Iterative Solution of Linear Systems

with Improved Arithmetic and Result Verification

Meinen Eltern,
Inge und Armin

Iterative Solution of Linear Systems

with Improved Arithmetic and Result Verification

Zur Erlangung des akademischen
Grades eines

DOKTORS DER
NATURWISSENSCHAFTEN

von der Fakultät für Mathematik der
Universität Karlsruhe (TH)

genehmigte

DISSERTATION

von

Dipl.-Math. techn. Axel Facius
aus Schwäbisch Gmünd

Contents

List of Algorithms

Introduction

During the last decades, we have had an exponential growth of processor speed and storage capacity. Due to Moore's law, these quantities increase by a factor of two every 18 months. This means, we have about one thousand times the computing power today than we had in 1985, the year when the IEEE floating point standard 754 [4] was released. This standard proposes a 64 bit arithmetic for floating-point operations and up to now there are hardly any improvements innovated by hardware manufactures. The complete computing power is still utilized to increase the size of problems that can be handled.

"There will still be people who say not all the relevant physics is in the models, but that's still a much smaller criticism than not being able to establish firm results for the physics that is being modeled." (John Gustafson, 1998 [54])

One of the largest consumers of floating-point arithmetics are iterative solvers for linear systems of equations. Most of todays big problems in scientific computing, e.g, in structural engineering or fluid dynamics are modeled by differential equations which lead to large linear systems after discretization. Therefore it is an important task to develop accurate and reliable algorithms for this purpose.

Frequently, preconditioned Krylov subspace methods are used to solve these large and often sparse linear systems. Theoretically, Krylov methods have many favorable properties concerning convergence rates, accuracy, computing time, and storage requirements. Unfortunately, these properties prevalently do not hold in the presence of roundoff errors. Computing time increases due to unnecessarily many iterations and expensively computed stopping criteria and more storage is needed, e.g, because of reorthogonalization strategies. Moreover, convergence does sometimes not happen at all or stagnates without delivering the desired accuracy.

There are many investigations which try to quantize the attainable accuracy or convergence rates in finite precision arithmetic. However, the purpose of this work is not to try to get the best result with the given arithmetic. We aim to fix

our requirements on final accuracy and then choose an appropriate arithmetic that enables us to meet these requirements.

After identifying the critical parts which mostly suffer from finite precision arithmetic, we selectively introduce arithmetical improvements to reduce the propagated errors and thereby reducing the number of needed iterations.

The work in this thesis is based on recent developments on *state of the art* linear system solvers, on arithmetical tools for verification and highly accurate computing, as well as on high performance object oriented programming. Starting from there, we develop powerful algorithms which are capable to deliver almost any desired accuracy. Additionally, if the system is not too large, we are often able to prove the correctness of our results, i.e., we can give a rigorous upper bound for the error norm.

The thesis is structured as follows.

Chapter 1 gives an introduction into the theory of preconditioned Krylov subspace methods. We start with a unified description of subspace methods in general and various Krylov subspace generation techniques. After summarizing the most important results of *infinite precision* convergence theory, we give a detailed introduction into generic preconditioners. Particularly, we focus on splitting techniques and incomplete factorizations with a special emphasis on suitable modifications made for the requirements of our verification methods. Based on these fundamentals, we give a broad overview over Krylov subspace solvers and describe some of the most important variants in more details. For each method we prove the important short recurrence properties and give pseudo-programming language formulation of the preconditioned algorithm.

In **Chapter 2**, we investigate the behavior of preconditioned Krylov Methods in the presence of finite precision arithmetic. After introducing the basic concepts of floating-point numbers and arithmetic with these numbers, we present the central results of *finite precision* theory of Krylov methods. Based on the error analysis of C. Paige, we proved the direct dependency of the level of orthogonality among the Krylov basis vectors on the used arithmetic. Subsequently, we give some examples, demonstrating the influence of rounding errors on preconditioning and solving.

To narrow the gap between exact precision behavior as described in Chapter 1 and finite precision behavior stated in Chapter 2, **Chapter 3** introduces several important techniques which increase the precision and reliability of computer arithmetic. Namely, we describe high precision arithmetics, interval arithmetic, and the exact scalar product.

Chapter 4 summarizes important techniques for computing error bounds for solutions of large linear systems. Introducing the basic concepts of classical verification methods, such as interval Gaussian elimination or interval fixed point methods we pass over into recently developed verification techniques based on perturbation theory. In particular, we describe a fundamental method to bound error norms via residual norms and then deduce a more advanced technique improving these bounds by exploiting the *Lanczos-Gauß connection*.

The next two chapters focus on implementation techniques on a computer. **Chapter 5** gives a broad overview of several powerful programming techniques for writing high performance object oriented numerical linear algebra routines.

Almost all ideas and concepts presented in this thesis are implemented in the **v**ariable precision **k**rylov solver **vk**. **Chapter 6** describes the structure of the code and how to use it. Additionally there is a graphical user interface **xvk** which is also described in this chapter.

Finally, **Chapter 7** exemplifies the techniques and methods, described in this thesis. Particularly, we show that improved arithmetics are not only capable to deliver more accurate results but also can accelerate convergence significantly. Moreover we present highly precise verified solutions for systems with up to $2\,000\,000$ unknowns or condition numbers of approximately 10^{62}.

Notation

Throughout this thesis, all matrices are denoted by bold capital letters ($\boldsymbol{A}$), vectors by bold lowercase letters ($\boldsymbol{a}$), and scalar variables by ordinary lowercase letters (a, or α). Interval variables are enclosed in square brackets ($[\boldsymbol{A}]$, $[\boldsymbol{a}]$, $[a]$, or $[\alpha]$).

If not mentioned explicitly, all matrices are square with dimension n and the vectors are n-vectors.

By default the column vectors of a matrix are denoted with the same but lowercase letter and the elements are printed with the same letter, too, but in medium weight. For example we have

$$\boldsymbol{A} = (\boldsymbol{a}_1 | \cdots | \boldsymbol{a}_n) = (a_{i,j})_{i,j=1}^n.$$

This notational convention is also used in the other direction, i.e., when we have a sequence of vectors and need them collected in a matrix. Calligraphic letters ($\mathcal{K}$) denote (affine) vector-spaces or sets.

Some letters have a predefined sense in this thesis. That is the system matrix $\boldsymbol{A}$, the right hand side vector $\boldsymbol{b}$, the exact solution

$$\boldsymbol{x}^* = \boldsymbol{A}^{-1}\boldsymbol{b},$$

any approximation $\tilde{\boldsymbol{x}}$ to the solution, the residual vector

$$\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{x}},$$

the standard basis vectors $\boldsymbol{e}_i = (0, \ldots, 0, 1, 0, \ldots, 0)^T$ with the '1' at the ith place, and the identity matrix $\boldsymbol{I}$. The letter ϵ always denotes the machine precision, but since we deal with different number formats, we also have different values of ϵ. Therefore the actual size is always given in the context if necessary. Eigenvalues are always called λ and singular values σ. Particularly the smallest singular value is denoted with $\sigma_{\min}$.

We tried not to use variable names twice while simultaneously respecting traditionally used notational conventions. There is only one exception, where we preferred convention over uniqueness. That are the scalar variables in various Krylov algorithms (α and β) which conflict with scalar coefficients in the (Bi-)Lanczos algorithm.

Table 1 shows the most important used functions and operators with the according definition.

