mathbf{I} - \omega \mathbf{D}^{-1}\mathbf{A}\right)}_{\mathbf{J}(\omega)} \mathbf{r}_k, \tag{10.32}$$

where $\mathbf{D}$ contains the diagonal entries of $\mathbf{A}$ and $\mathbf{r}_k$ is the residual associated with $\mathbf{x}_k$. By choosing the optimal $\omega = 2/3$, see [Hac85], the Jacobi operator $\mathbf{J}(\omega)$ acts as damping device on the high frequency components of the residual $\mathbf{r}$ only, as shown in Figure 10.6.

Intuitively, one can think of recasting the original problem (and the associated residual in particular) into an auxiliary problem, where the former low part of the spectrum will become the high part of the new spectrum. Analogous to before, these high frequencies can be damped by using an appropriate smoother. The procedure can be recursively repeated yielding a sequence of problems related to the orignal one.

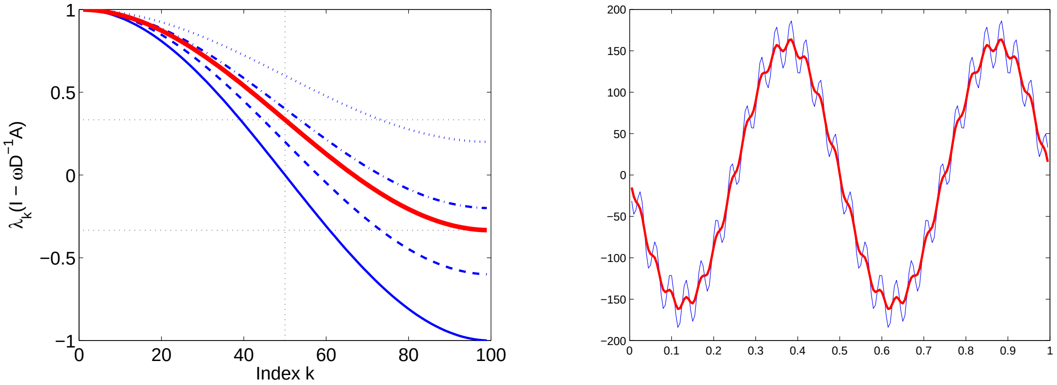


Figure 10.6: Effect of applying a smoother *(left)* The spectrum of the damped Jacobi operator for several values ω (—: 1.0, − −: 0.8, —: 0.67, − · −: 0.6, $\cdots$: 0.4). The optimum is attained at $\omega = 2/3$, where the first eigenvalue in the upper half of the spectrum and the last one have the same magnitude. *(right)* Applying the damped Jacobi iteration to an actual approximation dampens the high frequency coefficients, whereas the low frequency coefficients are preserved.

This sequence of auxiliary problems can be constructed by resorting to geometric or algebraic procedures. In fact, if the original problem is set on a mesh which has been obtained by several refinement steps, one can use this grid hierarchy to define *transfer operators* between the finer and the coarser meshes, then speaking of *geometric multigrid* (GMG) [Bra86]. Conversely, if no such hierarchy is available, transfer operators can algebraicaly be derived by considering the system matrix (on the corresponding level) leading to *algebraic multigrid* (AMG) schemes [RS85]. More about such schemes, their implementations and their implications can be found in [BCF$^+$00], [Wes92], [Hac94] and [St01].

For the sake of simplicity we will restrict ourselves to the simplest case of a GMG which, as in the Domain Decomposition case presented in Section 10.6, will be illustrated using the 1D Poisson problem (10.13). Assume that we are given a sequence of nested grids G_k, $k = 1, \dots, L$, with mesh widths h_k and that we want to compute the solution to the problem

$$\mathbf{A}_k \mathbf{x}_k = \mathbf{b}_k \tag{10.33}$$

defined on the finest mesh ($h \equiv h_1$), where the matrix $\mathbf{A}_1 \in \mathbb{R}^{n_1 \times n_1}$ and the vector $\mathbf{b}_1 \in \mathbb{R}^{n_1}$ stem from a (FE) discretisation say of a PDE problem. Moreover, let

$$\mathbf{R}_k \in \mathbb{R}^{n_{k+1} \times n_k} \quad \text{and} \quad \mathbf{P}_k \in \mathbb{R}^{n_k \times n_{k+1}} \quad (k = 1, \dots, L-1) \tag{10.34}$$

be *restriction* and *prolongation* operators that will be used to transfer vectors between the grids G_k and G_{k+1}.

> **[1D Poisson Problem revisited]** For the 1D Poisson problem, we generate a sequence of grids $G_1, \dots, G_L$ so that $h_k = 2^{k-1} h_1$. The transfer operators $\mathbf{P}_k$ and $\mathbf{R}_k$ are chosen to be the linear interpolation and extrapolation operators between the corresponding grids G_k and G_{k+1} and are defined as
>
> $$\begin{aligned} \mathbf{R}_k(i, [2i-1, 2i, 2i+1]) &= [0.25, 0.5, 0.25] \\ \mathbf{P}_k([2i-1, 2i, 2i+1], i) &= [0.50, 1.0, 0.50]^{\mathrm{T}} \end{aligned}$$
>
> for $k = 1, \dots, L-1$ and $i = 1, \dots, n_k$ where
>
> $$N_{k+1} = \frac{1}{2} N_k \quad \text{and} \quad n_k = N_k - 1. \tag{10.35}$$
>
> Figure 10.7 shows a grid hiearchy with $L = 4$ and illustrates the action of the restriction and prolongation operators.

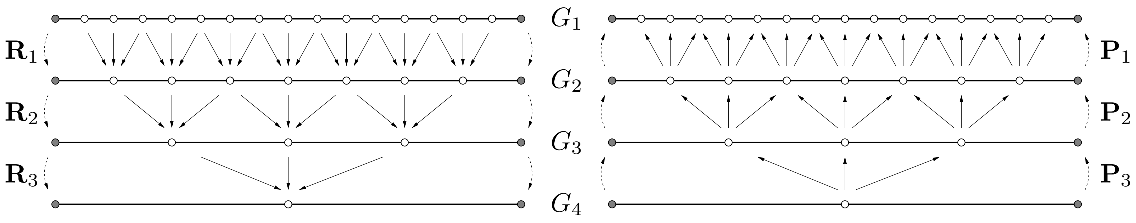


Figure 10.7: Grid hierarchy with associated restriction and prolongation operators.

Given an approximate solution $\mathbf{x}_1$ to problem (10.33) we start by applying the smoother $\mathcal{S}_1$ in order to damp the high frequency components of the residual, i.e.

$$\bar{\mathbf{x}}_1 = \mathcal{S}_1(\mathbf{x}_1, \mathbf{b}). \tag{10.36}$$

Typical smoothers are matrix splitting based methods (Damped Jacobi, SOR, SSOR), incomplete factorisations (ILUT, ILUS, ILUC), polynomial methods, SPAIs and one-level domain decomposition methods. For details we refer the interested reader to [Hac85, BHM00, Wes92, TOS01]. Then, we compute the associated residual

$$\mathbf{r}_1 = \mathbf{b}_1 - \mathbf{A}_1\bar{\mathbf{x}}_1 \tag{10.37}$$

and recall the residual relation

$$\mathbf{A}_1\mathbf{e}_1 = -\mathbf{r}_1, \tag{10.38}$$

where $\mathbf{e}_1 = \bar{\mathbf{x}}_1 - \mathbf{x}_1^\star$ denotes the approximation error and $\mathbf{x}_1^\star$ the exact solution.

At this point we proceed by restricting the residual equation onto the next coarser grid G_2, i.e. we construct the *coarse grid projected problem*

$$\mathbf{A}_2\mathbf{x}_2 = (\mathbf{R}_1\mathbf{A}_1\mathbf{P}_1)\mathbf{x}_2 = -\mathbf{R}_1\mathbf{r}_1 = \mathbf{b}_2. \tag{10.39}$$

Note, that since $\mathbf{r}_1$ is smooth by construction there will be no aliasing artifacts when restricting to the next coarser level.

Once problem (10.39) has been solved, we prolongate the solution $\mathbf{x}_2$, which corresponds to the restricted approximate error $\mathbf{e}_1$, and we add it to the current approximation

$$\hat{\mathbf{x}}_1 = \mathcal{S}_1(\bar{\mathbf{x}}_1 - \mathbf{P}_1\mathbf{x}_2), \tag{10.40}$$

where, in order to alleviate prolongation artifacts, a so called *post smoothing* is performed.

If we apply this idea recursively, i.e. if we (approximately) solve the linear system (10.39) by using the same smooth–restrict–approximate–prolongate–smooth mechanism described in (10.36), (10.39) and (10.40) on the next coarser grids $G_2, \ldots, G_L$, we come up with the well known *V-cycle* for the *multigrid iteration* scheme, whose pseudo code formulation reads

Algorithm 10.2.

```
MGV($\mathbf{A}, \mathbf{b}, \mathbf{x}, k$){
  if $(k = L)$ then
    $\hat{\mathbf{x}} := \mathbf{A}^{-1}\mathbf{b}$
  else
    $\bar{\mathbf{x}} := \mathcal{S}_k(\mathbf{b}, \mathbf{x})$
    $\mathbf{r} := \mathbf{b} - \mathbf{A}\bar{\mathbf{x}}$
    $\mathbf{e} :=$ MGV($\mathbf{R}_k\mathbf{A}\mathbf{P}_k, -\mathbf{R}_k\mathbf{r}, \mathbf{0}, k+1$)
    $\hat{\mathbf{x}} := \mathcal{S}_k(\mathbf{b}, \bar{\mathbf{x}} - \mathbf{P}_k\mathbf{e})$
  endif
  return($\hat{\mathbf{x}}$)
}
```

Bibliography

[Axe94] O. Axelsson. *Iterative Solution Methods*. Cambridge University Press, Cambridge, 1994.

[BCF$^+$00] M. Brezina, A.J. Cleary, R.D. Falgout, V.E. Henson, J.E. Jones, T.A. Manteuffel, S.F. McCormick, and J.W. Ruge. Algebraic multigrid based on element interpolation (AMGe). *SIAM Journal on Scientific Computing*, 22(5):1570–1592, 2000.

[BCT00] M. Benzi, J. K. Cullum, and M. Tůma. Robust approximate inverse preconditioning for the conjugate gradient method. *SIAM Journal on Scientific Computing*, 22(4):1318–1332, 2000.

[BGH05] S. Börm, L. Grasedyck, and W. Hackbusch. *Hierarchical Matrices*. Max–Planck-Institut für Mathematik in den Naturwissenschaften, revised edition, 2005. Lecture notes, `http://www.mis.mpg.de/scicomp/Fulltext/WS_HMatrices.pdf`.

[BH03] M. Bebendorf and W. Hackbusch. Existence of $\mathcal{H}$-matrix approximants to the inverse fe-matrix of elliptic operators with l^∞-coefficients. *Numerische Mathematik*, 95:1–28, 2003.

[BHM00] William L. Briggs, Van Emden Henson, and Steve McCormick. *A multigrid tutorial, Second Edition*. SIAM, Philadelphia, 2000.

[BMT96] M. Benzi, C.D. Meyer, and M. Tůma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM Journal on Scientific Computing*, 17(5):1135–1149, 1996.

[Bra77] A. Brandt. Multi-level Adaptive Solutions to Boundary-Value Problems. *Math. Comp.*, 31:333–390, 1977.

[Bra86] A. Brandt. Algebraic multigrid: the symmetric case. *Appl. Math. Comp.*, 19:23–56, 1986.

[BT98] M. Benzi and M. Tůma. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM Journal on Scientific Computing*, 19(3):968–994, 1998.

[BT99] M. Benzi and M. Tůma. A comparative study of sparse approximate inverse preconditioners. *Applied Numerical Mathematics: Transactions of IMACS*, 30(2–3):305–340, 1999.

[BT03] M. Benzi and M. Tůma. A robust incomplete factorization preconditioner for positive definite matrices. *Numerical Linear Algebra with Applications*, 10:385–400, 2003.

[CGZ99] T.F. Chan, S. Go, and J. Zou. Boundary treatments for multilevel methods on unstructured meshes. *SIAM Journal on Scientific Computing*, 21(1):46–66, 1999.

[Cho00] E. Chow. A priory sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM Journal on Scientific Computing*, 21:1804–1822, 2000.

[CM94] T.F. Chan and T.P. Mathew. Domain decomposition algorithms. In *Acta Numerica*, pages 61–143. Cambridge University Press, 1994.

[CS99] X.-C. Cai and M. Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM Journal on Scientific Computing*, 21:239–247, 1999.

[Cv97] T.F. Chan and H. van der Vorst. Approximate and incomplete factorizations. In D.E. Keyes, A. Sameh, and V. Venkatakrishnan, editors, *Parallel Numerical Algorithms*, ICASE/LaRC Interdisciplinary Series in Science and Engineering, pages 167–202. Kluwer, Dordrecht, 1997.

[CZ96] T.F. Chan and J. Zou. A convergence theory of multilevel additive Schwarz methods on unstructured grids. *Numer. Algorithms*, 13:365–398, 1996.

[DW90] M. Dryja and O.B. Widlund. Towards a unified theory of domain decomposition algorithms for elliptic problems. In T.F. Chan, R. Glowinski, J. Périaux, and O. Widlund, editors, *Third International Symposium on Domain Decomposition Methods for Partial Differential Equations*, pages 3–21. SIAM, Philadelphia, PA, 1990.

[FS01] A. Frommer and D.B. Szyld. An algebraic convergence theory for restricted additive Schwarz methods using weighted max norms. *SIAM Journal on Numerical Analysis*, 39:463–479, 2001.

[GH97] M. J. Grote and Th. Huckle. Parallel preconditioning with sparse approximate inverses. 18(3):838–853, 1997.

[GH03] L. Grasedyck and W. Hackbusch. Construction and arithmetics of $\mathcal{H}$-matrices. *Computing*, 70:295–334, 2003.

[Gre97] A. Greenbaum. *Iterative Methods for Solving Linear Systems*. Frontiers in Applied Mathematics 17. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[GS98] N. Gould and J. Scott. Sparse approximate-inverse preconditioners using norm-minimization techniques. *SIAM Journal on Scientific Computing*, 19(2):605–625, 1998.

[Hac85] W. Hackbusch. *Multi-grid Methods and Applications*. Springer-Verlag, Berlin, 1985.

[Hac94] W. Hackbusch. *Iterative Solution of Large Sparse Linear Systems of Equations*. Springer-Verlag, Berlin, 1994.

[Hac99] W. Hackbusch. A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: Introduction to $\mathcal{H}$-matrices. *Computing*, 62:89–108, 1999.

[QV99] A. Quarteroni and A. Valli. *Domain Decomposition Methods for Partial Differential Equations*. Oxford University Press, Oxford, 1999.

[RS85] J.W. Rube and K. Stüben. Efficient solution of finite difference and finite element equations by algebraic multigrid (AMG). In D. J. Paddon and H. Holstein, editors, *Multigrid Methods for Integral and Differential Equations*, *The Institute of Mathematics and its Application Conference Series*, 3, pages 169–212. Claredon Press, Oxford, 1985.

[RS87] J. Ruge and K. Stuben. Algebraic multigrid (AMG). In S. McCormick, editor, *Multigrid Methods*. Frontiers in Applied Mathematics, 1987.

[Rut59] H. Rutishauser. Theory of gradient methods. In *Refined Iterative Methods for Computation of the Solution and the Eigenvalues of Self-Adjoint Boundary Value Problems*, Mitt. Inst. angew. Math. ETH Zürich, Nr. 8, pages 24–49. Birkhäuser, Basel, 1959.

[Saa96a] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS, Boston, 1996.

[Saa96b] Youcef Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing, Boston, 1996.

[SBG96a] B. Smith, P. Bjorstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel methods for elliptic partial differential equations*. Cambridge University Press, 1996.

[SBG96b] B. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition. Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.

[Sch70] H.-A. Schwarz. Uber einen Grenzübergang durch alternierendes Verfahren. *Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich*, 15:272–286, 1870.

[SHT04] M. Sala, J. Hu, and R. Tuminaro. ML 3.1 smoothed aggregation user's guide. Technical Report SAND-4819, Sandia National Laboratories, September 2004.

[St01] K. Stben. A review of algebraic multigrid. *Journal of Computational and Applied Mathematics*, 128(1-2):281–309, 2001.

[TOS01] U. Trottenberg, C. W. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, Inc., 2001.

[Var00] R. Varga. *Matrix Iterative Analysis, Second Edition*. Springer-Verlag, Berlin, 2000.

[VBM98] P. Vanek, M. Brezina, and J. Mandel. Convergence of Algebraic Multigrid Based on Smoothed Aggregation. Technical Report report 126, UCD/CCM, Denver, CO, 1998.

[Wes92] P. Wesseling. *An Introduction To Multigrid Methods*. Wiley, New York, 1992.

[Zha98] J. Zhang. A sparse approximate inverse technique for parallel preconditioning of general sparse matrices. Technical Report 281-98, Department of Computer Science, University of Kentucky, Lexington, KY, 1998.

Chapter 11

Finite Difference and Finite Element Discretisations of Eigenvalue Problems★

Before we start with the subject of this chapter we want to show how one actually gets to large eigenvalue problems at all. What large means at all. We thereby restict ourselves on problems from physics [CH68, Str86].

11.1 What makes eigenvalues interesting?

Eigenvalues are connected to vibrations. Objects like violin strings, drums, bridges, sky scrapers can swing. They do this at certain frequencies. And in some situations they swing so much that the are destroyed. On November 7, 1940, the Tacoma narrows bridge collapsed, less than half a year after their opening. Strong winds exited the bridge so much that the platform in reinforced concrete fell in pieces. A few years ago the London millenium footbridge started wobbling in a way that it had to be closed. The wobbling had been excited by the pedestrians passing the bridge. These are prominent examples of vibrating structures.

But eigenvalues appear in many other places. Electric fields in cyclotrones, a special form of particle accelerators, have to vibrate in a precise manner, in order to accelerate the charged particles that circle around its center. The solutions of the Schrödinger equation from quantum physics and quantum chemistry have solutions that correspond to vibrations of the, say, molecule it models. The eigenvalues correspond to energy levels that molecule can occupy.

Many characteristic quantities in science *are* eigenvalues,

- decay factors
- frequencies
- norms of operators (or matrices)
- singular values
- condition numbers

In the squel we give a few simple examples that show why computing eigenvalues is important. At the same time we introduce some notation.

11.2 Example 1: The vibrating string

11.2.1 Problem setting

Let us consider a string as displayed in Fig. 11.1. The string is fixed at both ends, at $x = 0$ and $x = L$.

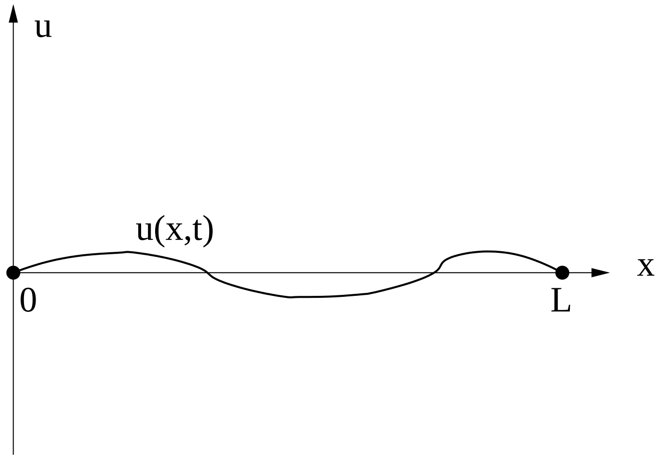


Figure 11.1: The vibrating string

The x-axis coincides with the string's equilibrium position. The displacement of the rest position at x, $0 < x < L$, and time t is denoted by $u(x, t)$.

We will assume that the spacial derivatives of u are not very big:

$$|\frac{\partial u}{\partial x}| \text{ is small.}$$

This assumption entails that we neglect terms of higher order.