Symbol	Definition
$\langle \boldsymbol{x} \mid \boldsymbol{y} \rangle$	scalar product of $\boldsymbol{x}$ and $\boldsymbol{y}$
$\boldsymbol{x} \perp \boldsymbol{y}$	$\boldsymbol{x}$ and $\boldsymbol{y}$ are orthogonal, i.e., $\langle \boldsymbol{x} \mid \boldsymbol{y} \rangle = 0$
$\mathrm{span}\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$	linear hull of $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$
$\mathrm{rank}(\boldsymbol{A})$	rank of $\boldsymbol{A}$
$\rho(\boldsymbol{A})$	spectral radius of $\boldsymbol{A}$
$\mathrm{cond}(\boldsymbol{A})$	condition number of $\boldsymbol{A}$
$\triangle(\cdot),\ \triangledown(\cdot),\ \diamond(\cdot),\ \square(\cdot)$	round upwards, downwards, to interval, to nearest
$\boxplus,\ \boxminus,\ \boxtimes,\ \boxslash$	upward rounded arithmetic operations
$\triangledown\!\!\!+,\ \triangledown\!\!\!-,\ \triangledown\!\!\!\times,\ \triangledown\!\!\!/$	downward rounded arithmetic operations
$\oplus,\ \ominus,\ \otimes,\ \oslash$	interval arithmetic operations

Table 1: Functions and operators with the according definition.

1

Preconditioned Krylov Subspace Methods

" Ich empfehle Ihnen diesen Modus zur Nachahmung.
Schwerlich werden Sie je wieder direkt eliminieren,
wenigstens nicht, wenn Sie mehr als zwei Unbekannte haben.
Das indirekte [iterative] Verfahren läßt sich halb im Schlaf ausführen
oder man kann während desselben an andere Dinge denken.[1] *"*

Carl Friedrich Gauß to Christoph Ludwig Gerling,
December 26, 1823

Krylov subspace methods are used both to solve systems of linear equations $\boldsymbol{Ax} = \boldsymbol{b}$ and to find eigenvalues of $\boldsymbol{A}$ [21, 22]. In this work we focus on linear system solving, however particularly in investigating theoretical properties of Krylov methods, we also need some facts from eigenvalue theory [96].

Krylov algorithms assume that $\boldsymbol{A}$ is accessible only via a *black-box* subroutine that returns $\boldsymbol{y} = \boldsymbol{Az}$ for any $\boldsymbol{z}$ (and perhaps $\boldsymbol{y} = \boldsymbol{A}^T\boldsymbol{z}$ if $\boldsymbol{A}$ is nonsymmetric). This is an important assumption for several reasons. First, the cheapest non-trivial

---

[1] "I recommend this method for your imitation. You will hardly ever again eliminate directly, at least not when you have more than two unknowns. The indirect [iterative] procedure can be done while half asleep, or while thinking about other things."

operation that one can perform on a sparse matrix is to multiply it by a vector — if $\boldsymbol{A}$ has nnz nonzero entries, a matrix-vector multiplication costs nnz multiplications and (at most) nnz additions. Secondly, $\boldsymbol{A}$ may not be represented explicitly as a matrix but may be available only as a subroutine for computing $\boldsymbol{Az}$.

This chapter is organized as follows. In Section 1.1 we give some basic facts about subspace solvers in general. Section 1.2 describes how information about $\boldsymbol{A}$ is extracted via matrix-vector multiplication. In Krylov subspace methods, this information is stored in the so called Krylov subspaces. In Section 1.3 we present some convergence theory for symmetric and nonsymmetric Lanczos-like algorithms. Since for real life problems, one will hardly ever solve a linear system of equations without preconditioning, we describe the basic facts of preconditioning and introduce the most important generic preconditioners in Section 1.4. Based on these introductory sections, Section 1.5 gives an overview about several Krylov solvers and describes some of the most important variants in more detail.

1.1   Subspace Methods

The basic idea of subspace methods is generating a sequence of subspaces $\mathcal{V}_m$ with increasing dimension m and finding a vector $\boldsymbol{x}_m$ within each of these subspaces that is in some sense an *optimal* approximation in $\mathcal{V}_m$ to the solution $\boldsymbol{x}^*$ of the entire problem. Clearly, this optimality measure has to guarantee that we choose $\boldsymbol{x}_m = \boldsymbol{x}^*$ if $\boldsymbol{x}^* \in \mathcal{V}_m$ (at the latest if $m = n$). Therefore, designing a subspace method is subdivided in two tasks. First we have to define the sequence of subspaces and secondly we have to decide in which way we select a vector out of each subspace, i.e., which condition we provide to $\boldsymbol{x}_m \in \mathcal{V}_m$ in order to get a *good* approximation for $\boldsymbol{x}^*$ [20].

Let us assume for the moment that we already have chosen these subspaces $\mathcal{V}_m$ and now look for a criterion to select $\boldsymbol{x}_m \in \mathcal{V}_m$. From approximation theory we know that an optimal subspace approximation $\boldsymbol{x}_m$ is characterized by the fact that the error $\boldsymbol{x}^* - \boldsymbol{x}_m$ stays orthogonal on the subspace where $\boldsymbol{x}_m$ is chosen from, i.e.,

$$\boldsymbol{x}^* - \boldsymbol{x}_m \perp \mathcal{V}_m \tag{1.1}$$

must hold. Unfortunately, we do not know $\boldsymbol{x}^* - \boldsymbol{x}_m$ since we do not know the exact solution $\boldsymbol{x}^*$. However, we can compute the residual $\boldsymbol{r}_m := \boldsymbol{b} - \boldsymbol{Ax}_m$ which in some sense also is a measure for the *quality* of $\boldsymbol{x}_m$. Thus, a first idea might be simply to replace $\boldsymbol{x}^* - \boldsymbol{x}_m$ by $\boldsymbol{r}_m$ in (1.1).

Since we have

$$\boldsymbol{x}^* - \boldsymbol{x}_m = \boldsymbol{A}^{-1}\boldsymbol{b} - \boldsymbol{A}^{-1}\boldsymbol{Ax}_m = \boldsymbol{A}^{-1}\boldsymbol{r}_m$$

it might be advantageous to choose $\boldsymbol{x}_m \in \mathcal{V}_m$ satisfying

$$\boldsymbol{A}^{-1}\boldsymbol{r}_m \perp \mathcal{V}_m \quad \Leftrightarrow \quad \boldsymbol{r}_m \perp \boldsymbol{A}\mathcal{V}_m, \tag{1.2}$$

that is, $\boldsymbol{r}_m$ may not stay orthogonal on $\mathcal{V}_m$ but on another subspace, say $\mathcal{W}_m$.

We fix these ideas in the following definition.

Definition 1.1 *A projecting method for solving a linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ is a procedure that constructs approximate solutions $\boldsymbol{x}_m \in \mathcal{V}_m$ under the constraint*

$$\boldsymbol{r}_m = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m \perp \mathcal{W}_m \qquad (1.3)$$

where $\mathcal{V}_m$ and $\mathcal{W}_m$ are m-dimensional subspaces of $\mathbb{R}^n$.

In the case $\mathcal{W}_m = \mathcal{V}_m$ we have an orthogonal projecting method and (1.3) is called a Galerkin condition whereas the general case $(\mathcal{W}_m \neq \mathcal{V}_m)$ is called a skew or oblique projecting method with a Petrov-Galerkin condition in equation (1.3).

In Figure 1.1 we illustrate the case $n = 2$, $m = 1$. Given $\mathcal{V}_1$ we select $\boldsymbol{x}_1 \in \mathcal{V}_1$ to satisfy $\boldsymbol{r}_1 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_1 \perp \mathcal{V}_1$ or $\boldsymbol{r}_1 \perp \mathcal{W}_1$ in the case of skew projections.