Let $v(x, t)$ be the velocity of the string at position x and at time t. Then the kinetic energy of a string section ds of mass $dm = \rho\, ds$ is given by

$$dT = \frac{1}{2} dm\ v^2 = \frac{1}{2} \rho\ ds\ \left(\frac{\partial u}{\partial t}\right)^2. \tag{11.1}$$

From Fig. 11.2 we see that $ds^2 = dx^2 + \left(\frac{\partial u}{\partial x}\right)^2 dx^2$ and thus

$$\frac{ds}{dx} = \sqrt{1 + \left(\frac{\partial u}{\partial x}\right)^2} = 1 + \frac{1}{2} \left(\frac{\partial u}{\partial x}\right)^2 + \text{ higher order terms.}$$

Plugging this into (11.1) and omitting all higher order terms (leaving just the number 1) gives

$$dT = \frac{\rho\ dx}{2} \left(\frac{\partial u}{\partial t}\right)^2.$$

The kinetic energy of the whole string is obtained by integrating over its length,

$$T = \int_0^L dT(x) = \frac{1}{2} \int_0^L \rho(x) \left(\frac{\partial u}{\partial t}\right)^2 dx$$

The potential energy of the string has two components

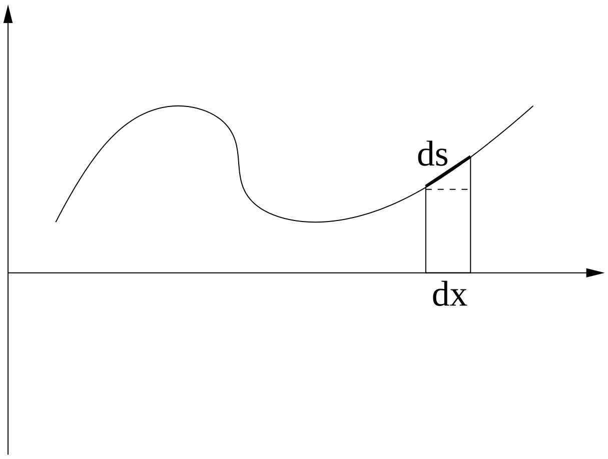


Figure 11.2: The vibrating string

1. the stretching times the excerted strain τ.

$$\tau\int_0^L ds - \tau\int_0^L dx = \tau\int_0^L\left(\sqrt{1+\left(\frac{\partial u}{\partial x}\right)^2}-1\right)dx$$
$$= \tau\int_0^L\left(\frac{1}{2}\left(\frac{\partial u}{\partial x}\right)^2 + \text{ higher order terms}\right)dx$$

2. exterior forces of density f

$$-\int_0^L f u dx$$

Summing, the kinetic energy of the string becomes

$$V = \int_0^L\left(\frac{\tau}{2}\left(\frac{\partial u}{\partial x}\right)^2 - fu\right)dx \tag{11.2}$$

To consider the motion (vibration) of the string in a certain time interval $t_1 \le t \le t_2$ we form the integral

$$\begin{aligned} I(u) &= \int_{t_1}^{t_2}(T-V)\,dt \\ &= \frac{1}{2}\int_{t_1}^{t_2}\int_0^L\left[\rho(x)\left(\frac{\partial u}{\partial t}\right)^2 - \tau\left(\frac{\partial u}{\partial x}\right)^2 - fu\right]dx\,dt \end{aligned} \tag{11.3}$$

Here functions $u(x,t)$ are admitted that are differentiable with respect to x and t and satisfy the **boundary conditions (BC)** that correspond to the fixed string,

$$u(0,t) = u(L,t) = 0, \qquad t_1 \le t \le t_2, \tag{11.4}$$

as well as given **initial conditions** and **end conditions**,

$$\begin{matrix} u(x,t_1) = u_1(x), \\ u(x,t_2) = u_2(x), \end{matrix} \qquad 0 < x < L. \tag{11.5}$$

According to the **principle of Hamilton** a mechanical system with kinetic energy T and potential energy V behaves in a time interval $t_1 \leq t \leq t_2$ for given initial and end positions such that

$$I = \int_{t_1}^{t_2} L \, dt, \qquad L = T - V,$$

is minimized.

Let $u(x,t)$ be such that $I(u) \leq I(w)$ *for all* w, that satisfy the initial, end, and boundary conditions. Let $w = u + \varepsilon\, v$ with

$$v(0,t) = v(L,t) = 0, \qquad v(x,t_1) = v(x,t_2) = 0.$$

v is called a *variation*. We now consider $I(u + \varepsilon\, v)$ as a function of ε. Then we have the equivalence

$$I(u) \text{ minimal} \quad \Longleftrightarrow \quad \boxed{\frac{dI}{d\varepsilon}(u) = 0 \text{ for all admitted } v.}$$

Plugging $u + \varepsilon\, v$ into eq. (11.3) we obtain

$$\begin{aligned} I(u + \varepsilon\, v) &= \frac{1}{2} \int\limits_{t_1}^{t_2} \int\limits_{0}^{L} \left[\rho(x) \left(\frac{\partial (u + \varepsilon\, v)}{\partial t} \right)^2 - \tau \left(\frac{\partial (u + \varepsilon\, v)}{\partial x} \right)^2 - 2f(u + \varepsilon\, v) \right] dx\, dt \\ &= I(u) + \varepsilon \int\limits_{t_1}^{t_2} \int\limits_{0}^{L} \left[\rho(x) \frac{\partial u}{\partial t} \frac{\partial v}{\partial t} - \tau \frac{\partial u}{\partial x} \frac{\partial v}{\partial x} + 2fv \right] dx\, dt + \mathcal{O}(\varepsilon^2). \end{aligned} \tag{11.6}$$

Thus,

$$\frac{\partial I}{\partial \varepsilon} = \int_{t_1}^{t_2} \int_{0}^{L} \left[-\rho \frac{\partial^2 u}{\partial t^2} + \tau \frac{\partial^2 u}{\partial x^2} + 2\, f \right] v \, dx \, dt = 0$$

for all admissible v. Therefore, the bracketed expression must vanish,

$$-\rho \frac{\partial^2 u}{\partial t^2} + \tau \frac{\partial^2 u}{\partial x^2} + 2\, f = 0. \tag{11.7}$$

This last differential equation is named **Euler-Lagrange equation**.

Next we want to solve a differential equation of the form

$$\boxed{\begin{array}{c} -\rho(x) \dfrac{\partial^2 u}{\partial t^2} + \dfrac{\partial}{\partial x} \left(p(x) \dfrac{\partial u}{\partial x} \right) + q(x) u(x,t) = 0. \\ u(0,t) = u(1,t) = 0 \end{array}} \tag{11.8}$$

which is a generalization of the Euler-Lagrange equation (11.7) Here, $\rho(x)$ plays the role of a mass density, $p(x)$ of a locally varying elasticity module. We do not specify initial and end conditions for the moment.

From physics we know that $\rho(x) > 0$ and $p(x) > 0$ for all x. These properties are of importance also from a mathematical view point! For simplicity, we assume that $\rho(x) = 1$.

11.2.2 The method of separation of variables

For the solution u in (11.8) we make the *ansatz*

$$u(x,t) = v(t) w(x). \tag{11.9}$$

Here, v is a function that depends only on the time t, while w depends only on the spacial variable x. With this ansatz (11.8) becomes

$$v''(t) w(x) - v(t)(p(x) w'(x))' + q(x) v(t) w(x) = 0. \tag{11.10}$$

Now we *separate* the variables depending on t from those depending on x,

$$\frac{v''(t)}{v(t)} = \frac{1}{w(x)}(p(x)w'(x))' + q(x).$$

This equation holds for any t and x. We can vary t and x independently of each other without changing the value on eech side of the equation. Therefore, each side of the equation must be equal to a constant value. We denote this value by $-\lambda$. Thus, from the left side we obtain the equation

$$-v''(t) = \lambda v(t). \tag{11.11}$$

This equation has the well-known solution $v(t) = a \cdot \cos(\sqrt{\lambda}t) + b \cdot \sin(\sqrt{\lambda}t)$ where $\lambda > 0$ is assumed. The right side of (11.10) gives a so-called **Sturm-Liouville problem**

$$\boxed{-(p(x)w'(x))' + q(x)w(x) = \lambda w(x), \qquad w(0) = w(1) = 0} \tag{11.12}$$

A value λ for which (11.12) has a *non-trivial* solution w is called an **eigenvalue**; w is a corresponding **eigenfunction**. It is known that all eigenvalues of (11.12) are positive. By means of our ansatz (11.9) we get

$$u(x,t) = w(x)\left[a \cdot \cos(\sqrt{\lambda}t) + b \cdot \sin(\sqrt{\lambda}t)\right]$$

as a solution of (11.8). It is known that (11.12) has infinitely many real positive eigenvalues $0 < \lambda_1 \leq \lambda_2 \leq \cdots$, $(\lambda_k \underset{k\to\infty}{\longrightarrow} \infty)$. (11.12) has a non-zero solution, say $w_k(x)$ *only* for these particular values λ_k. Therefore, the general solution of (11.8) has the form

$$u(x,t) = \sum_{k=0}^{\infty} w_k(x)\left[a_k \cdot \cos(\sqrt{\lambda_k}\,t) + b_k \cdot \sin(\sqrt{\lambda_k}\,t)\right]. \tag{11.13}$$

The coefficients a_k and b_k are determined by initial and end conditions. We could, e.g., require that

$$\begin{aligned} u(x,0) &= \sum_{k=0}^{\infty} a_k w_k(x) = u_0(x), \\ \frac{\partial u}{\partial t}(x,0) &= \sum_{k=0}^{\infty} \sqrt{\lambda_k}\, b_k w_k(x) = u_1(x), \end{aligned}$$

where u_0 and u_1 are given functions. It is known that the w_k form an orthogonal basis in the sapce of square integrable functions $L_2(0,1)$. Therefore, it is not difficult to compute the coefficients a_k and b_k.

In concluding, we see that the difficult problem to solve is the eigenvalue problem (11.12). Knowing the eigenvalues and eigenfunctions the general solution of the time dependant problem (11.8) is easy to form.

Eq. (11.12) can be solved analytically only in very special situation, e.g., if all coefficients are constants. In general a *numerical method* is needed to solve the Sturm-Liouville problem (11.12).

11.3 Numerical methods for solving 1-dimensional problems

In this section we consider three methods to solve the Sturm-Liouville problem.

11.3.1 Finite differences

We approximate $w(x)$ by its values at the discrete points $x_i = ih$, $h = 1/(n+1)$, $i = 1, \ldots, n$.

At point x_i we approximate the derivatives by **finite differences**. We proceed as follows. First we write

$$\frac{d}{dx}g(x_i) \approx \frac{g(x_{i+\frac{1}{2}}) - g(x_{i+\frac{1}{2}})}{h}.$$

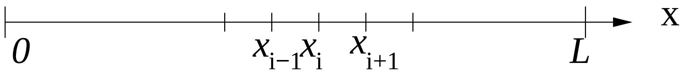


Figure 11.3: Grid points in the interval $(0, L)$.

For $g = p\frac{dw}{dx}$ we get

$$g(x_{i+\frac{1}{2}}) = p(x_{i+\frac{1}{2}})\frac{w(x_{i+1}) - w(x_i)}{h}$$

and finally, for $i = 1, \ldots, n$,

$$\begin{aligned} -\frac{d}{dx}\left(p\frac{dw}{dx}(x_i)\right) &\approx -\frac{1}{h}\left[p(x_{i+\frac{1}{2}})\frac{w(x_{i+1}) - w(x_i)}{h} - p(x_{i-\frac{1}{2}})\frac{w(x_i) - w(x_{i-1})}{h}\right] \\ &= \frac{1}{h^2}\left[p(x_{i-\frac{1}{2}})w_{i-1} + (p(x_{i-\frac{1}{2}}) + p(x_{i+\frac{1}{2}}))w_i - p(x_{i+\frac{1}{2}})w_{i+1}\right]. \end{aligned}$$

Note that at the interval endpoints $w_0 = w_{n+1} = 0$.

We can collect all equations in a matrix equation,

$$\begin{bmatrix} \frac{p(x_{\frac{1}{2}})+p(x_{\frac{3}{2}})}{h^2} + q(x_1) & -p(x_{\frac{3}{2}}) & & \\ -p(x_{\frac{3}{2}}) & \frac{p(x_{\frac{3}{2}})+p(x_{\frac{5}{2}})}{h^2} + q(x_2) & -p(x_{\frac{5}{2}}) & \\ & -p(x_{\frac{5}{2}}) & \ddots & \ddots \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_n \end{bmatrix} = \lambda \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_n \end{bmatrix}$$

or, briefly,

$$A\mathbf{w} = \lambda\mathbf{w}. \tag{11.14}$$

By construction, A ist symmetric and tridiagonal. One can show that it is positive definite as well.

11.3.2 The finite element method

We write (11.12) in the form

> Find a twice differentiable function w with $w(0) = w(1) = 0$ such that
>
> $$\int_0^1 \left[-(p(x)w'(x))' + q(x)w(x) - \lambda w(x)\right]\phi(x)dx = 0$$
>
> *for all* smooth functions ϕ that satisfy $\phi(0) = \phi(1) = 0$.

To relax the requirements on w we integrate by parts and get the new so-called weak form of the problem:

> Find a differentiable function w with $w(0) = w(1) = 0$ such that
>
> $$\int_0^1 \left[-p(x)w(x)'\phi'(x) + q(x)w(x)\phi(x) - \lambda w(x)\phi(x)\right]dx = 0 \tag{11.15}$$
>
> *for all* differentiable functions ϕ that satisfy $\phi(0) = \phi(1) = 0$.

We now write w as the linear combination

$$w(x) = \sum_{i=1}^{n} \xi_i \, \Psi_i(x) \tag{11.16}$$

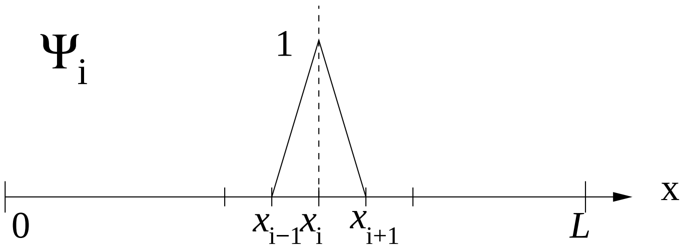


Figure 11.4: A basis function of the finite element space: a hat function.

where

$$\Psi_i(x) = \left(1 - \frac{|x - x_i|}{h}\right)_+ = \max\{0,\ 1 - \frac{|x - x_i|}{h}\}, \tag{11.17}$$

is the function that is linear in each interval (x_i, x_{i+1}) and satisfies

$$\Psi_i(x_k) = \delta_{ik} := \begin{cases} 1, & i = k, \\ 0, & i \neq k. \end{cases}$$

An example of such a basis function, a so-called *hat function*, is given in Fig. 11.4.

We now replace w in (11.15) by the linear combination (11.16), and replace testing ‘against all ϕ’ by testing against all Ψ_k. In this way (11.15) becomes

$$\int_0^1 \left(-p(x)(\sum_{i=1}^n \xi_i\, \Psi_i'(x))\Psi_k'(x) + (q(x) - \lambda)\sum_{i=1}^n \xi_i\, \Psi_i(x)\Psi_k(x)\right)\, dx, \quad \text{for all } k,$$

or,

$$\boxed{\textstyle\sum_{i=1}^n \xi_i \int\limits_0^1 \left(p(x)\Psi_i'(x)\Psi_k'(x) + (q(x) - \lambda)\Psi_i(x)\Psi_k(x)\right)\, dx = 0, \quad \text{for all } k.} \tag{11.18}$$

These last equations are called the **Rayleigh–Ritz–Galerkin** equations. Unknown are the n values ξ_i and the eigenvalue λ. In matrix notation (11.18) becomes

$$A\mathbf{x} = \lambda M\mathbf{x} \tag{11.19}$$

with

$$a_{ij} = \int_0^1 \left(p(x)\Psi_i'\Psi_j' + q(x)\Psi_i\Psi_j\right)\, dx \quad \text{and} \quad m_{ij} = \int_0^1 \Psi_i\Psi_j\, dx$$

For the specific case $p(x) = 1 + x$ and $q(x) = 1$ we get

$$\begin{aligned} a_{kk} &= \int_{(k-1)h}^{kh} \left[(1+x)\frac{1}{h^2} + \left(\frac{x - (k-1)h}{h}\right)^2\right]\, dx \\ &\qquad + \int_{kh}^{(k+1)h} \left[(1+x)\frac{1}{h^2} + \left(\frac{(k+1)h - x}{h}\right)^2\right]\, dx = 2(n+1+k) + \frac{2}{3}\frac{1}{n+1} \\ a_{k,k+1} &= \int_{kh}^{(k+1)h} \left[(1+x)\frac{1}{h^2} + \frac{(k+1)h - x}{h}\cdot\frac{x - kh}{h}\right]\, dx = -n - \frac{3}{2} - k + \frac{1}{6}\frac{1}{n+1} \end{aligned}$$

In the same way we get

$$M = \frac{1}{6(n+1)}\begin{bmatrix} 4 & 1 & & \\ 1 & 4 & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & 4 \end{bmatrix}$$

Notice that both matrices A and M are symmetric tridiagonal and positive definite.

11.3.3 Global functions

Formally we proceed as with the finite element method. But now we choose the $\Psi_k(x)$ to be functions with global support. We could, e.g., set

$$\Psi_k(x) = \sin k\pi x,$$

functions that are differentiable and satisfy the homogeneous boundary conditions. The Ψ_k are eigenfunctions of the nearby problem $-u''(x) = \lambda u(x)$, $u(0) = u(1) = 0$ corresponding to the eigenvalue $k^2\pi^2$. The elements of matrix A are given by

$$\begin{aligned} a_{kk} &= \int_0^1 \left[(1+x)k^2\pi^2\cos^2 k\pi x + \sin^2 k\pi x\right]\, dx = \frac{3}{4}k^2\pi^2 + \frac{1}{2}, \\ a_{kj} &= \int_0^1 \left[(1+x)kj\pi^2 \cos k\pi x \cos j\pi x + \sin k\pi x \sin j\pi x\right]\, dx \\ &= \frac{kj(k^2+j^2)((-1)^{k+j}-1)}{(k^2-j^2)^2}, \quad k \neq j. \end{aligned}$$

11.3.4 A numerical comparison

We consider the above 1-dimensional eigenvalue problem

$$-((1+x)w'(x))' + w(x) = \lambda w(x), \qquad w(0) = w(1) = 0, \tag{11.20}$$

and solve it with the finite difference and finite element methods as well as with the global functions method. The results are given in Table 11.1.

Clearly the global function method is the most powerful of them all. With 80 basis functions the eigenvalues all come right. The convergence rate is exponential.

With the finite difference and finite element methods the eigenvalues exhibit quadratic convergence rates. If the mesh width h is reduced by a factor of $q = 2$, the error in the eigenvalues is reduced by the factor $q^2 = 4$.

11.4 Example 2: The heat equation

The instationary temperature distribution $u(\mathbf{x}, t)$ in an insulated container satisfies the equations

$$\begin{aligned} \frac{\partial u(\mathbf{x},t)}{\partial t} - \Delta u(\mathbf{x},t) &= 0, && \mathbf{x} \in \Omega,\ t > 0, \\ \frac{\partial u(\mathbf{x},t)}{\partial n} &= 0, && \mathbf{x} \in \partial\Omega,\ t > 0, \\ u(\mathbf{x},0) &= u_0(\mathbf{x}), && \mathbf{x} \in \Omega. \end{aligned} \tag{11.21}$$

Here Ω is a 3-dimensional domain[1]with boundary $\partial\Omega$. $u_0(\mathbf{x}), \mathbf{x} = (x_1, x_2, x_3)^T \in \mathbb{R}^3$, is a given bounded, sufficiently smooth function. $\Delta u = \sum \frac{\partial^2 u}{\partial {x_i}^2}$ is called the *Laplace operator* and $\frac{\partial u}{\partial n}$ denotes the derivative of u in direction of the outer normal vector $\mathbf{n}$. To solve the heat equation the **method of separation of variables** is employed. We write u in the form

$$u(\mathbf{x}, t) = v(t)w(\mathbf{x}). \tag{11.22}$$

If a constant λ can be found such that

$$\boxed{\begin{aligned} \Delta w(\mathbf{x}) + \lambda w(\mathbf{x}) &= 0, \quad w(\mathbf{x}) \neq 0, && \mathbf{x} \text{ in } \Omega, \\ \frac{\partial w(\mathbf{x},t)}{\partial n} &= 0, && \mathbf{x} \text{ on } \partial\Omega, \end{aligned}} \tag{11.23}$$

[1] In the sequel we understand a domain to be bounded and simply connected.

Table 11.1 Numerical solutions of problem (11.20)

Finite difference method

k	$\lambda_k(n=10)$	$\lambda_k(n=20)$	$\lambda_k(n=40)$	$\lambda_k(n=80)$
1	15.245	15.312	15.331	15.336
2	56.918	58.048	58.367	58.451
3	122.489	128.181	129.804	130.236
4	206.419	224.091	229.211	230.580
5	301.499	343.555	355.986	359.327
6	399.367	483.791	509.358	516.276
7	492.026	641.501	688.398	701.185
8	578.707	812.933	892.016	913.767
9	672.960	993.925	1118.969	1153.691
10	794.370	1179.947	1367.869	1420.585

Finite element method

k	$\lambda_k(n=10)$	$\lambda_k(n=20)$	$\lambda_k(n=40)$	$\lambda_k(n=80)$
1	15.447	15.367	15.345	15.340
2	60.140	58.932	58.599	58.511
3	138.788	132.657	130.979	130.537
4	257.814	238.236	232.923	231.531
5	426.223	378.080	365.047	361.648
6	654.377	555.340	528.148	521.091
7	949.544	773.918	723.207	710.105
8	1305.720	1038.433	951.392	928.983
9	1702.024	1354.106	1214.066	1178.064
10	2180.159	1726.473	1512.784	1457.733

Global function method

k	$\lambda_k(n=10)$	$\lambda_k(n=20)$	$\lambda_k(n=40)$	$\lambda_k(n=80)$
1	15.338	15.338	15.338	15.338
2	58.482	58.480	58.480	58.480
3	130.389	130.386	130.386	130.386
4	231.065	231.054	231.053	231.053
5	360.511	360.484	360.483	360.483
6	518.804	518.676	518.674	518.674
7	706.134	705.631	705.628	705.628
8	924.960	921.351	921.344	921.344
9	1186.674	1165.832	1165.823	1165.822
10	1577.340	1439.083	1439.063	1439.063

then the product $u = vw$ is a solution of (11.21) if and only if

$$\frac{dv(t)}{dt} + \lambda v(t) = 0, \tag{11.24}$$

the solution of which has the form $a \cdot \exp(-\lambda t)$. By separating variables, the problem (11.21) is divided in two subproblems that are hopefully easier to solve. A value λ, for which (11.23) has a *nontrivial* (i.e. a nonzero) solution is called an *eigenvalue*; w then is called a *corresponding eigenfunction*.

If λ_n is an eigenvalue of problem (11.23) with corresponding eigenfunction w_n, then

$$e^{-\lambda_n t} w_n(\mathbf{x})$$

is a solution of the first two equations in (11.21). It is known that equation (11.23) has *infinitely many* real eigenvalues $0 \leq \lambda_1 \leq \lambda_2 \leq \cdots, (\lambda_n \underset{t\to\infty}{\longrightarrow} \infty)$. Multiple eigenvalues are counted acording to their multiplicity. An arbitrary bounded piecewise continuous function can be represented as a linear combination of the eigenfunctions $w_1, w_2, \ldots$. Therefore, the solution of (11.21) can be written in the form

$$u(\mathbf{x}, t) = \sum_{n=1}^{\infty} c_n e^{-\lambda_n t} w_n(\mathbf{x}), \tag{11.25}$$

where the coefficients c_n are determined such that

$$u_0(\mathbf{x}) = \sum_{n=1}^{\infty} c_n w_n(\mathbf{x}). \tag{11.26}$$

The smallest eigenvalue of (11.23) is $\lambda_1 = 0$ with $w_1 = 1$ and $\lambda_2 > 0$. Therefore we see from (11.25) that

$$u(\mathbf{x}, t) \underset{t\to\infty}{\longrightarrow} c_1. \tag{11.27}$$

Thus, in the limit (i.e., as t goes to infinity), the temperature will be constant in the whole container. The convergence rate towards this equilibrium is determined by the smallest *positive* eigenvalue λ_2 of (11.23):

$$\begin{aligned}\|u(\mathbf{x}, t) - c_1\| = \|\sum_{n=2}^{\infty} c_n e^{-\lambda_n t} w_n(\mathbf{x})\| &\leq \sum_{n=2}^{\infty} |e^{-\lambda_n t}| \|c_n w_n(\mathbf{x})\| \\ &\leq e^{-\lambda_2 t} \sum_{n=2}^{\infty} \|c_n w_n(\mathbf{x})\| \leq e^{-\lambda_2 t} \|u_0(\mathbf{x})\|.\end{aligned}$$

Here we have assumed that the value of the constant function $w_1(\mathbf{x})$ is set to unity.

11.5 Example 3: The wave equation

The air pressure $u(\mathbf{x}, t)$ in a volume with acoustically "hard" walls satisfies the equations

$$\frac{\partial^2 u(\mathbf{x}, t)}{\partial t^2} - \Delta u(\mathbf{x}, t) = 0, \qquad \mathbf{x} \in \Omega, t > 0, \tag{11.28}$$

$$\frac{\partial u(\mathbf{x}, t)}{\partial n} = 0, \qquad \mathbf{x} \in \partial\Omega, t > 0, \tag{11.29}$$

$$u(\mathbf{x}, 0) = u_0(\mathbf{x}), \qquad \mathbf{x} \in \Omega, \tag{11.30}$$

$$\frac{\partial u(\mathbf{x}, 0)}{\partial t} = u_1(\mathbf{x}), \qquad \mathbf{x} \in \Omega. \tag{11.31}$$

Sound propagates with the speed $-\nabla \mathbf{u}$, i.e. along the (negative) gradient from high to low pressure.

To solve the wave equation we proceed as with the heat equation in section 11.4: separation of u according to (11.22) leads again to equation (11.23) but now together with

$$\frac{d^2v(t)}{dt^2} + \lambda v(t) = 0. \tag{11.32}$$

We know this equation from the analysis of the vibrating sting, see (11.11). From there we know that the general solution of the wave equation has the form

$$u(x,t) = \sum_{k=0}^{\infty} w_k(x) \left[a_k \cdot \cos(\sqrt{\lambda_k}\, t) + b_k \cdot \sin(\sqrt{\lambda_k}\, t) \right]. \tag{11.13}$$

where the w_k, $k = 1, 2, \ldots$ are the eigenfunctions of the eigenvalue problem (11.23). The coefficients a_k and b_k are determined by eqrefeq:wave3 and eqrefeq:wave4.

If a harmonic oscillation is forced on the system, an *inhomogeneous* problem

$$\frac{\partial^2 u(\mathbf{x},t)}{\partial t^2} - \Delta u(\mathbf{x},t) = f(\mathbf{x},t), \tag{11.33}$$

is obtained. The boundary and initial conditions are taken from (11.28)–(11.31). This problem can be solved by setting

$$\begin{aligned} u(\mathbf{x},t) &:= \sum_{n=1}^{\infty} \tilde{v}_n(t) w_n(\mathbf{x}), \\ f(\mathbf{x},t) &:= \sum_{n=1}^{\infty} \phi_n(t) w_n(\mathbf{x}). \end{aligned} \tag{11.34}$$

With this approach, $\tilde{v}_n$ has to satisfy equation

$$\frac{d^2\tilde{v}_n}{dt^2} + \lambda_n \tilde{v}_n = \phi_n(t). \tag{11.35}$$

If $\phi_n(t) = a \sin \omega t$, then the solution becomes

$$\tilde{v}_n = A_n \cos \sqrt{\lambda_n} t + B_n \sin \sqrt{\lambda_n} t + \frac{1}{\lambda_n - \omega^2} a \sin \omega t. \tag{11.36}$$

A_n and B_n are real constants that are determined by the initial conditions. If ω gets close to $\sqrt{\lambda_1}$, then the last term can be very large. In the limit, if $\omega = \sqrt{\lambda_n}$, $\tilde{v}_n$ gets the form

$$\tilde{v}_n = A_n \cos \sqrt{\lambda_n} t + B_n \sin \sqrt{\lambda_n} t + at \sin \omega t. \tag{11.37}$$

In this case, $\tilde{v}_n$ is not bounded in time anymore. This phenomenon is called *resonance*. Often resonance is not desirable; it may, e.g., mean the blow up of some structure. In order to prevent resonances eigenvalues have to be known. Possible remedies are changing the domain (the structure).

Remark 11.1. *Vibrating membranes satisfy the wave equation, too. In general the boundary conditions are different from* (11.29)*. If the membrane (of a drum) is fixed at its boundary, the condition*

$$u(\mathbf{x},t) = 0 \tag{11.38}$$

is imposed. This boundary conditions is called Dirichlet boundary conditions*. The boundary conditions in* (11.21) *and* (11.29) *are called* Neumann boundary conditions*. Combinations of these two can occur.*

11.6 Numerical methods for solving the Laplace eigenvalue problem in 2D

In this section we again consider the eigenvalue problem

$$-\Delta u(\mathbf{x}) = \lambda u(\mathbf{x}), \qquad \mathbf{x} \in \Omega, \tag{11.39}$$

with the more general boundary conditions

$$u(\mathbf{x}) = 0, \qquad \mathbf{x} \in C_1 \subset \partial\Omega, \tag{11.40}$$

$$\frac{\partial u}{\partial n}(\mathbf{x}) + \alpha(\mathbf{x})u(\mathbf{x}) = 0, \qquad \mathbf{x} \in C_2 \subset \partial\Omega. \tag{11.41}$$

Here, C_1 and C_2 are *disjoint* subsets of $\partial\Omega$ with $C_1 \cup C_2 = \partial\Omega$. We restrict ourselfs in the following on *two-dimensional* domains and write (x, y) instead of (x_1, x_2).

In general it is not possible to solve a problem of the Form (11.39)–(11.41) exactly (analytically). Therefore one has to resort to numerical approximations. Because we cannot compute with infinitely many variables we have to construct a finite-dimensional eigenvalue problem that represents the given problem as well as possible, i.e., that yields good approximations for the desired eigenvalues and eigenvectors. Since finite-dimensional eigenvalue problem only have a finite number of eigenvalues one cannot expect to get good approximations for all eigenvalues of (11.39)–(11.41).

Two methods for the discretization of eigenvalue problems of the form (11.39)–(11.41) are the *Finite Difference Method* [Ame77, Sch93] and the *Finite Element Method (FEM)* [Sch91, SF73]. We deal with these methods in the following subsections.

11.6.1 The finite difference method

In this section we just want to mediate some impression what the finite difference method is about. Therefore we assume for simplicity that the domain Ω is a square with sides of length 1: $\Omega = (0, 1) \times (0, 1)$. We consider the eigenvalue problem

$$\begin{aligned} &-\Delta u(x, y) = \lambda u(x, y), && 0 < x, y < 1 \\ &u(0, y) = u(1, y) = u(x, 0) = 0, && 0 < x, y < 1, \\ &\frac{\partial u}{\partial n}(x, 1) = 0, && 0 < x < 1. \end{aligned} \tag{11.42}$$

This eigenvalue problem occurs in the computation of eigenfrequencies and eigenmodes of a homogeneous quadratic membrane with three fixed and one free side. It can be solved analytically by separation of the two spatial variables x and y. The eigenvalues are

$$\lambda_{k,l} = \left(k^2 + \frac{(2l-1)^2}{4}\right)\pi^2, \quad k, l \in \mathbb{N},$$

and the corresponding eigenfunctions are

$$u_{k,l}(x, y) = \sin k\pi x \sin \frac{2l-1}{2}\pi y.$$

In the finite difference method one proceeds by defining a rectangular grid with grid points $(x_i, y_j), 0 \le i, j \le N$. The coordinates of the grid points are

$$(x_i, y_j) = (ih, jh), \qquad h = 1/N.$$

By a Taylor expansion one can show for sufficiently smooth functions u that

$$\begin{aligned} -\Delta u(x, y) = \frac{1}{h^2}(&4u(x, y) - u(x-h, y) - u(x+h, y) - u(x, y-h) - u(x, y+h)) \\ &+ O(h^2). \end{aligned}$$

It is therefore straightforward to replace the differential equation $\Delta u(x,y) + \lambda u(x,y) = 0$ by a difference equation at the interiour grid points

$$4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} = \lambda h^2 u_{i,j}, \quad 0 < i,j < N. \tag{11.43}$$

We consider the unknown variables $u_{i,j}$ as approximations of the eigenfunctions at the grid points (i,j):

$$u_{i,j} \approx u(x_i, x_j). \tag{11.44}$$

The Dirichlet boundary conditions are replaced by the equations

$$u_{i,0} = u_{i,N} = u_{0,i}, \qquad 0 < i < N. \tag{11.45}$$

At the points at the upper boundary of Ω we first take the difference equation (11.43)

$$4u_{i,N} - u_{i-1,N} - u_{i+1,N} - u_{i,N-1} - u_{i,N+1} = \lambda h^2 u_{i,N}, \quad 0 \le i \le N. \tag{11.46}$$

The value $u_{i,N+1}$ corresponds to a grid point *outside* of the domain! However the Neumann boundary conditions suggest to reflect the domain at the upper boundary and to extend the eigenfunction symmetrically beyond the boundary. This procedure leads to the equation $u_{i,N+1} = u_{i,N-1}$. Plugging this into (11.46) and multipling the new equation by the factor $1/2$ gives

$$2u_{i,N} - \frac{1}{2}u_{i-1,N} - \frac{1}{2}u_{i+1,N} - u_{i,N-1} = \frac{1}{2}\lambda h^2 u_{i,N}, \quad 0 < i < N. \tag{11.47}$$

In summary, from (11.43) and (11.47), taking into account that (11.45) we get the matrix equation

$$\begin{pmatrix}
4 & -1 & 0 & -1 & & & & & & & & \\
-1 & 4 & -1 & 0 & -1 & & & & & & & \\
0 & -1 & 4 & 0 & 0 & -1 & & & & & & \\
-1 & 0 & 0 & 4 & -1 & 0 & -1 & & & & & \\
 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & & & & \\
 & & -1 & 0 & -1 & 4 & 0 & 0 & -1 & & & \\
 & & & -1 & 0 & 0 & 4 & -1 & 0 & -1 & & \\
 & & & & -1 & 0 & -1 & 4 & -1 & 0 & -1 & \\
 & & & & & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\
 & & & & & & -1 & 0 & 0 & 2 & -\frac{1}{2} & 0 \\
 & & & & & & & -1 & 0 & -\frac{1}{2} & 2 & -\frac{1}{2} \\
 & & & & & & & & -1 & 0 & -\frac{1}{2} & 2
\end{pmatrix}
\begin{pmatrix} u_{1,1} \\ u_{1,2} \\ u_{1,3} \\ u_{2,1} \\ u_{2,2} \\ u_{2,3} \\ u_{3,1} \\ u_{3,2} \\ u_{3,3} \\ u_{4,1} \\ u_{4,2} \\ u_{4,3} \end{pmatrix}
= \lambda h^2
\begin{pmatrix}
1 & & & & & & & & & & & \\
 & 1 & & & & & & & & & & \\
 & & 1 & & & & & & & & & \\
 & & & 1 & & & & & & & & \\
 & & & & 1 & & & & & & & \\
 & & & & & 1 & & & & & & \\
 & & & & & & 1 & & & & & \\
 & & & & & & & 1 & & & & \\
 & & & & & & & & 1 & & & \\
 & & & & & & & & & \frac{1}{2} & & \\
 & & & & & & & & & & \frac{1}{2} & \\
 & & & & & & & & & & & \frac{1}{2}
\end{pmatrix}
\begin{pmatrix} u_{1,1} \\ u_{1,2} \\ u_{1,3} \\ u_{2,1} \\ u_{2,2} \\ u_{2,3} \\ u_{3,1} \\ u_{3,2} \\ u_{3,3} \\ u_{4,1} \\ u_{4,2} \\ u_{4,3} \end{pmatrix}. \tag{11.48}$$

For arbitrary $N > 1$ we define

$$\mathbf{u}_i := \begin{pmatrix} u_{i,1} \\ u_{i,2} \\ \vdots \\ u_{i,N-1} \end{pmatrix} \in \mathbb{R}^{N-1},$$

$$T := \begin{pmatrix} 4 & -1 & & \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix} \in \mathbb{R}^{(N-1)\times(N-1)},$$

$$I := \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{pmatrix} \in \mathbb{R}^{(N-1)\times(N-1)}.$$

In this way we obtain from (11.43), (11.45), (11.47) the discrete eigenvalue problem

$$\begin{pmatrix} T & -I & & \\ -I & T & \ddots & \\ & \ddots & \ddots & -I \\ & & -I & \frac{1}{2}T \end{pmatrix} \begin{pmatrix} \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_3 \\ \mathbf{u}_4 \end{pmatrix} = \lambda h^2 \begin{pmatrix} I & & & \\ & \ddots & & \\ & & I & \\ & & & \frac{1}{2}I \end{pmatrix} \begin{pmatrix} \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_{N-1} \\ \mathbf{u}_N \end{pmatrix} \tag{11.49}$$

of size $N \times (N-1)$. This is a **matrix eigenvalue problem** of the form

$$A\mathbf{x} = \lambda M\mathbf{x}, \tag{11.50}$$

where A and M are *symmetric* and M additionally is *positive definite*. If M is the identity matrix is, we call (11.50) a *special* and otherwise a *generalized* eigenvalue problem. In these lecture notes we deal with numerical methods, to solve eigenvalue problems like these.

In the case (11.49) it is easy to obtain a special (symmetric) eigenvalue problem by a simple transformation: By left multiplication by

$$\begin{pmatrix} I & & & \\ & I & & \\ & & I & \\ & & & \sqrt{2}I \end{pmatrix}$$

we obtain from (11.49)

$$\begin{pmatrix} T & -I & & \\ -I & T & -I & \\ & -I & T & -\sqrt{2}I \\ & & -\sqrt{2}I & T \end{pmatrix} \begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{u}_3 \\ \frac{1}{\sqrt{2}}\mathbf{u}_4 \end{pmatrix} = \lambda h^2 \begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{u}_3 \\ \frac{1}{\sqrt{2}}\mathbf{u}_4 \end{pmatrix}. \tag{11.51}$$

A property common to matrices obtained by the finite difference method are its *sparsity*. Sparse matrices have only very few nonzero elements.

In real-world applications domains often cannot be covered easily by a rectangular grid. In this situation and if boundary conditions are complicated the method of finite differences can be difficult to implement.

Because of this the finite element method is often the method of choice.

11.6.2 The finite element method (FEM)

Let $(\lambda, u) \in \mathbb{R} \times V$ be an eigenpair of problem (11.39)–(11.41). Then

$$\int_\Omega (\Delta u + \lambda u) v \, dx \, dy = 0, \quad \forall v \in V, \tag{11.52}$$

where V is vector space of bounded twice differentiable functions that satisfy the boundary conditions (11.40)–(11.41). By partial integration (Green's formula) this becomes

$$\int_\Omega \nabla u \nabla v \, dx \, dy + \int_{\partial\Omega} \alpha \, u \, v \, ds = \lambda \int_\Omega u \, v \, dx \, dy, \quad \forall v \in V, \tag{11.53}$$

or

$$a(u,v) = (u,v), \qquad \forall v \in V \tag{11.54}$$

where

$$a(u,v) = \int\limits_\Omega \nabla u \, \nabla v \, dx \, dy + \int\limits_{\partial\Omega} \alpha \, u \, v \, ds, \quad \text{and} \quad (u,v) = \int_\Omega u \, v \, dx \, dy.$$

We complete the space V with respect to the Sobolev norm [SF73, AB84]

$$\sqrt{\int_\Omega (u^2 + |\nabla u|^2) \, dx \, dy}$$

to become a Hilbert space H [AB84, Wei74]. H is the space of quadratic integrable functions with quadratic integrable first derivatives that satisfy the Dirichlet boundary conditions (11.40)

$$u(x,y) = 0 \quad (x,y) \in C_1.$$

(Functions in H in general no not satisfy the so-called *natural* boundary conditions (11.41).) One can show [Wei74] that the eigenvalue problem (11.39)–(11.41) is equivalent with the eigenvalue problem

$$\begin{array}{l} \text{Find } (\lambda, u) \in \mathbb{R} \times H \text{ such that} \\ a(u,v) = \lambda (u,v) \quad \forall v \in H. \end{array} \tag{11.55}$$

(The essential point is to show that the eigenfunctions of (11.55) are elements of V.)

The Rayleigh–Ritz–Galerkin method

In the Rayleigh–Ritz–Galerkin method one proceeds as follows: A set of linearly independent functions

$$\phi_1(x,y), \cdots, \phi_n(x,y) \in H, \tag{11.56}$$

are chosen. These functions span a *subspace* S of H. Then, problem (11.55) is solved where H is replaced by S.

$$\begin{array}{l} \text{Find } (\lambda, u) \in \mathbb{R} \times S \text{ such that} \\ a(u,v) = \lambda (u,v) \quad \forall v \in S. \end{array} \tag{11.57}$$

With the Ritz ansatz [Sch91]

$$u = \sum_{i=1}^n x_i \phi_i, \tag{11.58}$$

equation (11.57) becomes

$$\begin{array}{l} \text{Find } (\lambda, \mathbf{x}) \in \mathbb{R} \times \mathbb{R}^n \text{ such that} \\ \sum\limits_{i=1}^n x_i a(\phi_i, v) = \lambda \sum\limits_{i=1}^n x_i (\phi_i, v), \quad \forall v \in S. \end{array} \tag{11.59}$$

Eq. (11.59) must hold *for all* $v \in S$, in particular for $v = \phi_1, \cdots, \phi_n$. But since the $\phi_i, 1 \le i \le n$, form a basis of S, equation (11.59) is equivalent with

$$\sum_{i=1}^{n} x_i a(\phi_i, \phi_j) = \lambda \sum_{i=1}^{n} x_i(\phi_i, \phi_j), \quad 1 \le j \le n. \tag{11.60}$$

This is a matrix eigenvalue problem of the form

$$A\mathbf{x} = \lambda M \mathbf{x} \tag{11.61}$$

where

$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \quad A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix}, \quad M = \begin{pmatrix} m_{11} & \cdots & m_{1n} \\ \vdots & \ddots & \vdots \\ m_{n1} & \cdots & m_{nn} \end{pmatrix} \tag{11.62}$$

with

$$a_{ij} = a(\phi_i, \phi_j) = \int_\Omega \nabla\phi_i \, \nabla\phi_j \, dx \, dy + \int_{\partial\Omega} \alpha \, \phi_i \, \phi_j \, ds$$

and

$$m_{ij} = (\phi_i, \phi_j) = \int_\Omega \phi_i \, \phi_j \, dx \, dy.$$

The **finite element method (FEM)** ia a *special case* of the Rayleigh–Ritz method. In the FEM the subspace S and in particular the basis $\{\phi_i\}$ is chosen in a particularly clever way. For simplicity we assume that the domain Ω is a simply connected domain with a polygonal boundary, c.f. Fig 11.5. (This means that the boundary is composed of straight line segments entirely.) This domain is now partitioned into triangular subdomains $T_1, \cdots, T_N$, so-called *elements*, such that

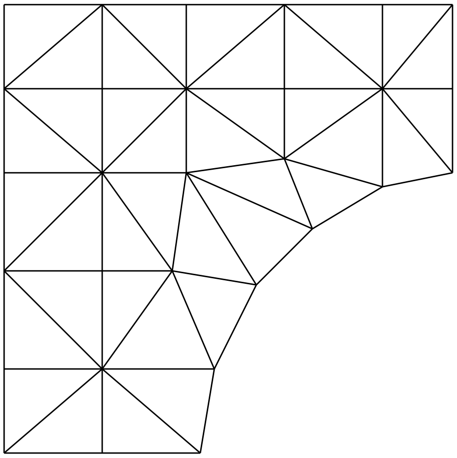


Figure 11.5: Triangulation of a domain Ω

$$\begin{aligned} T_i \cap T_j &= \emptyset, \qquad i \neq j, \\ \bigcup_e \overline{T_e} &= \overline{\Omega}. \end{aligned} \tag{11.63}$$

Finite element spaces for solving (11.39)–(11.41) are typically composed of functions that are *continuous* in Ω and are *polynomials* on the individual subdomains T_e. Such functions are called *piecewise polynomials*. Notice that this construction provides a subspace of the Hilbert space H but not of V, i.e., the functions in the finite element space are not very smooth and the natural boundary conditions are not satisfied.