(a) orthogonal projection   (b) skew projection
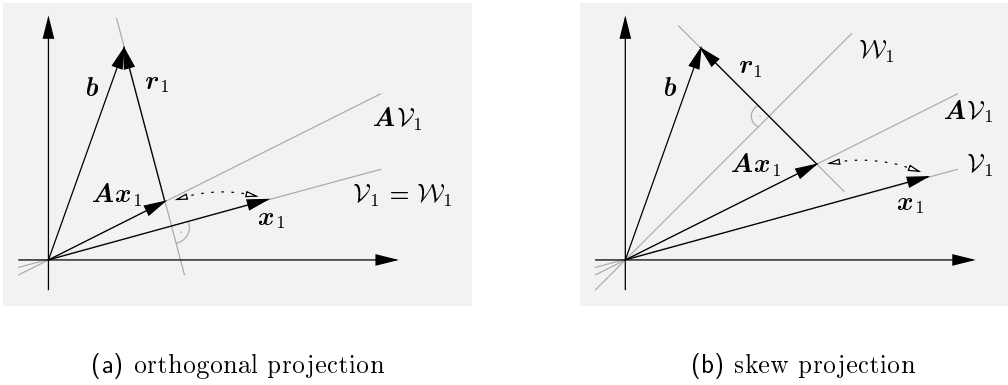
Figure 1.1: Projecting methods. Given $\mathcal{V}_1$ we compute $\boldsymbol{A}\mathcal{V}_1 = \{\boldsymbol{A}\boldsymbol{x} \mid \boldsymbol{x} \in \mathcal{V}_1\}$ and then select $\boldsymbol{x}_1 \in \mathcal{V}_1$ such that $\boldsymbol{r}_1 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_1 \perp \mathcal{V}_1$, respectively $\boldsymbol{r}_1 \perp \mathcal{W}_1$ in the case of skew projections.

Another way to characterize an optimal approximation is by its error norm. That means $\boldsymbol{x}_m \in \mathcal{V}_m$ is called optimal if $\|\boldsymbol{x}^* - \boldsymbol{x}_m\|$ minimizes $\|\boldsymbol{x}^* - \boldsymbol{x}\|$ for all $\boldsymbol{x} \in \mathcal{V}_m$. Again we have to replace $\boldsymbol{x}^* - \boldsymbol{x}_m$ by $\boldsymbol{r}_m$ (because we generally do not know $\boldsymbol{x}^*$) and fix our ideas in the following definition.

Definition 1.2 *A norm minimizing method for solving a linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ is a procedure that constructs approximate solutions $\boldsymbol{x}_m \in \mathcal{V}_m$ under the constraint*

$$\|\boldsymbol{r}_m\|_2 = \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m\|_2 = \min_{\boldsymbol{x} \in \mathcal{V}_m} \{\|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}\|_2\} \qquad (1.4)$$

where $\mathcal{V}_m$ is an m-dimensional subspace of $\mathbb{R}^n$.

We show the case $n = 2$, $m = 1$ in Figure 1.2. Given $\mathcal{V}_1$ we select $\boldsymbol{x}_1 \in \mathcal{V}_1$ minimizing $\|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}\|_2$ for all $\boldsymbol{x} \in \mathcal{V}_1$.

1.2   Generating Krylov Spaces

In this section we focus on the question of how to select the subspaces $\mathcal{V}_m$. To take a possibly given initial guess $\boldsymbol{x}_0$ into account, we allow $\mathcal{V}_m$ to be an affine subspace,
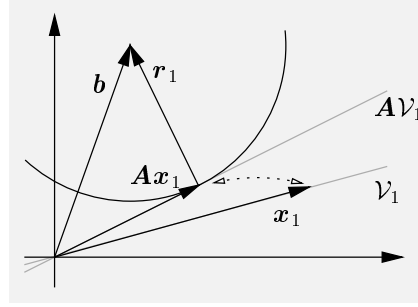
Figure 1.2: Norm minimizing methods. Given $\mathcal{V}_1$ we compute $\boldsymbol{A}\mathcal{V}_1 = \{\boldsymbol{A}\boldsymbol{x} \mid \boldsymbol{x} \in \mathcal{V}_1\}$ and then select $\boldsymbol{x}_1 \in \mathcal{V}_1$ to satisfy $\|\boldsymbol{r}_1\|_2 = \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_1\|_2 = \min_{\boldsymbol{x} \in \mathcal{V}_1}\{\|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}\|_2\}$.

i.e., $\mathcal{V}_m = \boldsymbol{x}_0 + \tilde{\mathcal{V}}_m$. There are various possibilities to choose $\tilde{\mathcal{V}}_m$ but it turns out that using Krylov subspaces has several advantageous properties as we will see later on.

Definition 1.3 *A Krylov subspace method is a projecting or a norm minimizing method (see Definitions 1.1 and 1.2) to solve a linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ where the subspaces $\tilde{\mathcal{V}}_m$ are chosen as Krylov subspaces*

$$\tilde{\mathcal{V}}_m = \mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0) := \mathrm{span}\{\boldsymbol{r}_0, \boldsymbol{A}\boldsymbol{r}_0, \dots, \boldsymbol{A}^{m-1}\boldsymbol{r}_0\}, \quad m = 1, 2, \dots \quad (1.5)$$

with $\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$.

Since we intend to work with vectors out of $\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0)$ we have to find a handy representation for them. One of the most powerful tricks in linear algebra that often simplifies problems significantly, is to find a *suitable* basis of the vector space we have to work with. Often orthonormal bases are a good choice but it is generally an enormous amount of work to compute one. The Arnoldi algorithm (Section 1.2.1) and the Lanczos algorithm (Section 1.2.2) are methods to compute orthonormal bases of Krylov spaces. In Section 1.2.3 we will see that there is a cheaper way to get a useful basis, although it is not orthonormal anymore.

1.2.1   Arnoldi Algorithm

The Krylov space $\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0)$ is given as the linear hull of $\{\boldsymbol{r}_0, \boldsymbol{A}\boldsymbol{r}_0, \dots, \boldsymbol{A}^{m-1}\boldsymbol{r}_0\}$ and a well known technique for orthonormalizing a sequence of vectors is the Gram-Schmidt algorithm. In the case $m = 1$, we simply have $\mathcal{K}_1(\boldsymbol{A}, \boldsymbol{r}_0) = \mathrm{span}\{\boldsymbol{r}_0\}$ and thus $\boldsymbol{v}_1 := \boldsymbol{r}_0/\|\boldsymbol{r}_0\|$ is a orthonormal basis (ONB) for this one dimensional Krylov space. Now suppose we have already an ONB $\boldsymbol{V}_m := \{\boldsymbol{v}_1, \dots, \boldsymbol{v}_m\}$ for $\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0)$ and look for $\boldsymbol{v}_{m+1}$ to extend $\boldsymbol{V}_m$ to $\boldsymbol{V}_{m+1}$. With Gram-Schmidt we compute $\boldsymbol{v}_{m+1}$ as

follows

$$\tilde{\boldsymbol{v}}_{m+1} = \boldsymbol{A}\boldsymbol{v}_m \qquad\qquad\qquad \text{(compute a prototype for } \boldsymbol{v}_{m+1})$$

$$\tilde{\boldsymbol{v}}_{m+1} \leftarrow \tilde{\boldsymbol{v}}_{m+1} - \sum_{i=1}^{m} \underbrace{\langle \tilde{\boldsymbol{v}}_{m+1} \mid \boldsymbol{v}_i \rangle}_{=:h_{im}} \boldsymbol{v}_i \qquad \text{(orthogonalize it against } \boldsymbol{V}_m) \qquad (1.6)$$

$$\boldsymbol{v}_{m+1} = \frac{\tilde{\boldsymbol{v}}_{m+1}}{\underbrace{\|\tilde{\boldsymbol{v}}_{m+1}\|}_{=:h_{m+1,m}}} . \qquad\qquad\qquad \text{(and finally normalize it)}$$