An essential issue is the selection of the *basis* of the finite element space S. If $S_1 \subset H$ is the space of contiuous, piecewise linear functions (the restriction to T_e is a polynomial of degree 1) then a function in S_1 is uniquely determined by its values at the vertices of the triangles. Let these *nodes*, except those on the boundary portion C_1, be numbered from 1 to n, see Fig. 11.6. Let the coordinates of the i-th node be (x_i, y_i). Then $\phi_i(x, y) \in S_1$ is defined by

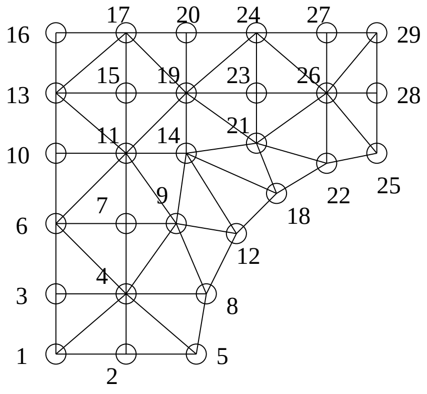


Figure 11.6: Numerierung der Knotenpunkte von Ω (stückweise lineare Polynome)

$$\phi_i((x_j, y_j)) := \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{11.64}$$

A typical basis function ϕ_i is sketched Fig. 11.7 dargestellt.

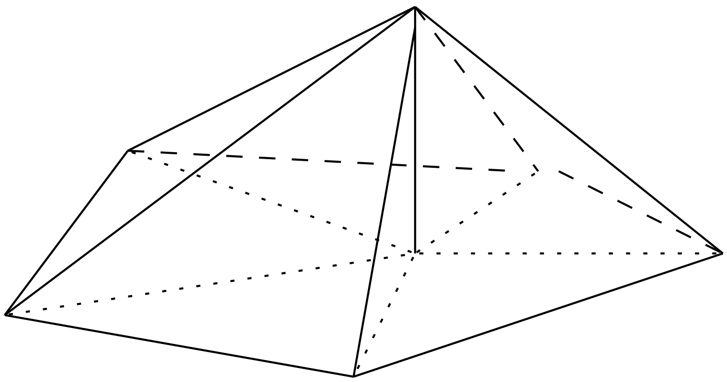


Figure 11.7: A piecewise linear basis function (or hat function)

Another often used finite element element space is $S_2 \subset H$, the space of continuous, piecewise quadratic polynomials. These functions are (or can be) uniquely determined by their values at the vertices and and edge midpoints of the triangle. The basis functions are defined according to (11.64). There are two kinds of basis functions ϕ_i now, first those that are 1 at a vertex and second those that are 1 in an edge midpoint, cf. Fig. 11.8. One immediatly sees that for most $i \neq j$

$$a(\phi_i, \phi_j) = 0, \quad (\phi_i, \phi_j) = 0. \tag{11.65}$$

Therefore the matrices A and M in (11.61) will be **sparse**. The matrix M is positive definite as

$$\mathbf{x}^T M \mathbf{x} = \sum_{i,j=1}^{N} x_i x_j m_{ij} = \sum_{i,j=1}^{N} x_i x_j (\phi_i, \phi_j) = (u, u) > 0, \quad u = \sum_{i=1}^{N} x_i \phi_i \neq 0, \tag{11.66}$$

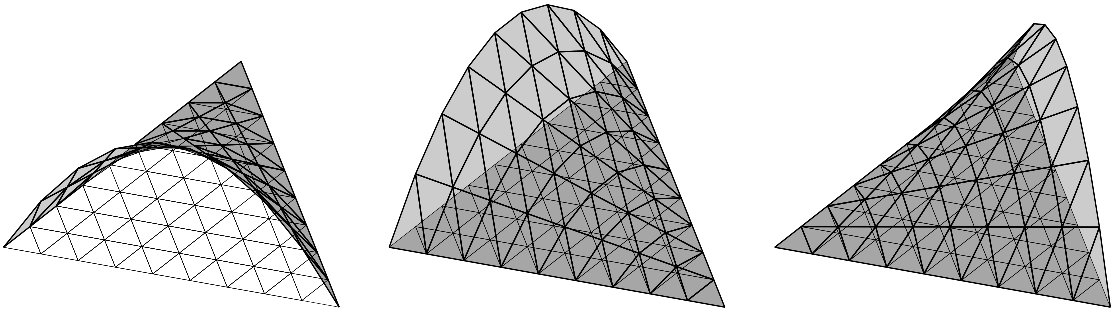


Figure 11.8: A piecewise quadratic basis function corresponding to a edge midpoint [Chi05]

because the ϕ_i are linearly independent and because $u = \sqrt{(u,u)}$ is a norm. Similarly it is shown that

$$\mathbf{x}^T A\mathbf{x} \geq 0.$$

It is possible to have $\mathbf{x^T A x} = 0$ for a nonzero vector $\mathbf{x}$. This is the case if the constant function $u = 1$ is contained in S. This is the case if Neumann boundary conditions $\frac{\partial u}{\partial n} = 0$ are posed on the whole boundary $\partial\Omega$. Then,

$$u(x,y) = 1 = \sum_i \phi_i(x,y),$$

i.e., we have $\mathbf{x^T A x} = 0$ for $\mathbf{x} = [1, 1, \ldots, 1]$.

11.6.3 A numerical example

We want to determine the acoustic eigenfrequencies and corresponding modes in the interior of a car. This is of interest in the manufacturing of cars, since an appropriate shape of the form of the interior can suppress the often unpleasant droning of the motor. The problem is three-dimensional, but by separation of variables the problem can be reduced to two dimensions. If rigid, acoustically hard walls are assumed, the mathematical model of the problem is again the Laplace eigenvalue problem (11.23) together with Neumann boundary conditions. The domain is given in Fig. 11.9 where three finite element triangulations are shown with 87 (grid$_1$), 298 (grid$_2$), and 1095 (grid$_3$) vertices (nodes), respectively. The results obtained with piecewise linear polynomials are listed in Table 11.2. From the results we notice the quadratic convergence rate. The smallest eigenvalue is always zero. The corresponding eigenfunction is the constant function. This function can be represented exactly by the finite element spaces, whence its value is correct (up to rounding error).

Table 11.2 Numerical solutions of acoustic vibration problem

Finite element method			
k	$\lambda_k(\text{grid}_1)$	$\lambda_k(\text{grid}_2)$	$\lambda_k(\text{grid}_3)$
1	0.0000	-0.0000	0.0000
2	0.0133	0.0129	0.0127
3	0.0471	0.0451	0.0444
4	0.0603	0.0576	0.0566
5	0.1229	0.1182	0.1166
6	0.1482	0.1402	0.1376
7	0.1569	0.1462	0.1427
8	0.2162	0.2044	0.2010
9	0.2984	0.2787	0.2726
10	0.3255	0.2998	0.2927

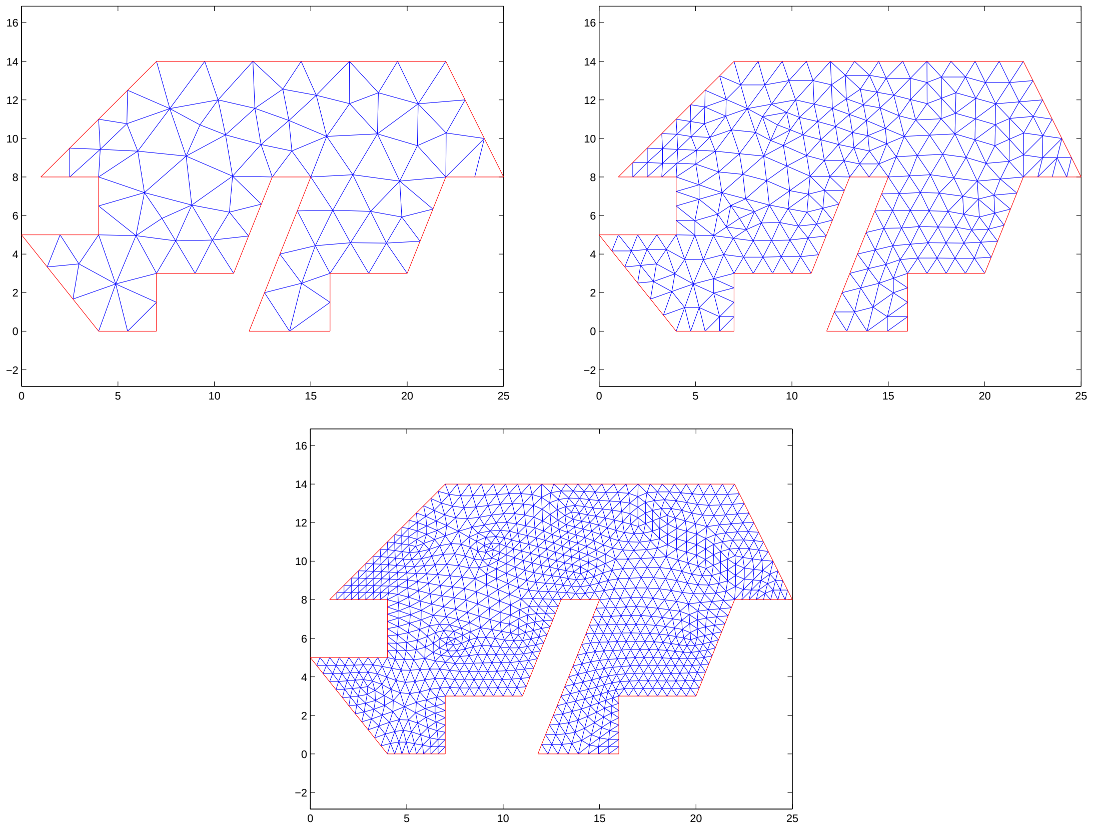


Figure 11.9: Three meshes for the car length cut

The fourth eigenfunction of the acoustic vibration problem is displayed in Fig. 11.10. The physical meaning of the function value is the difference of the presure at a given location to the normal pressure. Large amplitudes thus means that the corresponding noise is very much noticable.

11.7 Cavity resonances in particle accelerators

The Maxwell equations in vacuum are given by

$$\begin{aligned}
\mathbf{curl}\mathbf{E}(\mathbf{x},t) &= -\frac{\partial \mathbf{B}}{\partial t}(\mathbf{x},t), && \text{(Faraday's law)}\\
\mathbf{curl}\mathbf{H}(\mathbf{x},t) &= \frac{\partial \mathbf{D}}{\partial t}(\mathbf{x},t) + \mathbf{j}(\mathbf{x},t), && \text{(Maxwell–Ampère law)}\\
\operatorname{div}\big(\mathbf{D}(\mathbf{x},t)\big) &= \rho(\mathbf{x},t), && \text{(Gauss's law)}\\
\operatorname{div}\big(\mathbf{B}(\mathbf{x},t)\big) &= 0. && \text{(Gauss's law – magnetic)}
\end{aligned}$$

where $\mathbf{E}$ is the electric field intensity, $\mathbf{D}$ is the electric flux density, $\mathbf{H}$ is the magnetic field intensity, $\mathbf{B}$ is the magnetic flux density, $\mathbf{j}$ is the electric current density, and ρ is the electric charge density. Often the "optical" problem is analysed, i.e. the situation when the cavity is not driven (cold mode), hence $\mathbf{j}$ and ρ are assumed to vanish.

Again by separating variables, i.e. assuming a *time harmonic* behavior f the fields, e.g.,

$$\mathbf{E}(\mathbf{x},t) = \mathbf{e}(\mathbf{x})e^{i\omega t}$$

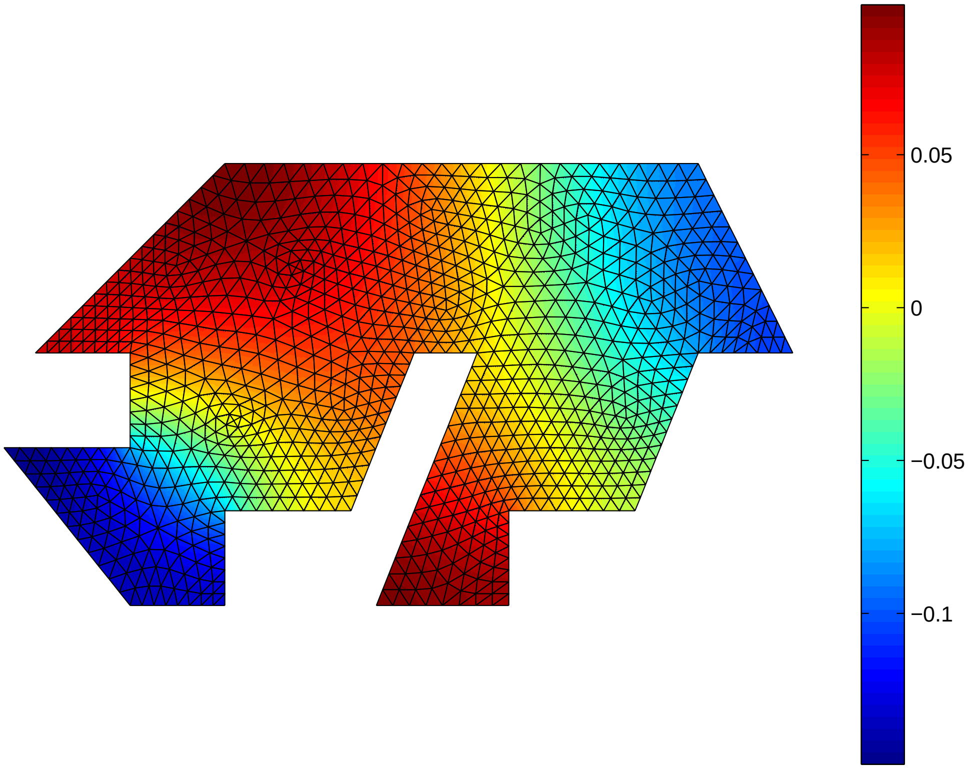


Figure 11.10: Fourth eigenmode of the acoustic vibration problem

using the constitutive relations

$$\mathbf{D} = \epsilon \mathbf{E}, \quad \mathbf{B} = \mu \mathbf{H}, \quad \mathbf{j} = \sigma \mathbf{E},$$

one obtains after elimination of the magnetic field intensity the so called **time-harmonic Maxwell equations**

$$\begin{aligned} \textbf{curl}\mu^{-1}\textbf{curl}\mathbf{e}(\mathbf{x}) &= \lambda\, \epsilon\, \mathbf{e}(\mathbf{x}), && \mathbf{x} \in \Omega, \\ \mathrm{div}\big(\epsilon\, \mathbf{e}(\mathbf{x})\big) &= 0, && \mathbf{x} \in \Omega, \\ \mathbf{n} \times \mathbf{e} &= 0, && \mathbf{x} \in \partial\Omega. \end{aligned} \tag{11.67}$$

Here, additionally, the cavity boundary $\partial\Omega$ is assumed to be *perfectly electrically conducting*, i.e. $\mathbf{E}(\mathbf{x}, t) \times \mathbf{n}(\mathbf{x}) = \mathbf{0}$ for $\mathbf{x} \in \partial\Omega$.

The eigenvalue problem (11.67) is a *constrained eigenvalue problem*. Only functions are taken into account that are divergence-free. This constraint is enforced by Lagrange multipliers. A weak formulation of the problem is then

Find $(\lambda, \mathbf{e}, p) \in \mathbb{R} \times H_0(\mathbf{curl};\ \Omega) \times H_0^1(\Omega)$ *such that* $\mathbf{e} \neq \mathbf{0}$ *and*
(a) $(\mu^{-1}\textbf{curl}\mathbf{e}, \textbf{curl}\boldsymbol{\Psi}) + (\textbf{grad}p, \boldsymbol{\Psi}) = \lambda(\epsilon\, \mathbf{e}, \boldsymbol{\Psi}), \quad \forall \boldsymbol{\Psi} \in H_0(\mathbf{curl};\ \Omega),$
(b) $(\mathbf{e}, \textbf{grad}q) = 0, \quad \forall q \in H_0^1(\Omega).$

With the correct finite element discretization this problem turns in a matrix eigenvalue problem of the form

$$\begin{bmatrix} A & C \\ C^T & O \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} = \lambda \begin{bmatrix} M & O \\ O & O \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix}.$$

The solution of this matrix eigenvalue problem correspond to vibrating electric fields.

Bibliography

[AB84] O. Axelsson and V.A. Barker. *Finite Element Solution of Boundary Value Problems*. Academic Press, Orlando FL, 1984.

[Ame77] W.F. Ames. *Numerical Methods for Partial Differential Equations*. Academic Press, New York NY, 2nd edition, 1977.

[CH68] R. Courant and D. Hilbert. *Methoden der Mathematischen Physik*. Springer, Berlin, 1968.

[Chi05] O. Chinellato. *The Complex-Symmetric Jacobi–Davidson Algorithm and its Application to the Computation of some Resonance Frequencies of Anisotropic Lossy Axisymmetric Cavities*. PhD Thesis No. 16243, ETH Zürich, 2005. (Available at URL `http://e-collection.ethbib.ethz.ch/show?type=diss&nr=16243`).

[Sch91] H. R. Schwarz. *Methode der finiten Elemente*. Teubner, Stuttgart, 3rd edition, 1991.

[Sch93] H. R. Schwarz. *Numerische Mathematik*. Teubner, Stuttgart, 3rd ed. edition, 1993.

[SF73] G. Strang and G. J. Fix. *Analysis of the Finite Element Method*. Prentice-Hall, Englewood Cliffs, 1973.

[Str86] G. Strang. *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, Wellesley, 1986.

[Wei74] H. F. Weinberger. *Variational Methods for Eigenvalue Approximation*. Regional Conference Series in Applied Mathematics 15. SIAM, Philadelphia, PA, 1974.

Chapter 12

Iterative Eigenvalue Solvers

Eigenvalue problems (EVPs) have a long standing history dating back more than a century and a half. Over all these years a lot of inventive methods to tackle such problems have been presented. Many of these ideas improved on already existing methods and customised the latter for special problem classes. Some of these ideas however had a pioneering character in that they reconsidered the problem from a brand new point of view which made the field accessible for new mathematical adventures.

This chapter's goal consists in roughly sketching some of the fundamental techniques used to solve nonsymmetric eigenvalue problems these days. Since a complete discussion of all the developments made on this field would go beyond the scope of these notes, the reader is referred to [vdVG97] and [GvdV00] and references therein, where a more elaborated treatment of the subject is given. Note that in the following we assume that the matrix $\mathbf{A}$ is large, sparse and diagonalisable.

In the following sections we introduce three popular EVP solver classes that have emerged over the years, namely the *vector iteration* method, the *inverse vector iteration* method and the *Arnoldi method*. Each algorithm that we are going to present could be improved on in manifold ways. However, modifications and optimisations often tend to conceal the fundamental ideas, which is the reason, why we content ourselves with sketching the original algorithms. Readers interested in detailed investigations and optimised variants of these algorithms are referred to [GV96], [Par98], [Ste01] and references therein.

12.1 The Vector Iteration Method

This method is especially well suited for the computation of the eigenpair $(\lambda, \mathbf{x})$ associated with one of the largest eigenvalues λ of the EVP

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}. \tag{12.1}$$

Extensions which allow for the computation of several eigenpairs are described in [Par98] and [Ste01] and will not be discussed here.

Let $\mathbf{A} = \mathbf{S}\mathbf{\Lambda}\mathbf{S}^{-1}$ be an eigenvalue factorisation with the eigenvalues λ_i sorted in descending order w.r.t. their modulus, i.e.

$$|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n|. \tag{12.2}$$

Moreover, let $\mathbf{x}$ be a general vector. The d-fold product of $\mathbf{A}$ with the vector $\mathbf{x}$ considered in the space spanned by the columns of $\mathbf{S}$ can be expressed as

$$\mathbf{y} = \mathbf{A}^d\mathbf{x} = \mathbf{S}\mathbf{\Lambda}^d\mathbf{S}^{-1}\mathbf{x} \quad \Leftrightarrow \quad \hat{\mathbf{y}} = \mathbf{\Lambda}^d\hat{\mathbf{x}}. \tag{12.3}$$

Obviously, the single entries of the coordinate vector $\hat{\mathbf{y}}$ grow proportional to the absolute value of the associated eigenvalue for increasing exponents d. For large enough exponent values the component $\hat{y}_1$, being associated with the largest eigenvalue in modulus, will be standing out and $\mathbf{y}$ will increasingly point in the direction of the corresponding eigenvector $\mathbf{s}_1$. Hence, the name *vector iteration method* (or the *power* *method*). Clearly, the direction of $\mathbf{y}$ will be mainly contaminated by the eigenvector associated with the second largest eigenvalue λ_2, wherefore the *convergence rate* can be bound by

$$\eta_{VI}(\mathbf{A}) = \frac{|\lambda_2|}{|\lambda_1|}, \tag{12.4}$$

see [Wil65].

If we now consider a *shifted* eigenvalue problem of the type

$$(\mathbf{A} - \tau\mathbf{I})\mathbf{x} = (\lambda - \tau)\mathbf{x} \tag{12.5}$$

we can carry out the same reasoning as above with the shifted eigenvalues. By applying the bound given in (12.4) we can readily derive

$$\eta_{SVI}(\mathbf{A}, \tau) = \eta_{VI}\big(\mathbf{A} - \tau\mathbf{I}\big) = \frac{\max_{j \neq k}|\lambda_j - \tau|}{\max_k|\lambda_k - \tau|}, \tag{12.6}$$

the convergence rate for the shifted vector iteration method.