In the context of Krylov spaces this algorithm is called Arnoldi algorithm [6] or full orthogonalization method (FOM) because we orthogonalize $\tilde{\boldsymbol{v}}_{m+1}$ against all previous basis vectors. Collecting the coefficients $h_{i,j}$ to vectors $\boldsymbol{h}_i := (h_{i,1}|\cdots|h_{i,m})^T$ we get

$$\boldsymbol{v}_{m+1}h_{m+1,m} = \boldsymbol{A}\boldsymbol{v}_m - \sum_{i=1}^{m} \boldsymbol{v}_i h_{i,m} = \boldsymbol{A}\boldsymbol{v}_m - \boldsymbol{V}_m \boldsymbol{h}_m$$

and further arranging[2] $\boldsymbol{H}_m := (\boldsymbol{h}_1 \dots \boldsymbol{h}_m)$ yields

$$\boldsymbol{A}\boldsymbol{V}_m = \boldsymbol{V}_m \boldsymbol{H}_m + h_{m+1,m} \cdot \boldsymbol{v}_{m+1} \boldsymbol{e}_m^T, \qquad\qquad (1.7)$$

where $\boldsymbol{H}_m$ is the upper Hessenberg matrix of recurrence coefficients

$$\boldsymbol{H}_m = \begin{pmatrix} h_{1,1} & h_{1,2} & \cdots & h_{1,m-1} & h_{1,m} \\ h_{2,1} & h_{2,2} & & \vdots & \vdots \\ & h_{3,2} & & \vdots & \vdots \\ & & \ddots & \vdots & \vdots \\ & & & h_{m,m-1} & h_{m,m} \end{pmatrix} .$$

Pictorially, this matrix equation looks like

Representing $\boldsymbol{x}_m \in \mathcal{V}_m$ as $\boldsymbol{x}_m = \boldsymbol{x}_0 + \boldsymbol{V}_m \boldsymbol{\xi}_m$, i.e., $\boldsymbol{x}_m$ is a shifted linear combination of $\boldsymbol{v}_1, \dots, \boldsymbol{v}_m$, the Galerkin condition $\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m \perp \mathcal{V}_m$ now writes as

$$\boldsymbol{V}_m^T(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m) = \boldsymbol{V}_m^T(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0 - \boldsymbol{A}\boldsymbol{V}_m \boldsymbol{\xi}_m) = 0$$

$$\Leftrightarrow \underbrace{\boldsymbol{V}_m^T \boldsymbol{A}\boldsymbol{V}_m}_{\boldsymbol{A}|_{\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_o)}} \boldsymbol{\xi}_m = \boldsymbol{V}_m^T \boldsymbol{r}_0.$$

---

[2]Here and further on we collect vectors of different dimensions by extending them with trailing zeros to common length.

This also clarifies the name *orthogonal projecting* method because the restricted system matrix $\boldsymbol{A}|_{\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0)}$ is obtained by projection with the orthogonal projector $\boldsymbol{V}_m$. Multiplying equation (1.7) with $\boldsymbol{V}_m^T$ from left yields

$$\boldsymbol{V}_m^T \boldsymbol{A} \boldsymbol{V}_m = \underbrace{\boldsymbol{V}_m^T \boldsymbol{V}_m}_{\boldsymbol{I}} \boldsymbol{H}_m + h_{m+1,m} \cdot \underbrace{\boldsymbol{V}_m^T \boldsymbol{v}_{m+1}}_{0} \cdot \boldsymbol{e}_m^T = \boldsymbol{H}_m.$$

Remembering $\boldsymbol{v}_1 = \boldsymbol{r}_0 / \|\boldsymbol{r}_0\|$ we finally get

$$\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m \perp \mathcal{V}_m \quad \Leftrightarrow \quad \boldsymbol{H}_m \boldsymbol{\xi}_m = \|\boldsymbol{r}_0\| \boldsymbol{e}_1. \tag{1.8}$$

Hence, the projected system matrix is not only of smaller dimension m but is also particularly structured (upper Hessenberg).

Note that the Arnoldi algorithm can terminate before $m = n$ if $\|\tilde{\boldsymbol{v}}_{m+1}\| = 0$. Fortunately, in this case $\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0)$ is an $\boldsymbol{A}$-invariant subspace, i.e, $\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0) = \boldsymbol{A}\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0)$. Therefore we have

$$\begin{aligned} & \boldsymbol{r}_0 \in \mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0) = \boldsymbol{A}\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0) \\ \Leftrightarrow \quad & \boldsymbol{A}^{-1}\boldsymbol{r}_0 \in \mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0) \\ \Leftrightarrow \quad & \boldsymbol{x}^* - \boldsymbol{x}_0 \in \mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0) \\ \Leftrightarrow \quad & \boldsymbol{x}^* \in \boldsymbol{x}_0 + \mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0) = \mathcal{V}_m. \end{aligned} \tag{1.9}$$

That is, in this *break down* situation we can already find the solution $\boldsymbol{x}^*$ in the shifted Krylov space computed so far.

Since the Gram-Schmidt algorithm tends to be unstable if angles between $\boldsymbol{A}\boldsymbol{v}_m$ and $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m$ are small, we use a computationally more robust variant, the so called modified Gram-Schmidt algorithm. Here $\boldsymbol{A}\boldsymbol{v}_m$ is successively orthogonalized. In a pseudo programming language we can write the Arnoldi procedure with modified Gram-Schmidt as shown in Algorithm 1.1.

> Given $\boldsymbol{x}_0$
> $\tilde{\boldsymbol{v}}_1 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
> $\boldsymbol{v}_1 = \tilde{\boldsymbol{v}}_1 / \|\tilde{\boldsymbol{v}}_1\|$
> **for** $m = 1, 2, \ldots$
> $\tilde{\boldsymbol{v}}_{m+1} = \boldsymbol{A}\boldsymbol{v}_m$
>   **for** $j = 1, \ldots, m$
> $h_{j,m} = \langle \tilde{\boldsymbol{v}}_{m+1} \mid \boldsymbol{v}_j \rangle$
> $\tilde{\boldsymbol{v}}_{m+1} = \tilde{\boldsymbol{v}}_{m+1} - h_{j,m}\boldsymbol{v}_j$
> $h_{m+1,m} = \|\tilde{\boldsymbol{v}}_{m+1}\|$
> $\boldsymbol{v}_{m+1} = \tilde{\boldsymbol{v}}_{m+1} / h_{m+1,m}$

Algorithm 1.1: Arnoldi algorithm with a modified Gram-Schmidt procedure.

Unfortunately, the Arnoldi algorithm is expensive in memory and computing time since we need access to all m previous basis vectors and have to perform $\mathrm{O}(nnz + nm)$ operations in the mth iteration.

1.2.2 Lanczos Algorithm

If $\boldsymbol{A}$ is symmetric the situation becomes much more favorable. We have

$$\boldsymbol{H}_m = \boldsymbol{V}_m^T \boldsymbol{A} \boldsymbol{V}_m = (\boldsymbol{V}_m^T \boldsymbol{A}^T \boldsymbol{V}_m)^T = \boldsymbol{H}_m^T,$$

that is $\boldsymbol{H}_m$ turns out to be symmetric, too. In this case $\boldsymbol{H}_m$ is a symmetric upper Hessenberg and therefore a symmetric tridiagonal matrix which is denoted with $\boldsymbol{T}_m$, where

$$\boldsymbol{T}_m = \begin{pmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{m-2} & \alpha_{m-1} & \beta_{m-1} \\ & & & \beta_{m-1} & \alpha_m \end{pmatrix}.$$

The recursion of depth m in (1.6) reduces to one of depth three.

$$\tilde{\boldsymbol{v}}_{m+1} = \boldsymbol{A}\boldsymbol{v}_m - \beta_{m-1}\boldsymbol{v}_{m-1} \qquad \text{(compute a prototype for } \boldsymbol{v}_{m+1}$$
$$\text{and orthogonalize it against } \boldsymbol{v}_{m-1})$$

$$\tilde{\boldsymbol{v}}_{m+1} \leftarrow \tilde{\boldsymbol{v}}_{m+1} - \underbrace{\langle \tilde{\boldsymbol{v}}_{m+1} \mid \boldsymbol{v}_m \rangle}_{=:\alpha_m} \boldsymbol{v}_m \qquad \text{(orthogonalize it against } \boldsymbol{v}_m) \qquad (1.10)$$

$$\boldsymbol{v}_{m+1} = \underbrace{\frac{\tilde{\boldsymbol{v}}_{m+1}}{\|\tilde{\boldsymbol{v}}_{m+1}\|}}_{=:\beta_m} . \qquad \text{(and finally normalize it)}$$

This special case of the Arnoldi algorithm for symmetric systems is called Lanczos algorithm [77]. Similar to the nonsymmetric case we have

$$\boldsymbol{A}\boldsymbol{V}_m = \boldsymbol{V}_m \boldsymbol{T}_m + \beta_m \boldsymbol{v}_{m+1} \boldsymbol{e}_m^T \qquad (1.11)$$
$$\Leftrightarrow \quad \boldsymbol{V}_m^T \boldsymbol{A} \boldsymbol{V}_m = \underbrace{\boldsymbol{V}_m^T \boldsymbol{V}_m}_{\boldsymbol{I}} \boldsymbol{T}_m + \beta_m \cdot \underbrace{\boldsymbol{V}_m^T \boldsymbol{v}_{m+1}}_{\boldsymbol{0}} \cdot \boldsymbol{e}_m^T = \boldsymbol{T}_m.$$

and thus

$$\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m \perp \mathcal{V}_m \quad \Leftrightarrow \quad \boldsymbol{T}_m \boldsymbol{\xi}_m = \|\boldsymbol{r}_0\| \boldsymbol{e}_1. \qquad (1.12)$$

That is, the Galerkin condition reduces to a symmetric tridiagonal system of dimension m.

Algorithm 1.2 shows the Lanczos procedure in a pseudo programming language.

In the Lanczos algorithm we only need access to the last three basis vectors (no increasing memory requirements per iteration) and the number of operation is $O(nnz)$. Unfortunately, this three-term-recurrence which makes the Lanczos algorithm so favorable, only exists for symmetric matrices $\boldsymbol{A}$, at least if we use orthogonal projecting methods.

1.2.3 Bi-Lanczos Algorithm

Analyzing the Arnoldi and Lanczos algorithm we find that the most important property was the simple structure of $\boldsymbol{A}|_{\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0)}$. We now try to retain the short recurrences of the Lanczos algorithm which led to the tridiagonal shape of $\boldsymbol{A}|_{\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0)}$

Given $\boldsymbol{x}_0$
$\tilde{\boldsymbol{v}}_1 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
for $m = 1, 2, \ldots$
$\quad \beta_{m-1} = \|\tilde{\boldsymbol{v}}_m\|$
$\quad \boldsymbol{v}_m = \tilde{\boldsymbol{v}}_m / \beta_{m-1}$
$\quad \tilde{\boldsymbol{v}}_{m+1} = \boldsymbol{A}\boldsymbol{v}_m$
$\quad$ **if** $m > 1$
$\quad\quad \tilde{\boldsymbol{v}}_{m+1} = \tilde{\boldsymbol{v}}_{m+1} - \beta_{m-1}\boldsymbol{v}_{m-1}$
$\quad \alpha_m = \langle \tilde{\boldsymbol{v}}_{m+1} \mid \boldsymbol{v}_m \rangle$
$\quad \tilde{\boldsymbol{v}}_{m+1} = \tilde{\boldsymbol{v}}_{m+1} - \alpha_m \boldsymbol{v}_m$

Algorithm 1.2: Lanczos algorithm for symmetric system matrices $\boldsymbol{A}$.

but without the necessity of symmetry [112]. To achieve this, we construct a skew projecting method, that is, we choose $\boldsymbol{x}_m \in \mathcal{V}_m$ as $\boldsymbol{x}_m = \boldsymbol{x}_0 + \boldsymbol{V}_m \boldsymbol{\xi}_m$ to satisfy the Petrov-Galerkin condition

$$\boldsymbol{W}_m^T(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m) = \boldsymbol{W}_m^T(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0 - \boldsymbol{A}\boldsymbol{V}_m\boldsymbol{\xi}_m) = 0$$
$$\Leftrightarrow \quad \boldsymbol{W}_m^T \boldsymbol{A}\boldsymbol{V}_m\boldsymbol{\xi}_m = \boldsymbol{W}_m^T \boldsymbol{r}_0,$$

Here we have to compute two sets of vectors: $\boldsymbol{V}_m = (\boldsymbol{v}_1 \ldots \boldsymbol{v}_m)$ and simultaneously $\boldsymbol{W}_m = (\boldsymbol{w}_1 \ldots \boldsymbol{w}_m)$, providing a *simple* structure of $\boldsymbol{W}_m^T \boldsymbol{A}\boldsymbol{V}_m$. This can be achieved by defining $\boldsymbol{V}_m$ and $\boldsymbol{W}_m$ via a pair of coupled three-term-recurrences

$$\alpha_m = \langle \boldsymbol{A}\boldsymbol{v}_m \mid \boldsymbol{w}_m \rangle \qquad\qquad \ldots = \langle \boldsymbol{v}_m \mid \boldsymbol{A}^T\boldsymbol{w}_m \rangle$$
$$\tilde{\boldsymbol{v}}_{m+1} = \boldsymbol{A}\boldsymbol{v}_m - \alpha_m\boldsymbol{v}_m - \beta_{m-1}\boldsymbol{v}_{m-1} \qquad \tilde{\boldsymbol{w}}_{m+1} = \boldsymbol{A}^T\boldsymbol{w}_m - \alpha_m\boldsymbol{w}_m - \gamma_{m-1}\boldsymbol{w}_{m-1}$$
$$\gamma_m = \|\tilde{\boldsymbol{v}}_{m+1}\| \qquad\qquad \beta_m = \langle \tilde{\boldsymbol{v}}_{m+1} \mid \tilde{\boldsymbol{w}}_{m+1} \rangle / \gamma_m \qquad (1.13)$$
$$\boldsymbol{v}_{m+1} = \tilde{\boldsymbol{v}}_{m+1}/\gamma_m \qquad\qquad \boldsymbol{w}_{m+1} = \tilde{\boldsymbol{w}}_{m+1}/\beta_m \ ,$$

starting with $\boldsymbol{v}_1 = \boldsymbol{r}_0/\|\boldsymbol{r}_0\|$ and $\boldsymbol{w}_1 = \boldsymbol{r}_0^{dual}/\|\boldsymbol{r}_0^{dual}\|$ with $\langle \boldsymbol{r}_0 \mid \boldsymbol{r}_0^{dual} \rangle \neq 0$, e.g. $\boldsymbol{r}_0 = \boldsymbol{r}_0^{dual}$.