In order to implement this scheme, we basically have to compute one matrix-vector product with the shifted operator $\mathbf{A} - \tau\mathbf{I}$ in each step. In order to check for convergence, we then compute the residual

$$\mathbf{r} = (\mathbf{A} - \tau\mathbf{I})\mathbf{x} - \rho\mathbf{x} \tag{12.7}$$

that is associated with the actual approximation $\mathbf{x}$. As can be seen, we need to have an eigenvalue estimator that computes an approximation ρ to a given $\mathbf{x}$. One such estimator is the so called *Rayleigh quotient* which reads

$$\rho(\mathbf{x}) = \frac{\mathbf{x}^{\mathrm{T}}(\mathbf{A} - \tau\mathbf{I})\mathbf{x}}{\mathbf{x}^{\mathrm{T}}\mathbf{x}}. \tag{12.8}$$

Note, that this quantity $\rho(\mathbf{x})$ minimises the residual $\mathbf{r}$ [Ste01]. Finally, in order to prevent the entries from growing unboundedly, we normalise the actual eigenvector approximation in every iteration step, leading to the following algorithm.

Algorithm 12.1.

```
ShiftedVectorIteration($\mathbf{A}, \mathbf{x}, \tau, \epsilon$){
  $\mathbf{x} := \mathbf{x}/\|\mathbf{x}\|$
  $\mathbf{y} := \mathbf{A}\mathbf{x} - \tau\mathbf{x}$
  $\rho := \mathbf{y}^{\mathrm{T}}\mathbf{x}$
  $\mathbf{r} := \mathbf{y} - \rho\mathbf{x}$
  while $(\|\mathbf{r}\| > \epsilon)$ do
    $\mathbf{x} := \mathbf{y}/\|\mathbf{y}\|$
    $\mathbf{y} := \mathbf{A}\mathbf{x} - \tau\mathbf{x}$
    $\rho := \mathbf{y}^{\mathrm{T}}\mathbf{x}$
    $\mathbf{r} := \mathbf{y} - \rho\mathbf{x}$
  end
  return($\rho + \tau, \mathbf{x}$)
}
```

As can be seen from Equation (12.6), the influence of the shift τ onto the behaviour of the shifted vector iteration is only limited. If the desired eigenvalue is not clearly isolated from the rest of the spectrum there is no way of substantially speeding up the algorithm, regardless from the choice of τ.

12.2 The Inverse Vector Iteration Method

It is often the case, that one is given an approximation (target) τ of a desired eigenvalue and would like to compute the exact value λ closest to the given target. In such cases, shifted vector iteration methods are barely usefull, given their disability in recovering interior eigenvalues, i.e. eigenvalues which lie somewhere in the interior of the spectrum. What is usually done in these cases is to recast the eigenvalue problem (12.1) into the *shift-and-inverted* form

$$(\mathbf{A} - \tau\mathbf{I})^{-1}\mathbf{x} = \mu\mathbf{x} \quad \text{where} \quad \mu = \frac{1}{\lambda - \tau}. \tag{12.9}$$

The largest eigenvalues of the matrix on the left hand side are the ones which are closest to τ. Hence, the shifted vector iteration method can again be applied, however, to the matrix $(\mathbf{A} - \tau\mathbf{I})^{-1}$. Hence the name *inverse vector iteration scheme*.

By making use of Equation (12.4), we can derive the following convergence rate

$$\eta_{IVI}(\mathbf{A}, \tau) = \eta_{VI}\big((\mathbf{A} - \tau\mathbf{I})^{-1}\big) = \frac{\min_k |\lambda_k - \tau|}{\min_{j \neq k} |\lambda_j - \tau|}. \tag{12.10}$$

Contrary to the shifted vector iteration method, the inverse vector iteration scheme is extremely responsive to shift values as can be seen when comparing the rates (12.6) and (12.10). In fact, the desired eigenvalue λ need only be reasonably well separated from the remaining ones in order to allow for the choice of a good shift $\tau \approx \lambda$ and hence for a swift convergence.

Unfortunately, one needs to solve a linear system of equation per iteration steps in order to implement this scheme. Besides this, however, the implementation is almost identical to the one of Algorithm 12.1, except for the computation of the residual [Ste01], and leads to

Algorithm 12.2.

```
InverseVectorIteration($\mathbf{A}, \mathbf{x}, \tau, \epsilon$){
  $\mathbf{x} := \mathbf{x}/\|\mathbf{x}\|$
  $\mathbf{y} := (\mathbf{A} - \tau\mathbf{I})^{-1}\mathbf{x}$
  $\rho := \mathbf{y}^{\mathrm{T}}\mathbf{x}$
  $\mathbf{r} := \mathbf{x} - \rho^{-1}\mathbf{y}$
  while ($\|\mathbf{r}\| > \epsilon$) do
    $\mathbf{x} := \mathbf{y}/\|\mathbf{y}\|$
  $\mathbf{y} := (\mathbf{A} - \tau\mathbf{I})^{-1}\mathbf{x}$
  $\rho := \mathbf{y}^{\mathrm{T}}\mathbf{x}$
  $\mathbf{r} := \mathbf{x} - \rho^{-1}\mathbf{y}$
  end
  return($\rho^{-1} + \tau, \mathbf{x}$)
}
```

12.3 The Arnoldi Method

In contrast to the vector iteration methods, where only a single vector is used to scout the space for solutions, *Krylov space methods* construct particular *subspaces* in which solutions are searched. In fact, an examination of the behavior of the sequence of vectors produced by the shifted/inverted vector iteration method suggests that the successive vectors may contain considerable information along eigenvector directions corresponding to eigenvalues other than the one with largest magnitude. The expansion coefficients of the vectors in the sequence evolve in a very structured way. Therefore, linear combinations of the these vectors can be constructed to enhance convergence to additional eigenvectors. A vector iteration scheme based on a single vector simply ignores this additional information. We again assume a target value τ to be given, close to a desired eigenvalue.

Let $\mathbf{M}$ be a given matrix and $\mathbf{u}_0$ be a vector with unit norm. We define the *Krylov space*

$$\mathscr{K}_k(\mathbf{M}, \mathbf{u}_0) = \{\mathbf{u}_0, \mathbf{M}\mathbf{u}_0, \mathbf{M}^2\mathbf{u}_0, \ldots, \mathbf{M}^k\mathbf{u}_0\}. \tag{12.11}$$

One possible way that allows for a simple and efficient handling of linear combinations of eigenvector approximations is the one of performing the *Arnoldi process* presented in Section 9.8.1. In fact, this process induces a procedure which incrementally computes an orthonormal basis $\mathbf{U}_k = (\mathbf{u}_0, \ldots, \mathbf{u}_k)$ for the

Krylov space $\mathscr{K}_k(\mathbf{M}, \mathbf{u}_0)$. After performing k steps of this procedure, the relations

$$\mathbf{M}\mathbf{U}_k = \mathbf{U}_k\mathbf{H}_k + h_{k+1,k}\mathbf{u}_{k+1}\mathbf{e}_k^{\mathrm{T}} \quad \text{and} \quad \mathbf{U}_k^{\mathrm{T}}\mathbf{U}_k = \mathbf{I}_k \tag{12.12}$$

hold, where $\mathbf{H}_k$ is an upper Hessenberg matrix and $\mathbf{e}_k$ is the kth unit vector. Once that such a basis $\mathbf{U}_k$ has been constructed, one can build linear combinations of eigenvector approximations by just performing a matrix-vector product with the basis vectors. Note that in order to increase the subspace by one additional vector, one application of the $\mathbf{M}$ operator is required, among other things.

In analogy to the vector iteration methods, we define the residual $\mathbf{r}(\theta, \mathbf{x})$ of the eigenpair approximation $(\theta, \mathbf{x})$ as

$$\mathbf{r}(\theta, \mathbf{x}) = \mathbf{M}\mathbf{x} - \theta\mathbf{x} = \mathbf{M}\mathbf{U}_k\hat{\mathbf{x}} - \theta\mathbf{U}_k\hat{\mathbf{x}}, \tag{12.13}$$

where this time the eigenvector is constrained to the subspace spanned by $\mathbf{U}_k$, i.e. $\mathbf{x} = \mathbf{U}_k\hat{\mathbf{x}}$. If we require the residual to be orthogonal to $\mathbf{U}_k$, i.e. if we require the eigenvalue problem to be solved exactly in the subspace spanned by $\mathbf{U}_k$, we obtain the so called *Ritz projected eigenvalue problem*

$$\mathbf{U}_k^{\mathrm{T}}\mathbf{M}\mathbf{U}_k\hat{\mathbf{x}} = \theta\mathbf{U}_k^{\mathrm{T}}\mathbf{U}_k\hat{\mathbf{x}}. \tag{12.14}$$

The orthogonality condition stated above is called *Galerkin condition*. By left-multiplying the decomposition given in Equation (12.12) with $\mathbf{U}_k^{\mathrm{T}}$ and by exploiting the orthonormality of $\mathbf{U}_k$, we can simplify problem (12.14) such as to finally be left with the small eigenvalue problem

$$\mathbf{H}_k\hat{\mathbf{x}} = \theta\hat{\mathbf{x}}, \tag{12.15}$$

which can quickly be solved, e.g. by means of the QR algorithms presented in Chapters 3 and 4. We then pick the eigenpair $(\theta_i, \hat{\mathbf{x}}_i)$ closest to τ and have in this way obtained an eigenpair approximation $(\theta_i, \mathbf{U}_k\hat{\mathbf{x}}_i)$ for the eigenproblem

$$\mathbf{M}\mathbf{x}_i = \mathbf{M}\mathbf{U}_k\hat{\mathbf{x}}_i \approx \theta_i\mathbf{U}_k\hat{\mathbf{x}}_i = \theta_i\mathbf{x}_i. \tag{12.16}$$

By alternately applying the Arnoldi process and the *Ritz extraction* to the original EVP we will finally obtain the desired result.

The residual associated with the actual approximation $\mathbf{U}_k\hat{\mathbf{x}}$ can be obtained at almost no extra cost. To show this, we combine Equations (12.12) and (12.13) and obtain

$$\begin{aligned}
\mathbf{r}(\theta, \mathbf{U}_k\hat{\mathbf{x}}) &= \mathbf{M}\mathbf{U}_k\hat{\mathbf{x}} - \theta\mathbf{U}_k\hat{\mathbf{x}} \\
&= (\mathbf{U}_k\mathbf{H}_k + h_{k+1,k}\mathbf{u}_{k+1}\mathbf{e}_k^{\mathrm{T}})\hat{\mathbf{x}} - \theta\mathbf{U}_k\hat{\mathbf{x}} \\
&= (\mathbf{U}_k, \mathbf{u}_{k+1}) \begin{pmatrix} \mathbf{H}_k - \theta\mathbf{I} \\ h_{k+1,k}\mathbf{e}_k^{\mathrm{T}} \end{pmatrix} \hat{\mathbf{x}}.
\end{aligned}$$

Since $(\theta, \hat{\mathbf{x}})$ is an eigenpair of the reduced matrix $\mathbf{H}_k$, we see that

$$\|\mathbf{r}(\theta, \mathbf{U}_k\hat{\mathbf{x}})\| = |h_{k+1,k}\hat{\mathbf{x}}_k|. \tag{12.17}$$

The following algorithm shows a possible implementation of the Arnoldi method, where, in analogy to the (shifted) vector iteration method, we assume the orignal problem to be $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$.

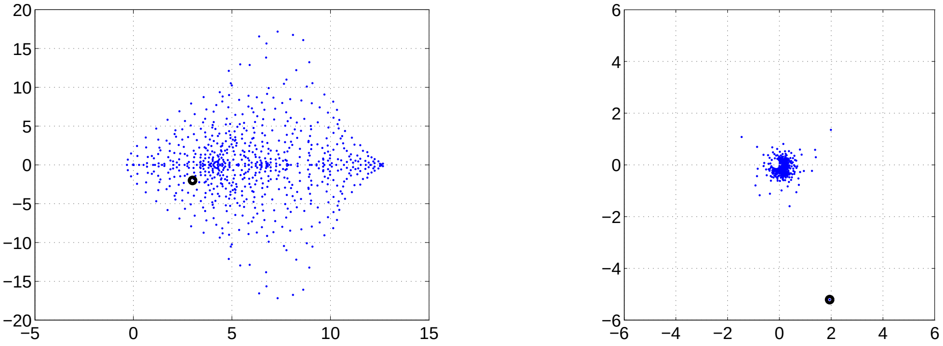


Figure 12.1: Effect of a shift-and-invert transform. *(left)* Spectrum of the test matrix $\mathbf{A}$ (`cavity07.mtx`) obtained from the Matrix-Market repository [BPR$^+$97]. The marker shows the target $\tau = 3 - 2i$. Given that the dsired eigenvalue is in the interior of the spectrum one is expected to perform numerous Arnoldi iteration steps. *(right)* Spectrum of the shift-and-inverted matrix $(\mathbf{A} - \tau\mathbf{I})^{-1}$. Note the clearly isolated eigenvalue (with the largest modulus) in the lower right part of the spectrum.

Algorithm 12.3.

```
ArnoldiMethod($\mathbf{M}, \mathbf{u}, \tau, \epsilon$){
  $\mathbf{u}_1 := \mathbf{u}/\|\mathbf{u}\|$
  $k := 1$
  $\rho := 2\epsilon$
  while $(\rho > \epsilon)$ do
    $\mathbf{u}_\mathrm{A} := \mathbf{A}\mathbf{u}_k$
    for $i := 1, \ldots, k$ do
      $\mathbf{H}_{i,k} := \mathbf{u}_i^\mathrm{T}\mathbf{u}_\mathrm{A}$
      $\mathbf{u}_\mathrm{A} := \mathbf{u}_\mathrm{A} - \mathbf{H}_{i,k}\mathbf{u}_i$
    end
    $\mathbf{H}_{k+1,k} := \|\mathbf{u}_\mathrm{A}\|$
    $\mathbf{u}_{k+1} := \mathbf{u}_\mathrm{A}/\mathbf{H}_{k+1,k}$

    $[\boldsymbol{\theta}, \hat{\mathbf{X}}] :=$ EVPSolve($\mathbf{H}(1:k, 1:k)$)
    $[\theta, j] :=$ min($|\theta_1 - \tau|, \ldots, |\theta_k - \tau|$)
    $\rho := |\mathbf{H}_{k+1,k}\hat{\mathbf{X}}(k, j)|$
    $k := k + 1$
  end
  return($\theta, \mathbf{U}(:, 1:k-1)\hat{\mathbf{X}}(:, j)$)
}
```

As has been shown in [Par98], the eigenvalues θ_i of the reduced system swiftly converge towards the extremal eigenvalues of the original problem $\mathbf{M}\mathbf{x} = \lambda\mathbf{x}$. This suggests, that the original EVP should be transformed in such a way as to place the desired eigenvalues close to the margin of the spectrum. As we have seen in Section 12.2, this can be accomplished by the transformation

$$\underbrace{(\mathbf{A} - \tau\mathbf{I})^{-1}}_{=\mathbf{M}}\mathbf{x} = \theta\mathbf{x} \quad \text{where} \quad \theta = \frac{1}{\lambda - \tau}, \tag{12.18}$$

see Figure 12.1 for an example.

An implementation of the Arnoldi method which takes care of the necessary details in the shift-and-invert context is given in the following.

Algorithm 12.4.

```
ArnoldiMethodSI($\mathbf{A}, \mathbf{u}, \tau, \epsilon$){
  $\mathbf{u}_1 := \mathbf{u}/\|\mathbf{u}\|$
  $k := 1$
  $\rho := 2\epsilon$
  while $(\rho > \epsilon)$ do
    $\mathbf{u}_\mathrm{A} := (\mathbf{A} - \tau)^{-1}\mathbf{u}_k$
    for $i := 1, \ldots, k$ do
      $\mathbf{H}_{i,k} := \mathbf{u}_i^\mathrm{T}\mathbf{u}_\mathrm{A}$
      $\mathbf{u}_\mathrm{A} := \mathbf{u}_\mathrm{A} - \mathbf{H}_{i,k}\mathbf{u}_i$
    end
    $\mathbf{H}_{k+1,k} := \|\mathbf{u}_\mathrm{A}\|$
    $\mathbf{u}_{k+1} := \mathbf{u}_\mathrm{A}/\mathbf{H}_{k+1,k}$

    $[\boldsymbol{\theta}, \hat{\mathbf{X}}] :=$ EVPSolve($\mathbf{H}(1:k, 1:k)$)
    $[\theta, j] :=$ max($\theta_1, \ldots, \theta_k$)
    $\rho := |\mathbf{H}_{k+1,k}\hat{\mathbf{X}}(k, j)/\theta|$
    $k := k + 1$
  end
  return($\theta^{-1} + \tau, \mathbf{U}(:, 1:k-1)\hat{\mathbf{X}}(:, j)$)
}
```

Remark 12.1. *Note that in the shift-and-invert case, the norm of the residual associated with the original problem*

$$\|\mathbf{r}(\lambda, \mathbf{U}_k\hat{\mathbf{x}})\| = \mathbf{A}\mathbf{U}_k\hat{\mathbf{x}} - \lambda\mathbf{U}_k\hat{\mathbf{x}} = \mathbf{A}\mathbf{U}_k\hat{\mathbf{x}} - \big(\frac{1}{\theta} + \tau\big)\mathbf{U}_k\hat{\mathbf{x}} \tag{12.19}$$

typically deviates from the one computed in (12.17)*. We refer the reader to [Ste01] for a more detailed discussion and content ourselves with the use of a modified residual norm (see computation of ρ in the Algorithm below).*

A closer look at the algorithm reveals that, as in the case of the inverse vector iteration method, a linear system needs to be solved in each iteration step. Besides being a costly operation, it requires the availability of a good preconditioner, in order not to waste too much computation time on the construction of the next Arnoldi vector. Moreover care has to be taken when solving the system, due to the possibly poor conditioning induced by the choice of τ.

In addition to that, the storage requirements grow linearly with the number of iteration steps k. To prevent an exceedingly high memory consumption it is therefore common to periodically perform so called *restarts* whenever a certain fixed number of iteration steps has been carried out. By computing the actual approximation and using it to start the Arnoldi method over again, one can limit the storage that is used by the algorithm — a procedure refered to as *explicit restart*. More involved methods, such as the *implicit restart* procedure, can be found in [Ste01] and references therein. Note, however, that since a restart implies the elimination of some of the directions contained in the search space, the follow-up approximations tend to be worse, at least in the very beginning of each restart period, see Figure 12.2.

In order to add an explicit restart mechanism in Algorithm 12.4 it suffices to add the following code fraction to the main loop.

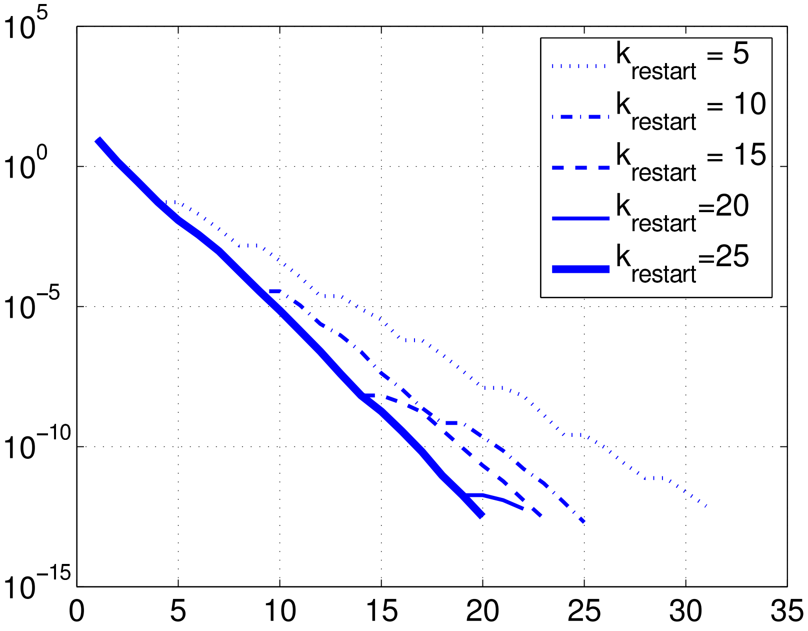


Figure 12.2: Influence of the restart parameter. By varying the restart threshold $k_{restart}$ one can limit the amount of storage that is required at the price of a potentially slower convergence. Here we show 5 residual norm histories for the problem described in Figure 12.1 with the restart sizes given in the legend.

```
[...]
$\rho := |\mathbf{H}_{k+1,k}\hat{\mathbf{X}}(k,j)/\theta|$
$k := k + 1$

if $(k = k_{restart})$ then
   $\mathbf{u}_1 := \mathbf{U}(:, 1 : k - 1)\hat{\mathbf{X}}(:, j)$
   $k := 1$
end
[...]
```

12.4 Available Software

The (shift-and-inverted) Arnoldi method presented in the preceding section is the method of choice in nowadays spectral decomposition software packages, which is mainly due to its relatively simple implementation and its robustness. Among the (freely) available packages, ARPACK [LSY98] is certainly the most popular one. In the following, we will provide a brief of the problems ARPACK is able to solve and of how this is accomplished.

Every problem that is solved in ARPACK, be it a standard EVP or a generalised EVP, is internally considered as problem of the form

$$\mathbf{C}\mathbf{x} = \mu\mathbf{x}, \tag{12.20}$$

where the definition of μ and $\mathbf{C}$ depends on the original problem class. The following table summarises the definitions and transformations that are used in the according cases (τ is a user-specified target).