In matrix form these recurrences can be written as

$$\boldsymbol{A}\boldsymbol{V}_m = \boldsymbol{V}_m\boldsymbol{T}_m + \gamma_m\,\boldsymbol{v}_{m+1}\,\boldsymbol{e}_m^T$$
$$\boldsymbol{A}^T\boldsymbol{W}_m = \boldsymbol{W}_m\boldsymbol{T}_m^T + \beta_m\boldsymbol{w}_{m+1}\boldsymbol{e}_m^T, \qquad (1.14)$$

where $\boldsymbol{T}_m$ is the m-by-m tridiagonal matrix of recurrence coefficients

$$\boldsymbol{T}_m = \begin{pmatrix} \alpha_1 & \beta_1 & & & \\ \gamma_1 & \alpha_2 & \beta_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \gamma_{m-2} & \alpha_{m-1} & \beta_{m-1} \\ & & & \gamma_{m-1} & \alpha_m \end{pmatrix}.$$

It turns out that $\boldsymbol{V}_m$ is a basis for $\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0)$ and $\boldsymbol{W}_m$ is a basis for $\mathcal{K}_m(\boldsymbol{A}^T, \boldsymbol{r}_0)$.

Multiplying the upper equation in (1.14) with $\boldsymbol{W}_m^T$ from left yields

$$\boldsymbol{W}_m^T \boldsymbol{A}\boldsymbol{V}_m = \boldsymbol{W}_m^T \boldsymbol{V}_m\boldsymbol{T}_m + \gamma_m\boldsymbol{W}_m^T \boldsymbol{v}_{m+1}\boldsymbol{e}_m^T. \qquad (1.15)$$

To obtain the desired tridiagonal structure of $\boldsymbol{W}_m^T \boldsymbol{A} \boldsymbol{V}_m$ we need the so called bi-orthogonality condition for $\boldsymbol{V}_m$ and $\boldsymbol{W}_m$, i.e., $\boldsymbol{W}_m^T \boldsymbol{V}_m = \boldsymbol{V}_m^T \boldsymbol{W}_m = \boldsymbol{I}$.

Since it is not clear in advance that this mutual orthogonality between $\boldsymbol{W}_m$ and $\boldsymbol{V}_m$ holds, we prove it in the following theorem (compare [48]).

Theorem 1.1 *Suppose $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{m+1}$ and $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{m+1}$ exist, that is, $\langle \boldsymbol{v}_j \mid \boldsymbol{w}_j \rangle \neq 0$ for $j = 1, \ldots, m + 1$. Then $\boldsymbol{V}_m^T \boldsymbol{W}_m = \boldsymbol{W}_m^T \boldsymbol{V}_m = \boldsymbol{I}$.*

Proof: Since $\boldsymbol{v}_j = \tilde{\boldsymbol{v}}_j / \gamma_{j-1}$ and $\boldsymbol{w}_j = \tilde{\boldsymbol{w}}_j / \beta_{j-1}$ we have

$$\langle \boldsymbol{v}_j \mid \boldsymbol{w}_j \rangle = \frac{1}{\gamma_{j-1} \beta_{j-1}} \langle \tilde{\boldsymbol{v}}_j \mid \tilde{\boldsymbol{w}}_j \rangle = \frac{\langle \tilde{\boldsymbol{v}}_j \mid \tilde{\boldsymbol{w}}_j \rangle}{\| \langle \tilde{\boldsymbol{v}}_j \mid \tilde{\boldsymbol{w}}_j \rangle \|} = 1.$$

We prove the orthogonality of $\boldsymbol{v}_i$ and $\boldsymbol{w}_j$ for $i \neq j$ with $i, j \leq m + 1$ by induction.

We have $\langle \boldsymbol{v}_1 \mid \boldsymbol{w}_1 \rangle = \| \boldsymbol{r}_0 \|^{-1} \| \boldsymbol{r}_0^{dual} \|^{-1} \langle \boldsymbol{r}_0 \mid \boldsymbol{r}_0^{dual} \rangle \neq 0$. Assume that $\langle \boldsymbol{v}_i \mid \boldsymbol{w}_j \rangle = 0$ holds for $i \neq j$ with $i, j \leq m$. Because of the symmetry of (1.14) we only have to show $\langle \boldsymbol{v}_{m+1} \mid \boldsymbol{w}_j \rangle = 0$ for $j \leq m$. Then we get for $j = m$

$$\langle \boldsymbol{v}_{m+1} \mid \boldsymbol{w}_m \rangle = \frac{1}{\gamma_m} \langle \boldsymbol{A} \boldsymbol{v}_m - \alpha_m \boldsymbol{v}_m - \beta_{m-1} \boldsymbol{v}_{m-1} \mid \boldsymbol{w}_m \rangle$$

$$= \frac{1}{\gamma_m} \Big(\underbrace{\langle \boldsymbol{A} \boldsymbol{v}_m \mid \boldsymbol{w}_m \rangle}_{= \alpha_m} - \alpha_m \underbrace{\langle \boldsymbol{v}_m \mid \boldsymbol{w}_m \rangle}_{= 1} - \beta_{m-1} \underbrace{\langle \boldsymbol{v}_{m-1} \mid \boldsymbol{w}_m \rangle}_{= 0} \Big)$$

$$= \frac{1}{\gamma_m} (\alpha_m - \alpha_m) = 0,$$

for $j = m - 1$ we get

$$\langle \boldsymbol{v}_{m+1} \mid \boldsymbol{w}_{m-1} \rangle = \frac{1}{\gamma_m} \langle \boldsymbol{A} \boldsymbol{v}_m - \alpha_m \boldsymbol{v}_m - \beta_{m-1} \boldsymbol{v}_{m-1} \mid \boldsymbol{w}_{m-1} \rangle$$

$$= \frac{1}{\gamma_m} \Big(\langle \boldsymbol{A} \boldsymbol{v}_m \mid \boldsymbol{w}_{m-1} \rangle - \alpha_m \underbrace{\langle \boldsymbol{v}_m \mid \boldsymbol{w}_{m-1} \rangle}_{= 0} - \beta_{m-1} \underbrace{\langle \boldsymbol{v}_{m-1} \mid \boldsymbol{w}_{m-1} \rangle}_{= 1} \Big)$$

$$= \frac{1}{\gamma_m} \Big(\langle \boldsymbol{v}_m \mid \boldsymbol{A}^T \boldsymbol{w}_{m-1} \rangle - \beta_{m-1} \Big)$$

$$= \frac{1}{\gamma_m} \Big(\langle \boldsymbol{v}_m \mid \tilde{\boldsymbol{w}}_m + \alpha_{m-1} \boldsymbol{w}_{m-1} + \gamma_{m-2} \boldsymbol{w}_{m-2} \rangle - \beta_{m-1} \Big)$$

$$= \frac{1}{\gamma_m} \Big(\beta_{m-1} \underbrace{\langle \boldsymbol{v}_m \mid \boldsymbol{w}_m \rangle}_{= 1} + \alpha_{m-1} \underbrace{\langle \boldsymbol{v}_m \mid \boldsymbol{w}_{m-1} \rangle}_{= 0} + \gamma_{m-2} \underbrace{\langle \boldsymbol{v}_m \mid \boldsymbol{w}_{m-2} \rangle}_{= 0} - \beta_{m-1} \Big)$$

$$= \frac{1}{\gamma_m} (\beta_{m-1} - \beta_{m-1}) = 0,$$

and finally for $j < m - 1$ we get

$$\langle \boldsymbol{v}_{m+1} \mid \boldsymbol{w}_j \rangle = \frac{1}{\gamma_m} \langle \boldsymbol{A} \boldsymbol{v}_m \mid \boldsymbol{w}_j \rangle = \frac{1}{\gamma_m} \langle \boldsymbol{v}_m \mid \boldsymbol{A}^T \boldsymbol{w}_j \rangle = \frac{\beta_j}{\gamma_m} \langle \boldsymbol{v}_m \mid \boldsymbol{w}_{j+1} \rangle = 0.$$

∎

This means, the Bi-Lanczos algorithm provides a short recurrence formula to compute bases for Krylov spaces for nonsymmetric matrices $\boldsymbol{A}$. Although these bases are not (self) orthonormal, they suffice to obtain Lanczos like short recurrences.