Origin. EVP	Transf. EVP	$\mathbf{MC} \overset{?}{=} \mathbf{C}^{\mathrm{T}}\mathbf{M}$	Driver	Mode
$\mathbf{Ax} = \lambda\mathbf{x}$	$\mathbf{C} = \mathbf{A}$ $\mu = \lambda$ $\mathbf{M} = \mathbf{I}$	yes no	dsaupd dnaupd	1 1
$\mathbf{Ax} = \lambda\mathbf{Bx}$	$\mathbf{C} = \mathbf{B}^{-1}\mathbf{A}$ $\mu = \lambda$ $\mathbf{M} = \mathbf{B}$	yes no	dsaupd dnaupd	2 2
$\mathbf{Ax} = \lambda\mathbf{Bx}$	$\mathbf{C} = \mathsf{Re}\left[(\mathbf{A} - \tau\mathbf{B})^{-1}\mathbf{B}\right]$ $\mu = \frac{2\lambda-(\tau+\bar{\tau})}{2(\lambda-\tau)(\lambda-\bar{\tau})}$ $\mathbf{M} = \mathbf{B}$	yes no	dsaupd dnaupd	3 3
$\mathbf{Ax} = \lambda\mathbf{Bx}$	$\mathbf{C} = \mathsf{Im}\left[(\mathbf{A} - \tau\mathbf{B})^{-1}\mathbf{B}\right]$ $\mu = \frac{\tau-\bar{\tau}}{2i(\lambda-\tau)(\lambda-\bar{\tau})}$ $\mathbf{M} = \mathbf{B}$	yes no	dsaupd dnaupd	4 4

Remark 12.2. *The driver `dsaupd` is the symmetric counterpart to the Arnoldi Method, i.e. the Lanczos method, and is thus to be preferred in symmetric cases due to its lower storage reuqirements.*

Remark 12.3. *As a matter of fact, there exist several submodes which are slight variations of the ones specified above and differ mainly in the representations of the operators $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$. For the sake of brevity, however, we refer the reader to the documentation of ARPACK [LSY98] for a (detailed) mathematical description, whereas a more software-oriented description can be found in the source files `dsaupd.f` and `dnaupd.f` contained in the package.*

In order to guarantee a certain flexibility and reusability of the software package, ARPACK assumes (and requires) that the user provides the means to evaluate the operators $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$, whenever they need to be applied. This allows for a proper decoupling of the EVP solution process from the EVP transformation process. The mechanism that is used to accomplish this separation is called *reverse communication*.

This mechanism is based on the oservation that the aforementioned operators are used as atomic operations in only a few places during of the Arnoldi method(s), as can be seen from Algorithms 12.3 and 12.4. Therefore it seems reasonable, to interrupt the respective Arnoldi algorithm whenever an operator $\mathbf{A}$, $\mathbf{B}$ or $\mathbf{C}$ needs to be evaluated and to transfer control to the user. The latter, after having performed the necessary computations needed to evaluate the relevant operation, can then restitute control to the ARPACK library which will continue by carrying out the follow-up computations until the next control transfer operation is required.

For the sake of illustration we will conclude this chapter by having a closer look at a concrete example, where we use the ARPACK library to solve the standard (non-symmetric) EVP $\mathbf{Ax} = \lambda\mathbf{x}$. We thereby assume that the system matrix is available and that we can apply it as operator to a given vector. In this example we do not use shift-and-invert.

```
// Fixed ARPACK parameters
int    NEV      = 1;       // Number of desired eigenpairs
int    NCV      = 20;      // Maximal number of Arnoldi vects.
char*  BMAT     = "I";     // M = B = I
char*  WHICH    = "SM";    // Search lambda with |lambda|=min
char   TOL      = 1e-6;    // Residual norm
int    EVECALSO = 0;       // Find only lambdas

// The state indicators
int  IDO;                  // Actual state of ARPACK
int  IPARAM[11];           // Additional parameters (see below)
int  INFO;                 // Information (ok or error)
```

```
// Arnoldi vectors
double*  V;

// Additional working space
double*  RESID;             // Residual vector
double   LAMBDAr[NEV+1];    // Eigenvalue (real part)
double   LAMBDAi[NEV+1];    // Eigenvalue (imaginary part)
int      IPTR[14];          // Indices to work space
double*  WORKD;             // Work space (pointed to by IPTR)
double*  WORKL;             // Work space (pointed to by IPTR)
double*  WORKEV;            // Work space (pointed to by IPTR)
int      SELECT[NCV];       // Selects desired Ritzpairs
int      LWORKL = 3*NCV*NCV + 6*NCV;


// Read the system matrix ...
CSCMatrix A;
mmRead("...", &A);

// Allocate space for the working space and the residual
RESID  = (double*)malloc(A.m*sizeof(double));
V      = (double*)malloc(A.m*NCV*sizeof(double));
WORKD  = (double*)malloc(3*A.m*sizeof(double));
WORKL  = (double*)malloc(LWORKL*sizeof(double));
WORKEV = (double*)malloc(3*NCV*sizeof(double));
```

The code shown so far is responsible for the allocation of the storage required to carry out the Arnoldi process and to compute the eigenvalues of the Hesseneberg matrix. In the next phase, some of these quantities need to be initialised.

```
// Initialise ARPACK by setting special values
IDO  = 0;             // Initialise the library
INFO = 0;             // Use a random initial vector
IPARAM[0]  = 1;       // Shift choice
IPARAM[1]  = 0;       // ---
IPARAM[2]  = A.m;     // Maximum number of iterations
IPARAM[3]  = 1;       // Blocksize
IPARAM[4]  = 0;       // ---
IPARAM[5]  = 0;       // ---
IPARAM[6]  = 1;       // Mode: See manual
IPARAM[7]  = 0;       // ---
IPARAM[8]  = 0;       // ---
IPARAM[9]  = 0;       // ---
IPARAM[10] = 0;       // ---

// Peform the main loop (IDO=99 means (ARPACK) termination)
while(IDO != 99){

  // Call driver with next task IDO
  dnaupd_(&IDO, BMAT, &A.n, WHICH, &NEV, &TOL, RESID,
          &NCV, V, &A.m, IPARAM, IPTR, WORKD,
          WORKL, &LWORKL, &INFO, 1, 2);
  // REVERSE COMMUNICATION: Check which kind of
  // operation needs to be provided
  switch (IDO){
  case -1,1:
    MatVec(A, &WORKD[IPTR[0]-1], &WORKD[IPTR[1]-1]);
    break;

  case 2:
    break;
  case 3:
    break;
  }
}
```

After termination the variable `INFO` contains information on why the Arnoldi loop has been halted. Usually, the required eigenpairs have converged and hence the computation can be stopped. However, possible errors that were encountered are also signaled through this variable. Once the eigenpairs have been found, a postprocessing has to be performed to extract all of the desired information. To this end one can use the routines `dneupd` (for general systems) or `dseupd` (for symmetric systems), which will retrieve the eigenvalues and the eigen-/Ritz-/Schur-vectors, whichever is preferred.

Bibliography

[BPR$^+$97] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard F. Barrett, and Jack J. Dongarra. Matrix market: a web resource for test matrix collections. In *Proceedings of the IFIP TC2/WG2.5 working conference on Quality of numerical software*, pages 125–137, London, UK, UK, 1997. Chapman & Hall, Ltd.

[GV96] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, 1996.

[GvdV00] G. H. Golub and H. A. van der Vorst. Eigenvalue computation in the 20th century. *Journal of Computational and Applied Mathematics*, 123:35–65, 2000.

[LSY98] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems by Implicitely Restarted Arnoldi Methods*. SIAM, Philadelphia, PA, 1998. (The software and this manual are available at URL `http://www.caam.rice.edu/software/ARPACK/`).

[Par98] B. N. Parlett. *The Symmetric Eigenvalue Problem*. Classics in Applied Mathematics. SIAM, 2nd edition, 1998.

[Ste01] G. W. Stewart. *Matrix Algorithms. Volume II: Eigensystems*. SIAM, 2001.

[vdVG97] H. A. van der Vorst and G. H. Golub. 150 years old and still alive: eigenproblems. *The State of the Art in Numerical Analysis*, pages 93–119, 1997. Clarendon Press, Oxford.

[Wil65] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Monographs on Numerical Analysis. Oxford Science Publications, 1965.

Part III

Parallel Sparse Linear Algebra

Chapter 13

Basic Aspects of Parallel Computing

13.1 Taxonomies for Parallel Computers

The definition of an efficient parallel algorithm requires a model of the architecture one is actually using. This model should represent the main features that are supposed to influence the performances of the algorithm: mainly, the memory organisation and the ratio communication/computation.

A first taxonomy divides computers into three categories: *single instruction single data stream* (SISD), *single instruction multiple data stream* (SIMD), *multiple instruction multiple data stream* (MIMD). The SISD model takes a single sequence of instructions and operates on a single sequence of data. The speed of SISD is limited by the execution of a single sequence of instruction and the speed at which information is exchanged between memory and CPU. Information exchange can be increased by the implementation of cache memory and the execution rate can be improved by the execution pipelining techniques.

A SISD architecture is a sequential computer, while SIMD and MIMD architectures both belong to the parallel computer category. A SIMD architecture system has a single control unit furnishing instruction to each processing element in the system. The SIMD architecture is capable of applying exactly the same instruction stream to multiple streams of data simultaneously. For certain classes of problems this architecture is perfectly suited to achieve very high processing rates. This requires data to be split into many different independent pieces, so that multiple instruction units can operate on them at the same time. SIMD architectures proved to be too inflexible; consequenly, they are now used only for very specific applications.

A MIMD architecture system has each control unit in each processing element in such a way that each processor is capable of executing a different program independent of the others in the system. A MIMD architecture is capable of running in true “multiple-instruction” mode, with every processor doing something different, or every processor can be given the same code. The latter case is sometimes called Single Program Multiple Data Stream (SPMD). SPMD architecture is a generalisation of SIMD with much less strict synchronisation requirements.

A second taxonomy is based on *granularity*, i.e., the ratio of the time required for a basic communication operation to the time required for a basic computation. Parallel computers with small granularity are suitable for algorithms requiring frequent communication and the ones with large granularity are suitable for algorithms that do not require frequent communication. The granularity of a parallel computer may also be defined by the number or processors and the speed of each individual processor in the system. Computers with large number of less powerful processors will have small granularity and are called fine-grain computers or Massively Parallel Processors (MPP). In contrast computers with small number of very powerful processors have large granularity and are called coarse-grain computers or multicomputers.

In the middle of the 80’s MPP computers were predicted to be the next big advance in high performance computing. Many companies were formed to build MPP systems, like nCUBE and Thinking Machine. In addition, several existing companies like INTEL, IBM, and Cray Research decided to build these systems. However, nowadays only few centres host and currently use MPP systems. Industry is more oriented to Symmetric Multi Processors (SMP) systems. An SMP system is a coarse-grain parallel computer. To reduce price, they are usually composed of commodity processors. In addition to SMP systems, the so-

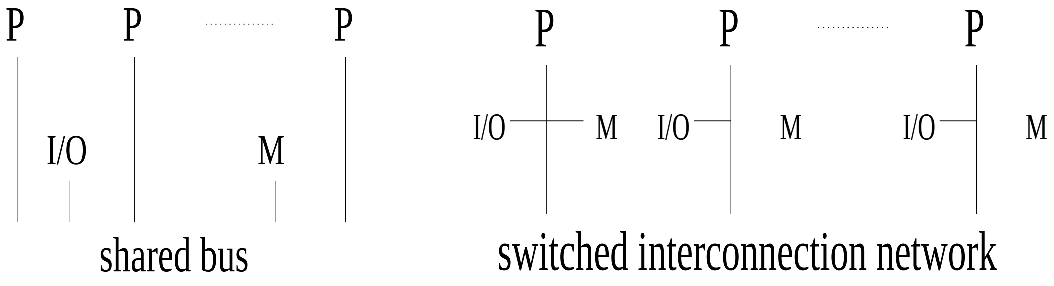


Figure 13.1: Shared memory systems (on the left) and distributed memory systems (on the right). **P** represents the CPU unit, **M** the memory, **I/O** a generic input/output systems.

called Commercial Off-the-Shelf (COTS) systems are increasingly adopted by many institutions, even if these systems are not really a single parallel computer but a collection of systems. Both SMP and COTS systems belong to the class of MIMD systems, with a moderate number of high-performance (and relative low price) processors.

More recently, research is focusing on NOW (Network Of Workstation) architectures, and its generalisation, COMPS (Cluster Of Multi-Processor Systems). Among the NOW systems, an example is represented by the Beowulf system. Is consists of a cluster of PCs or workstations dedicated to running high-performance computing tasks. Instead, a COMPS is a network of multiprocessor computers, and it differs from NOW in that it links multiprocessor workstations. The last frontier is now represented by *grid computing.* A *grid* is a collection of distributed computing resources available over a local or wide area network that appear to an end user or application as one large virtual computing system. For high-performance computing, the present feeling is that NOW, COMPS and grid computing are still not competitive with SMP systems.

A third taxonomy is based on memory organisation, and distinguishes parallel computers between *shared memory* systems and *distributed memory* systems. Shared memory systems were extensively used in the beginning of the 90's. A parallel code running on a shared memory system uses more processors (even if generally few), all sharing the same main memory. The CRAY X-MP, Y-MP, C-90, the IBM 3090-600, ALLIANT FX/80, the NEC SX/4 and SX/5, and the CONVEX C3800 belong to this class.

Using shared memory systems, the communication between processors is implicit and transparent. Processors access memory through the shared bus; see the picture on the left of figure 13.1. Processors do not explicitly communicate with each other so communication protocols are hidden within the system. This means communication can be close to the hardware (or built into the processor), with the shared bus system deciding on how most efficiently to manage communication. Therefore, a serial code can run on these systems with no modification (although maybe performances will be poor).

On the contrary, in distributed memory systems the processors must explicitly communicate with each other through messages. See the picture on the right of figure 13.1 for a graphical representation. The resulting parallel computer has a "simple" architecture overall, since it is composed by local components, connected by a network. The construction of the network (and the definition of a software library to access it) becomes the key to success in the definition of the parallel computer.

Because of cost reasons, industry has now turned the attention to distributed memory systems. The Connection Machine, the INTEL iPSC860 and Paragon, the IBM SP/2, the SGI Origin series, and the ASCI project supercomputers are example of this class of parallel machines. Here every processor has its own local memory and data is communicated among the processors via communication network. Due to the explicit interface to communication, distributed memory systems are more scalable in the sense that hundreds or even thousands of identical processors can work on one problem, without the need of an (expensive) global memory. Moreover, the distributed memory eliminates the cache coherency problems that are typical of the shared memory systems. The price to pay is a more aggressive redesign of the algorithm the code is actually running.

We conclude this section with few references on the subject. More details about the classification of parallel computers and a recent overview of the history of high-performance computing may be found, for instance, in [WDH$^+$99, SDMS99]. The book by Hockney and Reship [HJ88] is a valid reference for vector and parallel computers till the end of the 1980's, while the book by Ortega [Ort88] is a good mix between implementation details and mathematical aspects of vector and parallel computers.

13.2 Cost of Communication Among Processors

We now discuss communication in more details, following [Ort88]. The typical communication problem is to send n words from the local memory of one processor, say P_1, to the memory of another processor, say P_2. On a distributed memory system this communication will be accomplished by a combination of hardware and software. First, the data are loaded into a buffer memory or collected into contiguous locations in memory. Then, a *send* command will be executed on P_1, while P_2 will execute a *receive* command and the data will be routed to their final destination in the memory of P_2. Although many technical details have been omitted in the previous description, the main points of communications are clear:

1. data are collected from the memory of the sending processor;
2. data are physically transfered over the network;
3. the received information are put in the correct memory location of the receiving processor.

Usually, the time t_n required for these 3 phases to send n words is approximately given by

$$t_n = s + \alpha\, n, \tag{13.1}$$

where s is the *startup time* and α is the *incremental time* necessary for each of the n words to be sent. In nowadays implementations, s is generally several order of magnitude greater than α.

This simple – although effective – model can furnish a guideline in the development of parallel algorithms: it is better to have an algorithm which groups communication, that is, sent many data few times, instead of one which requires many send/receive with a small amount of carried data. The number of processors that a given processor has to communicate with, is also to be minimised, since it contributes to additional communication time. This is because each time a processor has to communicate with a processor, a latency penalty for starting a new message is incurred.

Using equation (13.1), the parallel performances of simple algorithms can be studied to optimise the algorithms themselves. Instead, the parallel performances of complex algorithms are usually analysed looking to the global behaviour of a code implementing the algorithm, as the number of processors varies. This global evaluation considers the following parameters:

- *elapsed time* T_p, that is the CPU-time needed to carry out the algorithm using p processors;
- *speedup* s_p, defined as

  $$s_p = \frac{\text{execution time using a single processor}}{\text{execution time using } p \text{ processors}} = \frac{T_1}{T_p}.$$

  One can also define the speedup of a parallel algorithm over the best serial algorithm:

  $$s'_p = \frac{\text{execution time using a single processor for the fast serial algorithm}}{\text{execution time using } p \text{ processors}} = \frac{T'_1}{T_p};$$

- *relative speedup* $s_{p,q}$, defined as the ratio

  $$s_{p,q} = \frac{T_q \times q}{T_p} \tag{13.2}$$

where $T_q \approx T_1/q$ is a reference time using q processors. In fact, often s_p turns out to be hard to compute accurately. Problems that can run on a large number of processors may not fit the memory of a single processor, so that the time for one processor would have to include I/O time to secondary memory – typically, the one of another processor. In this case, q represents the minimum number of processors which can run the defined problem without requiring secondary memory.

- *efficiency* e_p, defined as
$$e_p = T_1/(pT_p) = s_p/p.$$
It is also possible to define the efficiency of an algorithm with respect to the best serial algorithm as $e'_p = s'_p/p$.

One goal in the development of a parallel algorithm is to achieve as large a speedup as possible. For perfectly parallel algorithms we would expect an *ideal* speedup $s_p = s'_p = p$ (that is, using p processors we can reduce the elapsed time by a factor p). In practise, this value cannot be achieved unless the algorithm does not require communications, or the ratio between communication and computations is extremely low (the so-called *embarassing parallelism*). In general, an efficient parallel algorithm will present speedup values close enough to the ideal ones.

Remark 13.1. *Sometimes* superlinear *speedups may be observed, that is, $s_p > p$ or $s'_p > p$. This is due to non-linear effects, like, for instance, the cache reuse: as the dimension of the problem to be solved diminishes, the code can use in a more efficient way the values in the high-performance cache memory, therefore resulting in a consistently lower CPU time.*

The degradation of the speedup value is mainly due to the following factors:

1. lack of perfect degree of parallelism in the algorithm and/or of load balance. Load balance means the assignment of tasks to processors of the system so as to keep each processor doing useful work as much as possible. Load balance may be done either *statically* or *dynamically*. In static load balance tasks (and data) are assigned to processors at the beginning of computation. In dynamic load balance tasks (and data) are assigned to processors as computation proceeds. The implementation of a dynamic load balance algorithm may be difficult on distributed memory systems since data transfer between local memories is in general required as a part of a task assignment;

2. communication, contention, and synchronisation time. We have already discussed about communication. Data contention mainly arises in shared memory computers. Synchronisation is necessary when certain parts of computation must be completed before the overall computation can proceed. There are two aspects of synchronisation that contribute to the overhead. The first is time required to do the synchronisation. The second aspect is that some processors may become idle. Communication, contention, and synchronisation time are usually referred to as *data ready time*.

All these factors may be grouped in a model for the speedup, which reads:

$$s_p = \frac{T_1}{(\alpha_1 + \alpha_2/k + \alpha/p)T_1 + t_d}.$$

Here T_1 is the time for a single processor, α_1 is the fraction of operations done with one processor, α_2 the fraction of operations done with k processors ($k < p$), α if the fraction of operations done with degree of parallelism p, and t_d is data ready time. Clearly, $\alpha + \alpha_1 + \alpha_2 = 1$.

A special (simple) case can be obtained considering $\alpha_2 = 0$ (that is, all the operations have the maximum degree of parallelism) and with no delays ($t_d = 0$). In this case, we obtain

$$s_p = \frac{T_1}{(1 - \alpha + \frac{\alpha}{p})T_1} = \frac{1}{1 - \alpha + \frac{\alpha}{p}} \tag{13.3}$$

and

$$e_p = \frac{\frac{1}{p}}{\frac{\alpha}{p} + (1 - \alpha)} = \frac{1}{\alpha + (1 - \alpha)p}. \tag{13.4}$$

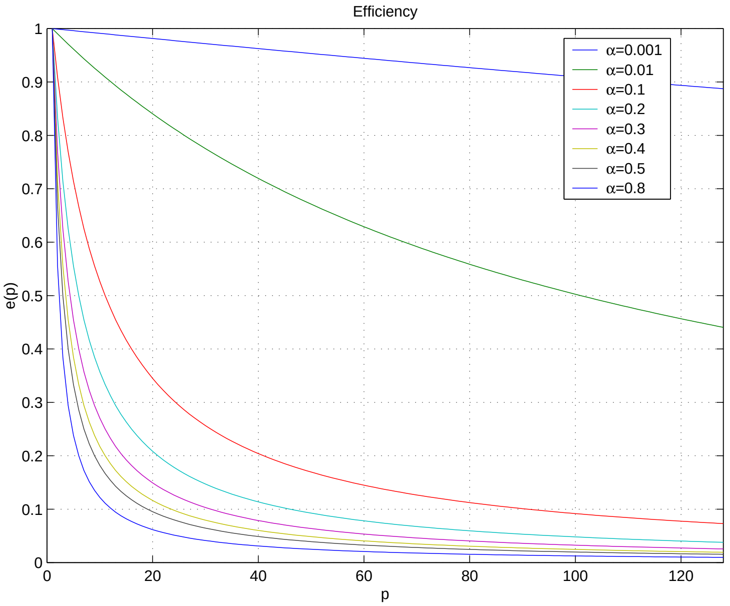


Figure 13.2: Efficiency e_p for various values of α.