However, we stress that the Bi-Lanczos process can terminate in two different situations [114]. First, if $\|\tilde{\boldsymbol{v}}_{m+1}\| = 0$ or $\|\tilde{\boldsymbol{w}}_{m+1}\| = 0$, then the algorithm has found an $\boldsymbol{A}$-invariant subspace with $\boldsymbol{x}^* \in \mathcal{V}_m$ or an $\boldsymbol{A}^T$-invariant subspace with $\boldsymbol{x}^* \in \mathcal{W}_m$, respectively. This is referred to as *regular termination* because we can find the solution in the Krylov spaces of the previous step (compare equation (1.9) on page 12).

The second case, called *serious breakdown*, occurs when $\langle \tilde{\boldsymbol{v}}_{m+1} \mid \tilde{\boldsymbol{w}}_{m+1} \rangle = 0$ but neither $\tilde{\boldsymbol{v}}_{m+1} = 0$ nor $\tilde{\boldsymbol{w}}_{m+1} = 0$. Hence we have no invariant subspace and thus cannot guarantee to find $\boldsymbol{x}^*$. However, in some later step, say $m + l$, there might exist nonzero vectors $\tilde{\boldsymbol{v}}_{m+l} \in \mathcal{K}_{m+l}(\boldsymbol{A}, \boldsymbol{r}_0)$ and $\tilde{\boldsymbol{w}}_{m+l} \in \mathcal{K}_{m+l}(\boldsymbol{A}^T, \boldsymbol{r}_0)$ with $(\boldsymbol{w}_1 | \cdots | \boldsymbol{w}_m | \boldsymbol{w}_{m+l})^T \cdot (\boldsymbol{v}_1 | \cdots | \boldsymbol{v}_m | \boldsymbol{v}_{m+l}) = \boldsymbol{I}$. Thus we have to skip these $l - 1$ intermediate steps. For practical implementations, it turns out that we also have to cover *near breakdown situations* where $\langle \tilde{\boldsymbol{v}}_{m+1} \mid \tilde{\boldsymbol{w}}_{m+1} \rangle$ is sufficiently small to cause numerical instabilities. This technique is called *look-ahead* and is described in further details in several papers including [14, 15, 63, 99, 102].

In Algorithm 1.3 we formulate the Bi-Lanczos process without look-ahead in a pseudo programming language.

> Given $\boldsymbol{x}_0$
> $\tilde{\boldsymbol{v}}_1 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
> $\boldsymbol{v}_1 = \boldsymbol{w}_1 = \tilde{\boldsymbol{v}}_1 / \|\tilde{\boldsymbol{v}}_1\|$
> **for** $m = 1, 2, \ldots$
> $\quad \tilde{\boldsymbol{v}}_{m+1} = \boldsymbol{A}\boldsymbol{v}_m$
> $\quad \tilde{\boldsymbol{w}}_{m+1} = \boldsymbol{A}^T \boldsymbol{w}_m$
> $\quad \alpha_m = \langle \tilde{\boldsymbol{v}}_{m+1} \mid \boldsymbol{w}_m \rangle$
> $\quad \tilde{\boldsymbol{v}}_{m+1} = \tilde{\boldsymbol{v}}_{m+1} - \alpha_m \boldsymbol{v}_m$
> $\quad \tilde{\boldsymbol{w}}_{m+1} = \tilde{\boldsymbol{w}}_{m+1} - \alpha_m \boldsymbol{w}_m$
> $\quad$ **if** $m > 1$
> $\quad\quad \tilde{\boldsymbol{v}}_{m+1} = \tilde{\boldsymbol{v}}_{m+1} - \beta_{m-1} \boldsymbol{v}_{m-1}$
> $\quad\quad \tilde{\boldsymbol{w}}_{m+1} = \tilde{\boldsymbol{w}}_{m+1} - \gamma_{m-1} \boldsymbol{w}_{m-1}$
> $\quad \gamma_m = \|\tilde{\boldsymbol{v}}_{m+1}\|$
> $\quad \boldsymbol{v}_{m+1} = \tilde{\boldsymbol{v}}_{m+1} / \gamma_m$
> $\quad \beta_m = \langle \boldsymbol{v}_{m+1} \mid \tilde{\boldsymbol{w}}_{m+1} \rangle$
> $\quad \boldsymbol{w}_{m+1} = \tilde{\boldsymbol{w}}_{m+1} / \beta_m$

Algorithm 1.3: The Bi-Lanczos algorithm without look-ahead.

1.3   Convergence Properties

The biggest part of convergence theory and error estimates is for Lanczos procedures used as eigenvalue solvers. Many results are collected under the name *Kaniel-Paige-theory* [66, 93–95, 142] concerning the relations between the eigenvalues of $\boldsymbol{T}$ and $\boldsymbol{A}$

as well as convergence of the Ritz values (compare Theorem 2.1).

Many of these results are applicable to Lanczos procedures used to solve linear systems and thus we have a good knowledge about convergence at least for symmetric systems [51, 62, 128, 132], see Section 1.3.1.

Unfortunately, the situation becomes much less clear for nonsymmetric systems, because the proofs in the Lanczos theory are principally based on the symmetry of $\boldsymbol{A}$. However, there are some error estimates but they are neither as sharp as in the symmetric case nor practically useful at all [5, 47, 50, 64, 129, 141], see Section 1.3.2.

Strakos shows that, practically, the behavior of symmetric and nonsymmetric Krylov solvers is very similar. Unsatisfyingly, up to now nobody managed to prove this [130].

1.3.1 Symmetric Case

Convergence rates and also the quality of the iterated solutions of iterative linear system solvers depend strongly on the *good nature* of the system matrix $\boldsymbol{A}$. It can be observed that system matrices, close to the identity, are easier to solve. Closeness to $\boldsymbol{I}$ in this sense could be expressed, for example by

- $\boldsymbol{A} = \boldsymbol{I} + \boldsymbol{B}$ with rank($\boldsymbol{B}$) is small (small rank perturbation) or

- cond($\boldsymbol{A}$) ≈ 1.

Theorem 1.2 [17] *If $\boldsymbol{A} = \boldsymbol{I} + \boldsymbol{B}$ is an n by n matrix and rank($\boldsymbol{B}$) = m, then the Lanczos algorithm terminates after at most $m + 1$ steps.*

Proof: The dimension of

$$\mathcal{K}_k(\boldsymbol{A}, \boldsymbol{r}_0) = \operatorname{span}\{\boldsymbol{r}_0, \boldsymbol{A}\boldsymbol{r}_0, \ldots, \boldsymbol{A}^{k-1}\boldsymbol{r}_0\} = \operatorname{span}\{\boldsymbol{r}_0, \boldsymbol{B}\boldsymbol{r}_0, \ldots, \boldsymbol{B}^{\min\{m,k-1\}}\boldsymbol{r}_0\}$$

cannot exceed $m + 1$. Therefore at least $\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0)$ is an $\boldsymbol{A}$-invariant subspace of $\mathbb{R}^n$ and thus $\boldsymbol{A}^{-1}\boldsymbol{b} \in \mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0)$ (compare Section 1.2.2). ∎

An error bound of a different manner can be obtained in terms of the $\boldsymbol{A}$-norm ($\|\boldsymbol{z}\|_{\boldsymbol{A}} = \sqrt{\langle \boldsymbol{z} \mid \boldsymbol{A}\boldsymbol{z}\rangle}$). This norm is well defined if $\boldsymbol{A}$ is s.p.d. Therefore, the following theorem is restricted to the CG algorithm (see Section 1.5.2).