Expression (13.3) is known as the *Amdahl's law* or *Ware's law*. Although very simple, (13.3) is instructive. Consider, for example, $\alpha = 1/2$. Then,

$$s_p = \frac{2}{1+p} < 2,$$

that is, no matter how many processors there are, and ignoring all communication, delays, and synchronisation problems, the speedup is always less than 2. Hence, a "good" value of α is of fundamental importance to obtain interesting parallel properties. Figure 13.2 reports the behaviour of e_p with respect to p for several values of α.

13.3 Load Balancing

From the discussion of the previous section, it is clear that both load balance and data ready time are strongly affected by the way data is distributed among the processor. In a DD approach, this means that an algorithm to partition the computational domain is required, to have optimal load balance and as little communication between processors as possible. Ideally, one would like each processor to do exactly the same amount of work. That way, each processor is always working and not staying idle. The way to do this is to first determine what the basis computational task for the algorithm is. For a typical solver, the two main tasks are the matrix-vector product and the preconditioning step. Therefore, it makes sense to partition the grid such that each processor gets a (nearly) equal number of grid vertices. Moreover, the amount of communication among processors should be minimised, as well as the number of neighbouring processors.

During the last years, much work has been done is the area of unstructured grid partitioning. Here we briefly review three algorithms of increasing complexity.

The **Recursive Coordinate Bisection** (RCB) is the simplest and fastest partitioning strategy of the three presented here. It is based upon the ordering of the elements according to the spatial coordinates of their centroids.

Here is the principle. For a given grid, determine along which direction the bisection will operate (for example, the longest direction), say, the $x-$direction. Now, sort the elements according to the $x-$direction, and choose the median value of the $x-$components so that equal number of elements lies in each partition.

Although RCB is very simple, the partition it can produce may be disconnected, and this is not a desirable property since it will result in large communication times. Clearly, the poor parallel properties of RCB are caused by the fact that the algorithm ignores all the grid connectivity. The other two algorithms that we are about to present, instead, use the grid connectivity to obtain high-quality partitions.

The basic ideas of the **Recursive Graph Bisection** (RGB) algorithm are as follows. The two vertices that are furthest apart graphically within the grid are chosen as the starting points for the method[1]. Actually, this is a difficult problem in graph theory, by there are fast ways to find two vertices that are approximately the furthest apart.

Now, start at one of these vertices; call it root. Then, find all the first-order vertices to this root vertex. This set of vertices forms what is called the first level set. The next level set if found by determining all of the neighbours of the first level set that are not already part of a level set. This process continues until half of vertices are members of a level set. It turns out that RGB guarantees that one of the two subdomains produced will be connected. Note that the RGB algorithm is also called the Cuthill-McKee algorithm, and it is often used to reduce the bandwidth of sparse matrices.

The objective of **Recursive Spectral Bisection** (RSB) is to divide the graph into two parts having equal number of vertices such that the number of edges cut is approximately minimised. The key feature is its clever use of eigenstructure of what is called the Laplacian matrix of the graph.

Given a graph G with $|G| = n$ (where $|G|$ represents the number of nodes of G), the Laplacian of G is the matrix $L \in \mathbb{R}^{n\times n}$ defined as

$$L = -D + H,$$

where H is the adjacency matrix,

$$H_{i,j} = \begin{cases} 1 & \text{if edge}(v_i, v_j) \text{ belongs to } G \\ 0 & \text{otherwise,} \end{cases}$$

and D is a diagonal matrix with entries equal to the degree of each vertex, that is the number of its first-order neighbours. Clearly, the rows of L sum to zero, and therefore there is at least one zero eigenvalue. Disconnected graph may have multiple non-zero eigenvalues; however, for this discussion we assume that the graph is connected.

RSB is defined as in the following algorithm.

1. Compute the Laplacian matrix L of the graph.
2. Determine its smallest (in magnitude) non-zero eigenvalue and the corresponding eigenvector. This is called the Fiedler vector.
3. Determine the median value of the entries in the Fiedler vector.
4. Those vertices whose Fiedler vector entry is greater than zero form one subgraph. The remaining vertices form the other subgraph.

It is interesting to see why and how the Fiedler vector plays in this procedure. The argument given below follows closely that given in [Bar94].

Define a partitioning vector $\mathbf{p}$ that assigns each vertex of the graph either a $+1$ or a -1, with $p_i = +1$ if vertex i belongs to the first partition. An equal partition of the graph, which will be assumed to have an even number of vertices, means that

$$\mathbf{s} \perp \mathbf{p},$$

where $\mathbf{s}$ is a vector of 1's.

The key observation is to notice that the number of cut edges E_c is precisely related to the L_1-norm of the Laplacian matrix multiplied by the partitioning vector,

$$4E_c = \|L\mathbf{p}\|_1.$$

[1] The graphical distance between two vertices is defined as the minimum number of edges one has to traverse to get from one vertex to the other

Since the goal is to minimise the number of cut edges, one should find the vector $\mathbf{p} = \hat{\mathbf{p}}$ that minimises the norm $\|L\hat{\mathbf{p}}\|_1$, where $\|\hat{\mathbf{p}}\|_1 = n$, and $\mathbf{s} \perp \hat{\mathbf{p}}$. Because L is real symmetric negative semidefinite, it has a complete set of eigenvectors that can be orthogonalised with each others. Therefore, one can write $\hat{\mathbf{p}}$ as

$$\hat{\mathbf{p}} = \sum_{i=1}^{n} \alpha_i \mathbf{v}_i,$$

$\mathbf{v}_i$ being the eigenvectors of L. It is possible to show that $\|L\hat{\mathbf{p}}\|_1$ is minimised by choosing $\hat{\mathbf{p}} = n\mathbf{v}_2/\|\mathbf{v}_2\|_1$, and therefore the Fiedler vector defines a partition of the graph which minimises the number of cut edges. This is done using a Lanczos algorithm.

The numerical cost of RSB is higher than that of RCB or RGB because of the computation of the Fielder vector. However, RSB can be applied to partition general unstructured grids, and results in partition of good quality.

13.4 Parallel Programming Tools

Parallel programming is more challenging than serial programming for various reasons. First, parallel programs must include mechanisms for data exchange. Second, in an efficient parallel program the work must be evenly divided among processors. This is an algorithmic problem that has no counterpart in the serial programming. And finally, the data structures must be divided among the processors to preserve data locality. This is obviously true for distributed memory systems in which data movement is costly. It is also true for shared memory systems since data locality reduces the cost of maintaining cache coherence, hence resulting in a faster implementation of the algorithm.

To help the programmer in developing parallel codes, several different parallel methodologies have been pursued. In the class of MIMD systems, there are different aspects for developing a parallel program, one more suited for shared memory and the other for distributed memory systems. In shared memory programming, programmers view their programs as a collection of processes accessing a central pool of shared variables. In the message passing programming, programmers view their programs as a collection of processes with private local variables and the ability to send and receive data between processes by passing messages. On one side, we have the *implicit parallelisation*, automatically performed by the machine (and the compiler); on the other side we find the *explicit parallelisation*, that requires the users to take control of all the parallel aspects.

For shared memory computers, one can exploit the parallel programming using threads. The parallel code is composed by many independent processors, thus manipulating data stocked in a common area. Each thread is given to a different processor. A more recent approach for the programming of SMP is the standard OpenMP. It consists in a set of procedures and directives that are added to the (sequential) source code to aid the compiler to build up the parallel code. In this case, the aim is to produce parallel code starting from the sequential one. Some compiler directives are added to help the compiler in detecting the parallel cycles and the optimal distribution of data among the processors. An example of this approach is High Performance FORTRAN (HPF). This approach is very practical for the programmer, and it can work well for problems with a regular structure. Instead, it is in general quite difficult to apply for unstructured data and for a large number of processors. This approach is somehow less common than others, and in general the automatic parallelisation is more effective on shared memory than on distributed memory architectures. Moreover, the best results are obtained for codes lying on cycles whose parallelism does not rely on the input data.

For distributed memory computer, instead, the interaction among the processor is made up using messages. The parallel program is organised as a collection of processes running on a certain number of processors, each of them owning its private local memory. Data exchange is performed only by means of explicit send and receive operations.

Message passing is often preferred to other approached because, although explicit message passing may be more complex to code with respect to other parallelisation techniques, it has a key advantage over other approaches: it focalizes the programmer's attention on the performance-critical issue of the parallel hardware – *data locality*. The programmer is forced to exploit the data structure of the problem, and to use (or develop) numerical methods which minimise the use of non-local data in all the various phases of the code.

The two important standards for message passing are PVM and MPI [For95].

PVM was mainly built around the notion of ‘virtual machine’ – a set of heterogeneous hosts connected by a network that appears logically to the user as a single large parallel computer. The exchange of data, or in other words, communication, is accomplished using simple message-passing constructs. The concept of portability was considered much more important than using the natural message passing constructs of the underlying hardware. This in fact has paved the way for the present large scale use of PVM.

MPI, in contrast to PVM, was designed not with portability as the main aim, but with the view of creating a standard message passing interface for high-performance parallel computing. The focus of MPI developers’ team was more on performance, which they tried to improve with the natural message-constructs of the unrelying hardware. One of MPI’s prime goals is to produce a system that would allow manufacturers of high performance massively parallel processing computers to provide highly optimised and efficient implementations, while PVM was designed primarily for network of workstations with the goal of portability gained at the sacrifice of optimal performance. MPI has advantages in the areas of point to point communication, collective communication, ability to specify communication topologies and ability to create derived data types. There is also a study going on for developing a system PVMPI [FD96], which uses the already proved and widely ported MPI message-passing system within PVM to enable interoperability with different implementations execution on heterogeneous distributed hardware and more recent projects to built interfaces portable to both MPI and PVM [AKK99].

MPI is the *de-facto* standard approach to program distributed memory machines. Because of its low-level constructs, it allows programmers to precisely control how the parallel computation proceeds. Also, it allows programmers to write portable parallel programs that run well on shared-memory machines, massively supercomputers, clusters. A wide variety of operations are available: there are more than 125 functions in MPI 1.1, which is the version used by the majority of MPI programmers. This version of the library was released in 1995. Since then, the standard has evolved. The specification on an enhanced version, MPI 2.0, with parallel I/O (MPI-IO), dynamic process management, one-sided communication, and other advanced features was released in 1997. Unfortunately, it was such a complex addition to the standard that only few implementations support MPI 2.0.

Installing MPI on a Unix/Linux machine is quite simple. There are several implementations of MPI in common used. The two most common are LAM/MPI [BDV94] and MPICH [GL96]. Both may be downloaded free of charge and are quite straightforward to install, with support for a wide range of parallel computers. The following section shows how to compile and run few simple MPI programs.

13.4.1 A Simple MPI Program

The simplest MPI program is the parallel extension of the famous `Hello, World!` C program of Kernighan and Ritchie [KR78]. The C++ program reported below will output a sentence from each processor involved in the computation.

```
#include <iostream>
#include "mpi.h"

int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);

  int NumProcs; // number of processors in the group
  int MyPID; // a unique identifier ranging from 0 to (NumProcs - 1)

  MPI_Comm_size(MPI_COMM_WORLD, &NumProcs);
  MPI_Comm_rank(MPI_COMM_WORLD, &MyPID);

  std::cout << "Hello World from processor " << MyPID;
  std::cout << " of " << NumProcs << std::endl;
  MPI_Finalize();
  return(EXIT_SUCCESS);
}
```

The structure of this simple example is quite typical of all MPI programs. First, we include the MPI header file `mpi.h`. The MPI library is initialized by MPI_Init(), and finalized by MPI_Finalize(). No MPI instructions can be executed before `MPI_Init()` or after `MPI_Finalize()`^{2}. Among others, the initialization function creates the global communicator, `MPI_COMM_WORLD`. In most cases, MPI programmers only need a single communicator and just the default one; however new communicator can be created if required, to isolate few processors or to create different communication channels. Functions `MPI_Comm_rank()` returns the process ID of the calling processor, while `MPI_Comm_size()` returns the total number of processors in the computations.

To compile this program, you might simply need

```
$ mpic++ hello_world.cpp -o hello_world
```

You still need to run it. The MPI standard does not mandate how a job is started or executed, so there is considerable variation between different MPI implementations. Typically, `mpirun` is used to launch MPI programs; some architectures use `prun` (DEC) or `yod` (CRAY). The following instructions might suffice:

```
$ mpirun -np 2 ./hello_world
Hello World from processor 0 of 2
Hello World from processor 1 of 2
```

13.4.2 Basic Point-to-Point Message Passing

Once our first MPI program has been fired, let's explore MPI a little bit more. Let us suppose that we want to send the value of a `double` variable from processor 0 to processor 1. A possible solution is the following:

```
#include <iostream>
#include "mpi.h"

int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);

  int NumProcs; // number of processors in the group
  int MyPID; // a unique identifier ranging from 0 to (NumProcs - 1)

  MPI_Comm_size(MPI_COMM_WORLD, &NumProcs);
  MPI_Comm_rank(MPI_COMM_WORLD, &MyPID);

  double a = 0.0;
  MPI_Status status;
  if (MyPID == 0)
  {
    a = 123.4;
    int SendTo = 1;
    int Tag = 0;
    MPI_Send(&a, 1, MPI_DOUBLE, SendTo, Tag, MPI_COMM_WORLD);
  }
  else
  {
    int RecvFrom = 0;
    int Tag = 0;
    MPI_Recv(&a, 1, MPI_DOUBLE, RecvFrom, Tag, MPI_COMM_WORLD, &status);
```

[2] Function MPI_Finalized() can be used at any time to query the initialization state of the MPI library.

```
    std::cout << "Value of a on processor 1 = " << a << std:endl;
  }

  MPI_Finalize();
  return(EXIT_SUCCESS);
}
```

What we have just shown is the most commonly used message-passing functions in MPI: the blocking send/receive functions `MPI_Send()` and `MPI_Recv()`. The former returns when the buffer has been transmitted into the system and can safely be reused; the latter returns when the buffer has received the message and is ready to use. Using sends and receives can be more challenging than it may appear at a first sight. Since `MPI_Send()` is blocking, for large message sizes it will return only when the message has been sent—that is, when the corresponding `MPI_Recv()` returns. One should be sure to pair the send and receive phase correctly, or use non-blocking functions.

There are more than 21 functions in MPI 1.1 for point-to-point communication. This large set of message-passing functions provides the controls needed to optimize the use of communication buffers and specify how communication and computation overlap. An overview of the communication modes is as follows:

- **Standard Mode** (`MPI_Send()`). The standard MPI send, the send will not complete until the send buffer is empty and ready to reuse.

- **Synchronous mode** (`MPI_Ssend()`). The send does not complete until after a matching receive has been posted. This makes it possible to use the communication as a pairwise synchronization event.

- **Buffered Mode** (`MPI_Bsend()`). User-supplied buffer space is used to buffer the messages. The send will complete as soon as the send buffer is copied to the system buffer.

- **Read mode** (`MPI_Rsend()`). The send will transmit the message immediately under the assumption that a matching receive has already been posted. On some systems, ready mode communication is more efficient.

13.4.3 Collective Operations

In addition to the point-to-point message-passing functions, MPI includes a set of operations which all the processes in the group work together to carry out. The most commonly used collective operations include the following:

- `MPI_Barrier()`. A barrier defines a synchronization point at which all processes arrive before any of them are allows to proceed. For MPI, this means that every process using the indicated communicator must call the barrier function before any of them proceed.

- `MPI_Bcast()`. A broadcast sends a message from one processor to all the processes in a group.

- `MPI_Reduce()`. A reduction operation takes a set of values spread out around a process group and combines them using the indicated binary operation. To be meaningful, the operation in question must be associative. The most common examples for the binary function are summation and finding the maximum or a minimum of a set of values. The result is available only in the indicated destination process. If the value is needed by all processes, there is a variant called `MPI_Allreduce()`.

A simple example is reported below. First, we synchronize all the processors, then we perform a reduction operation and we store the result on processor 0, then we broadcast a value from processor 0 to all the other processors.

```
#include <iostream>
#include "mpi.h"
```

```
int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);

  int MyPID; // a unique identifier ranging from 0 to (NumProcs - 1)

  MPI_Comm_rank(MPI_COMM_WORLD, &MyPID);

  // Synchronize all the processors
  MPI_Barrier(MPI_COMM_WORLD);

  // sets some local values. LocalValue has a different
  // value on each processor; GlobalValue is not set
  int LocalValue = 10 * MyPID, GlobalValue;

  // Finds the mininum (i.e., 0), store the result on processor 0
  // Here "1" is the length of the array to be processed, which
  // is of type MPI_INT (i.e., int)
  MPI_Reduce(&LocalValue, &GlobalValue, 1, MPI_INT, MPI_MIN,
             0, MPI_COMM_WORLD);

  // reset LocalValue on processor 0
  if (MyPID == 0) LocalValue = 123;

  // Broadcast this value from process 0 to all processes
  MPI_Bcast(&GlobalValue, 1, MPI_INT, 0, MPI_COMM_WORLD);

  MPI_Finalize();
  exit(EXIT_SUCCESS);
}
```

This concludes our short overview of the MPI library. More details can be found, for example, ine [Pac96, GHLL+98].

Bibliography

[AKK99] V. Annamalai, C.S. Krishnamoorthy, and V. Kamakoti. Adaptive finite element analysis on a parallel and distributed environment. *Parallel Computing*, 25:1413–1434, 1999.

[Bar94] T.J. Barth. Aspects of unstructured grids and finite-volume solvers for the Euler and Navier-Stokes equations. In *VKI LS 1994-05, Computational Fluid Dynamics*, 1994.

[BDV94] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[FD96] G. Fagg and J. Dongarra. PVMPI: An integration of PVM and MPI systems. *Calculateurs Parallèles*, 8(2):151–166, 1996.

[For95] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical report, 1995.

[GHLL$^+$98] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI - The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT Press, Cambridge, MA, USA, 1998.

[GL96] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[HJ88] R.W. Hockney and C.R. Jesshope. *Parallel Computers 2*. Adam Hilger Ltd, Bristol, 1988.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Ort88] J. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems*. Plenum Press, New York, 1988.

[Pac96] Peter S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[SDMS99] E. Strohmaier, J.J. Dongarra, H.W. Meuer, and H.D. Simon. The marketplace of high-performance computing. *Parallel Computing*, 25:1517–1544, 1999.

[WDH$^+$99] D.E. Womble, S.S. Dosanjh, B. Hendrickson, M.A. Heroux, S.J. Plimpton, J.L. Tomkins, and D.S. Greenberg. Massively parallel computing: a Sandia perspective. *Parallel Computing*, 25:1853–1876, 1999.

Chapter 14

Distributed Linear Algebra with Trilinos

14.1 Installing Trilinos

To obtain Trilinos, please follow the instructions at the web site

```
http://software.sandia.gov/Trilinos
```

Trilinos has been compiled on a variety of architectures, including various flavors of Linux, Sun Solaris, SGI Irix, DEC, Mac OS X, ASCI Red, and many others. Trilinos has been designed to support parallel applications. However, it also compiles and runs on serial computers. Detailed comments on the installation, and an exhaustive list of FAQs, may be found at the web pages:

```
http://software.sandia.gov/Trilinos/installing_manual.html
http://software.sandia.gov/Trilinos/faq.html
```

After obtaining Trilinos, the next step is its compilation. The description here is for LINUX platforms with MPI, the compilation on other platforms being pretty similar. The configuration and compilation steps follow the classical tt configure; make; make install procedure. For more details, simply type

```
$ <your-trilinos-directory>/configure --help
```

All Trilinos packages can be build to run with or without MPI. If you want to configure Trilinos without MPI support, just type

```
$ <your-trilinos-directory>/configure
$ make
$ make install
```

If you want to enable to support for MPI, you may need `--enable-mpi` and `--with-mpi-compilers`. Extensive support is reported on the Trilinos' web page.

The Trilinos framework uses a two level software structure that connects a system of *packages*. A Trilinos package is an integral unit, usually developed to solve a specific task, by a (relatively) small group of experts. Packages exist beneath the Trilinos top level, which provides a common look-and-feel. Each package has its own structure, documentation and set of examples, and it is possibly available independently of Trilinos. However, each package is even more valuable when combined with other Trilinos packages. In the following, we suppose that Trilinos has been configured with support for TEUCHOS, EPETRA, AZTECOO, IFPACK and ML. All these packages are enabled by defauls, unless you configure with the option `--disable-default-packages`. TEUCHOS is a collection of utilities; EPETRA provides distributed linear algebra objects, and it is covered in Section 14.2; AZTECOO defines a variety of Krylov solvers and preconditioners, as described in Section 14.3; IFPACK and ML provides preconditioners. A simple usage of ML is decribed in Section 14.4.

14.2 Building Distributed Vectors and Sparse Matrices using Epetra

EPETRA is the core of Trilinos. Creating distributed vectors and sparse matrices with EPETRA is quite simple. The library also offers some capabilities to handle serial dense objects, an interface to BLAS and LAPACK.

Our overview of EPETRA is as follows:

1. We first create an communicator object in Section 14.2.1. If Trilinos has been configured with MPI support, then this object encapsulates MPI. Otherwise, it simply mimics MPI functionalities on a single-processor machine. By using this communicator wrapper, the same code can be compiled and executed with serial and MPI support, with no changes.
2. We specify the data layout of our vectors and matrices using maps, in Section 14.2.2.
3. We create distributed vectors in Section 14.2.3.
4. We assemble a distributed sparse matrix in Section 14.2.4.
5. We define a linear system, and we solve it using AZTECOO, in Section 14.3.
6. Finally, we present state-of-the-art preconditioners for elliptic equations in Section 14.4.

14.2.1 Encapsulating MPI

EPETRA encapsulates the all intra-processor communications in the Epetra_Comm virtual class. An Epetra_Comm object is required for building all Epetra_Map objects, which in turn are required for all other Epetra distributed objects. Epetra_Comm has two basic concrete implementations: Epetra_SerialComm for serial executions, and Epetra_MpiComm for MPI distributed memory executions. By encapsulating the communicator, the calls to MPI_Init() and MPI_Finalize() are likely to be the *only* MPI calls you have to explicitly introduce in your code.

For most basic applications, the user can create an Epetra_Comm object using the following code:

```
#include "Epetra_ConfigDefs.h"
#ifdef HAVE_MPI
#include "mpi.h"
#include "Epetra_MpiComm.h"
#else
#include "Epetra_SerialComm.h"
#endif

int main( int argv, char *argv[])
{
#ifdef HAVE_MPI
  MPI_Init(&argc, &argv);
  Epetra_MpiComm Comm(MPI_COMM_WORLD);
#else
  Epetra_SerialComm Comm;
#endif
  cout << "Hello World from processor " << Comm.MyPID();
  cout << " of " << Comm.NumProc() << endl;

  MPI_Finalize();
  return(EXIT_SUCCESS);
}
```

Most of Epetra_Comm methods are similar to MPI functions. The class provides methods such as `MyPID()`, `NumProc()`, `Barrier()`, `Broadcast()`, `SumAll()`, `GatherAll()`, `MaxAll()`, `MinAll()`, `ScanSum()`.

Note that the macro `HAVE_CONFIG_H` must be defined either in the user's code or as a compiler flag in all examples that use Trilinos.

14.2.2 Data Layout using Maps

The distribution of a set of integer labels (or elements) across the processes is here called a *map*, and its actual implementation is given by the Epetra_Map class (or, more precisely, by an Epetra_BlockMap, from which Epetra_Map is derived). Basically, the class handles the definition of the:

- global number of elements in the set (called `NumGlobalElements`);
- local number of elements (called `NumMyElements`);
- global numbering of all local elements (an integer vector of size `NumMyElements`, called `MyGlobalElements`).

There are three ways to define an map. The easiest way is to specify the global number of elements, and let Epetra decide:

```
Epetra_Map Map(NumGlobalElements,0,Comm);
```

In this case, the constructor takes the global dimension of the vector, the base index[1], and an `Epetra_Comm` object (introduced in Section 14.2.1). As a result, each process will be assigned a contiguous set of elements.

A second way to build the Epetra_Comm object is to furnish the local number of elements:

```
Epetra_Map Map(-1,NumMyElements,0,Comm);
```

This will create a vector of size $\sum_{i=0}^{NumProc-1}$ `NumMyElements`. Each process will get a contiguous set of elements.

A third more involved way to create an Epetra_Map, is to specify on each process both the number of local elements, and the global indexing of each local element. To understand this, consider the following code. A vector of global dimension 5 is split among processes `p0` and `p1`. Process `p0` owns elements 0 an 4, and process `p1` elements 1, 2, and 3.

```
#include "Epetra_Map.h"
// ...
MyPID = Comm.MyPID();
switch( MyPID ) {
case 0:
  MyElements = 2;
  MyGlobalElements = new int[MyElements];
  MyGlobalElements[0] = 0;
  MyGlobalElements[1] = 4;
  break;
case 1:
  MyElements = 3;
  MyGlobalElements = new int[MyElements];
  MyGlobalElements[0] = 1;
  MyGlobalElements[1] = 2;
  MyGlobalElements[2] = 3;
  break;
}

Epetra_Map Map(-1,MyElements,MyGlobalElements,0,Comm);
```

[1] The index base is the index of the lowest order element, and is usually, `0` for C or C++ arrays, and `1` for FORTRAN arrays. Epetra can indeed accept any number as index base. However, some other Trilinos package may require a C-style index base.

Once created, a Map object can be queried for the global and local number of elements, using

```
int NumGlobalElements = Map.NumGlobalElements();
int NumMyElements = Map.NumMyElements();
```

and for the global ID of local elements, using

```
int* MyGlobalElements = Map.MyGlobalElements();
```

that returns a pointer to the internally stored global indexing vector, or, equivalently,

```
int MyGlobalElements[NumMyElements];
Map.MyGlobalElements(MyGlobalElements);
```

that copies in the user's provided array the global indexing.

The class Epetra_Map is derived from Epetra_BlockMap. The class keeps information that describes the distribution of objects that have block elements (for example, one or more contiguous entries of a vector). This situation is common in applications like multiple-unknown PDE problems. A variety of constructors are available for the class.

Note that different maps may coexist in the same part of the code. The user may define vectors with different distributions (even for vectors of the same size). Two classes are provided to transfer data from one map to an other: Epetra_Import and Epetra_Export (not described here).

Remark 14.1. *Most Epetra objects overload the* << *operator. For example, to visualize information about the* `Map`*, one can simply write*

```
cout << Map;
```

14.2.3 Distributed Vectors

A distributed object is an entity whose elements are partitioned across more than one process. Epetra's distributed objects (derived from the Epetra_DistObject class) are created from a Map. For example, a distributed vector can be constructed starting from an Epetra_Map (or Epetra_BlockMap) with an instruction of type

```
Epetra_Vector x(Map);
```

(We shall see that this dependency on Map objects holds for all distributed Epetra objects.) This constructor allocates space for the vector and sets all the elements to zero. A copy constructor may be used as well:

```
Epetra_Vector y(x);
```

A variety of sophisticated constructors are indeed available. For instance, the user can pass a pointer to an array of double precision values,

```
Epetra_Vector x(Copy,Map,LocalValues);
```

Note the word `Copy` is input to the constructor. It specifies the Epetra_CopyMode, and refers to many Epetra objects. In fact, Epetra allows two data access modes:

1. `Copy`: allocate memory and copy the user-provided data. In this mode, the user data is not needed be the new Epetra_Vector after construction;

2. `View`: create a "view" of the user's data. The user data is assumed to remain untouched for the life of the vector (or modified carefully). From a data hiding perspective, View mode is very dangerous. But is is often the only way to get the required performance. Therefore, users are strongly encouraged to develop code using the Copy mode. Only use View mode as needed in a secondary performance optimization phase. To use the View mode, the user has to define the vector entries using a (double) vector (of appropriate size), than construct an Epetra_Vector with an instruction of type

```
    Epetra_Vector z(View,Map,z_values);
```

where `z_values` is a pointer a double array containing the values for `z`.

To set a locally owned element of a vector, one can use the `[]` operator, regardless of how a vector has been created. For example,

```
x[i] = 1.0*i;
```

where `i` is in the local index space.

Epetra also defines some functions to set vector elements in local or global index space. `ReplaceMyValues` or `SumIntoMyValues` will replace or sum values into a vector with a given indexed list of values, with indexes in the *local* index space; `ReplaceGlobalValues` or `SumIntoGlobalValues` will replace or sum values into a vector with a given indexed list of values in the *global* index space (but locally owned). It is important to note that no process may set vector entries locally owned by another process. In other words, both global and local insert and replace functions refer to the part of a vector assigned to the calling process.

The user might need (for example, for reasons of computational efficiency) to work on Epetra_Vectors as if they were `double *` pointers. `ExtractCopy` does not give access to the vector elements, but only copies them into the user-provided array. The user must commit those changes to the vector object, using, for instance, `ReplaceMyValues`.

A further computationally efficient way, is to extract a "view" of the (multi-)vector internal data. This can be done as follows, using method `ExtractView()`. Let `z` be an Epetra_Vector.

```
double* z_values;
z.ExtractView(&z_values);
for (int i = 0; i < MyLength ; ++i) z_values[i] *= 10;
```

In this way, modifying the values of `z_values` will affect the internal data of the Epetra_Vector `z`.

Remark 14.2. *The class Epetra_Vector is derived from Epetra_MultiVector. Roughly speaking, a multi-vector is a collection of one or more vectors, all having the same length and distribution.*

The user can also consider the function `ResetView`, which allows a (very) light-weight replacement of multi-vector values, created using the Epetra_DataMode `View`. Note that no checking is performed to see if the values passed in contain valid data. This method can be extremely useful in the situation where a vector is needed for use with an Epetra operator or matrix, and the user is not passing in a multi-vector. Use this method with caution as it could be extremely dangerous.

It is possible to perform a certain number of operations on vector objects. Some of them are reported in Table 14.1.

14.2.4 Distributed Sparse Matrices

Epetra provides an extensive set of classes to create and fill distributed sparse matrices. These classes allow row-by-row or element-by-element constructions. Support is provided for common matrix operations, including scaling, norm, matrix-vector multiplication and matrix-multivector multiplication.

Using Epetra objects, applications do not need to know about the particular storage format, and other implementation details such as data layout, the number and location of ghost nodes. Epetra furnishes two basic formats, one suited for point matrices, the other for block matrices. Here, we will consider the format for point matrices.

As an example, in this Section we will present how to construct a distributed (sparse) matrix, arising from a finite-difference solution of a one-dimensional Laplace problem. This matrix looks like:

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

Table 14.1 Some methods of the class `Epetra_Vector`

`int NumMyELements()`
returns the local vector length on the calling processor

`int NumGlobalElements()`
returns the global length

`int Norm1(double *Result) const`
returns the 1-norm (defined as $\sum_i^n |x_i|$ (see also `Norm2` and `NormInf`)

`Normweigthed(double *Result) const`
returns the 2-norm, defined as $\sqrt{\frac{1}{n}\sum_{j=1}^n (w_j x_j)^2}$)

`int Dot(const Epetra MultiVector A, double *Result) const`
computes the dot product of each corresponding pair of vectors

`int Scale(double ScalarA, const Epetra MultiVector &A`
Replace multi-vector values with scaled values of A, `this=ScalarA*A`

`int MinValue(double *Result) const`
compute minimum value of each vector in multi-vector (see also `MaxValue` and `MeanValue`)

`int PutScalar(double Scalar)`
Initialize all values in a multi-vector with constant value

`int Random()`
set multi-vector values to random numbers

Table 14.2 Mathematical methods of `Epetra_RowMatrix`

`virtual int Multiply (bool TransA, const Epetra_MultiVector &X, Epetra_MultiVector &Y) const=0`
Returns the result of a Epetra_RowMatrix multiplied by a Epetra_MultiVector X in Y.

`virtual int Solve (bool Upper, bool Trans, bool UnitDiagonal, const Epetra_MultiVector &X, Epetra_MultiVector &Y) const=0`
Returns result of a local-only solve using a triangular Epetra_RowMatrix with Epetra_MultiVectors X and Y.

`virtual int InvRowSums (Epetra_Vector &x) const=0`
Computes the sum of absolute values of the rows of the Epetra_RowMatrix, results returned in x.

`virtual int LeftScale (const Epetra_Vector &x)=0`
Scales the Epetra_RowMatrix on the left with a Epetra_Vector x.

`virtual int InvColSums (Epetra_Vector &x) const=0`
Computes the sum of absolute values of the cols of the Epetra_RowMatrix, results returned in x.

`virtual int RightScale (const Epetra_Vector &x)=0`
Scales the Epetra_RowMatrix on the right with a Epetra_Vector x.

Table 14.3 Atribute access methods of `Epetra_RowMatrix`

`virtual bool Filled () const=0`

If FillComplete() has been called, this query returns true, otherwise it returns false.

`virtual double NormInf () const=0`

Returns the infinity norm of the global matrix.

`virtual double NormOne () const=0`

Returns the one norm of the global matrix.

`virtual int NumGlobalNonzeros () const=0`

Returns the number of nonzero entries in the global matrix.

`virtual int NumGlobalRows () const=0`

Returns the number of global matrix rows.

`virtual int NumGlobalCols () const=0`

Returns the number of global matrix columns.

`virtual int NumGlobalDiagonals () const=0`

Returns the number of global nonzero diagonal entries, based on global row/column index comparisons.

`virtual int NumMyNonzeros () const=0`

Returns the number of nonzero entries in the calling processor's portion of the matrix.

`virtual int NumMyRows () const=0`

Returns the number of matrix rows owned by the calling processor.

`virtual int NumMyCols () const=0`

Returns the number of matrix columns owned by the calling processor.

`virtual int NumMyDiagonals () const=0`

Returns the number of local nonzero diagonal entries, based on global row/column index comparisons.

`virtual bool LowerTriangular () const=0`

If matrix is lower triangular in local index space, this query returns true, otherwise it returns false.

`virtual bool UpperTriangular () const=0`

If matrix is upper triangular in local index space, this query returns true, otherwise it returns false.

`virtual const Epetra_Map & RowMatrixRowMap () const=0`

Returns the Epetra_Map object associated with the rows of this matrix.

`virtual const Epetra_Map & RowMatrixColMap () const=0`

Returns the Epetra_Map object associated with the columns of this matrix.

`virtual const Epetra_Import * RowMatrixImporter () const=0`

Returns the Epetra_Import object that contains the import operations for distributed operations.

The example illustrates how to construct the matrix, and how to perform matrix-vector operations.

We start by specifying the global dimension (here is 5, but can be any number):

```
int NumGlobalElements = 5;
```

We create a map (for the sake of simplicity linear), and define the local number of rows and the global numbering for each local row:

```
Epetra_Map Map(NumGlobalElements,0,Comm);
int NumMyElements = Map.NumMyElements();
int* MyGlobalElements = Map.MyGlobalElements( );
```

In particular, we have that `j=MyGlobalElements[i]` is the global numbering for local node `i`. Then, we have to specify the number of nonzeros per row. In general, this can be done in two ways:

- Furnish an integer value, representing the number of nonzero element on each row (the same value for all the rows);
- Furnish an integer vector `NumNz`, of length `NumMyElements()`, containing the nonzero elements of each row.

The first approach is trivial: the matrix is create with the simple instruction

```
Epetra_CrsMatrix A(Copy,Map,3);
```

(The `Copy` keyword is explained in Section 14.2.3.) In this case, Epetra considers the number 3 as a "suggestion," in the sense that the user can still add more than 3 elements per row (at the price of a possible performance decay). The second approach is as follows:

```
int* NumNz = new int[NumMyElements];
for (int i = 0; i < NumMyElements; i++)
if (MyGlobalElements[i] == 0 ||
    MyGlobalElements[i] == NumGlobalElements-1)
  NumNz[i] = 2;
else
  NumNz[i] = 3;
```

We are building a tridiagonal matrix where each row has (-1 2 -1). Here `NumNz[i]` is the number of nonzero terms in the i-th global equation on this process (2 off-diagonal terms, except for the first and last equation).

Now, the command to create an Epetra_CsrMatrix is

```
Epetra_CrsMatrix A(Copy,Map,NumNz);
```

We add rows one at a time. The matrix A has been created in Copy mode, in a way that relies on the specified map. To fill its values, we need some additional variables: let us call them `Indexes` and `Values`. For each row, `Indices` contains global column indices, and `Values` the correspondingly values.

```
double Values[2];
Values[0] = -1.0; Values[1] = -1.0;
int Indices[2];
double two = 2.0;
int NumEntries;

for (int i = 0; i < NumMyElements; ++i)
{
  if (MyGlobalElements[i]==0)
  {
    Indices[0] = 1;
    NumEntries = 1;
  }
  else if (MyGlobalElements[i] == NumGlobalElements - 1)
  {
    Indices[0] = NumGlobalElements - 2;
    NumEntries = 1;
  }
  else
  {
    Indices[0] = MyGlobalElements[i] - 1;
    Indices[1] = MyGlobalElements[i] + 1;
    NumEntries = 2;
  }
  A.InsertGlobalValues(MyGlobalElements[i], NumEntries,
                       Values, Indices);
  // Put in the diagonal entry
  A.InsertGlobalValues(MyGlobalElements[i], 1, &two,
                       MyGlobalElements+i);
}
```

Note that column indices have been inserted using global indices (but a method called `InsertMyValues` can be used as well) . Finally, we transform the matrix representation into one based on local indexes. The transformation in required in order to perform efficient parallel matrix-vector products and other matrix operations.

```
A.FillComplete();
```

This call to `FillComplete()` will reorganize the internally stored data so that each process knows the set of internal, border and external elements for a matrix-vector product of the form $B = AX$. Also, the communication pattern is established. As we have specified just one map, Epetra considers that the the rows of A are distributed among the processes in the same way of the elements of X and B.

14.2.5 Epetra_LinearProblem

A linear system $AX = B$ is defined by an Epetra_LinearProblem class. The class requires an Epetra_RowMatrix or an Epetra_Operator object (often an Epetra_CrsMatrix or Epetra_VbrMatrix), and two (multi-)vectors X and B. X must have been defined using a map equivalent to the DomainMap of A, while B using a map equivalent ot the RangeMap of A (see Section 14.2.4).

14.2.6 Concluding Remarks on Epetra

More details about the Epetra project, and a technical description of classes and methods, can be found in [Her02].

14.3 Solving the Linear Systems using AztecOO

Once an Epetra_LinearProblem has been created with the command

```
Epetra_LinearProblem Problem(&A,&x,&b);
```

where `A` is an Epetra matrix, and both `x` and `b` are Epetra vectors, it can be easily solved using an iterative method of Krylov type with the AztecOO package. At this aim, we first need to create an AztecOO object,

```
AztecOO Solver(Problem);
```

You have to specify how to solve the linear system. All AZTECOO options are set using two vectors, one of integers and the other of doubles, as detailed in the Aztec's User Guide [THHS99]. For example, to use a Jacobi preconditioner, you can type

```
Solver.SetAztecOption(AZ_precond, AZ_Jacobi);
```

To use an ILU with fill-in of 3 and overlap of 2, do instead

```
Solver.SetAztecOption(AZ_precond, AZ_dom_decomp);
Solver.SetAztecOption(AZ_overlap, 2);
Solver.SetAztecOption(AZ_graph_fill, 3);
```

You are now ready to solve the linear system. Instruction

```
Solver.Iterate(1550,1E-9);
```

runs the Krylov solver, using 1550 maximum iterations, and a tolerance of 10^{-9} on the relative residual.

Note that the matrix must be in local form (that is, the command `A.FillComplete()` *must* have been invoked before solving the linear system). The same AZTECOO linear system solution procedure applies in serial and in parallel. However for some preconditioners, the convergence rate (and the number of iterations) depends on the number of processor.

When `Iterate()` returns, one can query for the number of iterations performed by the linear solver using `Solver.NumIters()`, while `Solver.TrueResidual()` gives the (unscaled) norm of the residual.

14.4 Multilevel Preconditioners using ML

We now present how to define a multilevel preconditioner based on smoothed aggregation using the ML package. This family of preconditioners is implemented in the MultiLevelPreconditioner class, defined in the ML_Epetra namespace.

The MultiLevelPreconditioner class automatically constructs all the components of the preconditioner, using the parameters specified in a TEUCHOS parameter list. The constructor of this class takes as input an Epetra_RowMatrix pointer and a TEUCHOS parameter list[2].

We now give a very simple fragment of code that uses the MultiLevelPreconditioner. The linear operator `A` is derived from an Epetra_RowMatrix, `Solver` is an AztecOO object, and `Problem` is an Epetra_LinearProblem object.

```
#include "ml_include.h"
#include "ml_MultiLevelPreconditioner.h"
#include "Teuchos_ParameterList.hpp"

...

Teuchos::ParameterList MList;

// set default values for smoothed aggregation in MLList
ML_Epetra::SetDefaults("SA",MLList);

// overwrite with user's defined parameters
MLList.set("max levels",6);
MLList.set("increasing or decreasing","decreasing");
MLList.set("aggregation: type", "MIS");
MLList.set("coarse: type","Amesos-KLU");
```

[2] In order to use the MultiLevelPreconditioner class, ML must be configured with options `-enable-epetra --enable-teuchos`.

```
// create the preconditioner
ML_Epetra::MultiLevelPreconditioner* MLPrec =
  new ML_Epetra::MultiLevelPreconditioner(A, MLList);

// create an AztecOO solver
AztecOO Solver(Problem)

// set preconditioner and solve
Solver.SetPrecOperator(MLPrec);
Solver.SetAztecOption(AZ_solver, AZ_gmres);
Solver.Iterate(Niters, 1e-12);

...

delete MLPrec;
```

The code snippet first includes few header files: `ml_config.h` (as first ML include), `Epetra_ConfigDefs.h` (as first Epetra include), `Epetra_RowMatrix.h`, `Epetra_MultiVector.h`, `Epetra_LinearProblem.h`, and `AztecOO.h`, and the `ml_MultiLevelPreconditioner.h` file.

All the parameters that affect the preconditioners are contained in a Teuchos parameter list. A *parameter list* is a container of objects; Table 14.4 briefly reports the most important methods of this class.

Table 14.4 Some methods of Teuchos::ParameterList class.

`set(Name,Value)`	Add entry `Name` with value and type specified by `Value`. Any C++ type (like int, double, a pointer, etc.) is valid.
`get(Name,DefValue)`	Get value (whose type is automatically specified by `DefValue`). If not present, return `DefValue`.
`subList(Name)`	Get a reference to sublist `List`. If not present, create the sublist.

The parameter list is passed to the constructor, together with a pointer to the matrix. Then, we have to pass the preconditioner object to AztecOO using the SetPrecOperator() method, then we solve the linear system using Iterate(). The hierarchy is destroyed using `MLPrec->DestroyPreconditioner()`. We suggest to always create the preconditioning object with `new` and to delete it using `delete`. Some MPI calls occur in `DestroyPreconditioner()`, so the user should not call `MPI_Finalize()` or delete the communicator used by ML before the preconditioning object is destroyed.

Bibliography

[Her02] M. A. Heroux. *Epetra Reference Manual*, 2.0 edition, 2002. http://software.sandia.gov/trilinos/packages/epetra/doxygen/latex/EpetraReferenceManual.pdf.

[THHS99] R. Tuminaro, M. Heroux, S. Hutchinson, and J. Shadid. Official Aztec user's guide: Version 2.1. Technical Report Sand99-8801J, Sandia National Laboratories, Albuquerque NM, 87185, Nov 1999.