Theorem 1.3 *Suppose $\boldsymbol{A}$ is an n by n s.p.d. matrix and $\boldsymbol{b}$ is an n vector. Then for the CG-iterates $\boldsymbol{x}_k$ there holds*

$$\|\boldsymbol{x} - \boldsymbol{x}_k\|_{\boldsymbol{A}} \leq 2\sqrt{\kappa} \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)^k \|\boldsymbol{x} - \boldsymbol{x}_0\|_{\boldsymbol{A}}$$

for any $\boldsymbol{x} \in \mathbb{R}^n$ or

$$\|\boldsymbol{x}_k - \boldsymbol{x}^*\|_2 \leq 2\sqrt{\kappa} \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)^k \|\boldsymbol{x}_0 - \boldsymbol{x}^*\|_2,$$

where $\kappa = \operatorname{cond}(\boldsymbol{A})$.

See [83] for a proof. This means, the nearer κ is to one, the faster the error will decrease.

1.3.2   Nonsymmetric Case

If $\boldsymbol{A}$ is a low rank perturbation of the identity then $\boldsymbol{A}^T$ has obviously the same property (since $\boldsymbol{B}$ is square, it has equal column and row rank). Thus, Theorem 1.2 is also applicable to nonsymmetric systems.

Theorem 1.4 *If $\boldsymbol{A} = \boldsymbol{I} + \boldsymbol{B}$ is an n by n matrix and $\text{rank}(\boldsymbol{B}) = m$, then the Arnoldi and Bi-Lanczos algorithm terminate after at most $m + 1$ steps.*

Proof: The dimensions of

$$\begin{aligned} \mathcal{K}_k(\boldsymbol{A}, \boldsymbol{r}_0) &= \text{span}\{\boldsymbol{r}_0, \boldsymbol{A}\boldsymbol{r}_0, \ldots, \boldsymbol{A}^{k-1}\boldsymbol{r}_0\} \\ &= \text{span}\{\boldsymbol{r}_0, \boldsymbol{B}\boldsymbol{r}_0, \ldots, \boldsymbol{B}^{\min\{m,k-1\}}\boldsymbol{r}_0\} \end{aligned}$$

and

$$\begin{aligned} \mathcal{K}_k(\boldsymbol{A}^T, \boldsymbol{r}_0) &= \text{span}\{\boldsymbol{r}_0, \boldsymbol{A}^T\boldsymbol{r}_0, \ldots, (\boldsymbol{A}^{k-1})^T\boldsymbol{r}_0\} \\ &= \text{span}\{\boldsymbol{r}_0, \boldsymbol{B}^T\boldsymbol{r}_0, \ldots, (\boldsymbol{B}^{\min\{m,k-1\}})^T\boldsymbol{r}_0\} \end{aligned}$$

cannot exceed $m + 1$. Therefore, at least $\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0)$ or $\mathcal{K}_m(\boldsymbol{A}^T, \boldsymbol{r}_0)$ is an $\boldsymbol{A}$-invariant (respectively $\boldsymbol{A}^T$-invariant) subspace of $\mathbb{R}^n$ and therefore $\boldsymbol{A}^{-1}\boldsymbol{b}$ is either in $\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0)$ or in $\mathcal{K}_m(\boldsymbol{A}^T, \boldsymbol{r}_0)$ (compare Sections 1.2.1 and 1.2.3). ∎

To illustrate the difficulties with error bounds for nonsymmetric matrices we present some results for GMRES (see Section 1.5.5). The 2-norm of the kth GMRES-residual $\boldsymbol{r}_k$ satisfies

$$\|\boldsymbol{r}_k\|_2 = \min_{\substack{\phi_k \in \mathcal{P}_k \\ \phi_k(0)=1}} \|\phi_k(\boldsymbol{A})\boldsymbol{r}_0\|_2 \tag{1.16}$$

where $\mathcal{P}_k$ is set set of polynomials of degree k or less [64]. Suppose $\boldsymbol{A}$ is diagonalizable then there exists an eigen-decomposition $\boldsymbol{A} = \boldsymbol{S}\boldsymbol{\Lambda}\boldsymbol{S}^{-1}$ where $\boldsymbol{\Lambda} = \text{diag}(\lambda_1, \ldots, \lambda_n)$ and the columns of $\boldsymbol{S}$ are the eigenvectors of $\boldsymbol{A}$. From (1.16) we obtain

$$\|\boldsymbol{r}_k\|_2 = \min_{\substack{\phi_k \in \mathcal{P}_k \\ \phi_k(0)=1}} \|\boldsymbol{S}\phi_k(\boldsymbol{\Lambda})\boldsymbol{S}^{-1}\boldsymbol{r}_0\|_2 \leq \text{cond}_2(\boldsymbol{S}) \min_{\substack{\phi_k \in \mathcal{P}_k \\ \phi_k(0)=1}} \|\phi_k(\boldsymbol{\Lambda})\|_2 \|\boldsymbol{r}_0\|_2$$

$$\Leftrightarrow \quad \frac{\|\boldsymbol{r}_k\|_2}{\|\boldsymbol{r}_0\|_2} \leq \text{cond}_2(\boldsymbol{S}) \min_{\substack{\phi_k \in \mathcal{P}_k \\ \phi_k(0)=1}} \left\{ \max_{i=1}^{n} |\phi_k(\lambda_i)| \right\}$$

If $\boldsymbol{A}$ is non-normal, then $\boldsymbol{S}$ does not need not to be unitary and thus $\text{cond}_2(\boldsymbol{S}) > 1$. Consequently, convergence of GMRES, or at least this bound of the residual norm does not solely depend on the eigenvalues of $\boldsymbol{A}$. Additionally, it can be shown that

Theorem 1.5 *Given a non-increasing positive sequence $r_0 \geq r_1 \geq \ldots \geq r_{n-1} > 0$ and an arbitrary set of nonzero complex numbers $\{\lambda_1, \ldots, \lambda_n\}$, there exists a matrix $\boldsymbol{A}$ with eigenvalues $\lambda_1, \ldots, \lambda_n$ and an initial residual $\boldsymbol{r}_0$ with $\|\boldsymbol{r}_0\|_2 = r_0$ such that the residual vectors $\boldsymbol{r}_k$ at each step of the GMRES method applied to $\boldsymbol{A}$ and $\boldsymbol{r}_0$ satisfy $\|\boldsymbol{r}_k\|_2 = r_k$ for $k = 1, 2, \ldots, n - 1$.*

See [49] for a proof.

The situation becomes even more difficult, if we have to take rounding errors into consideration. However, the aim of this work is not to fix the arithmetic and then try to get the best result, but it is to fix our requirements on accuracy and then to choose an appropriate arithmetic that enables us to reach this needed accuracy.

1.4   Preconditioners

The idea of preconditioning, i.e., *preliminary* reduction of the *condition* number, simply is to replace the original linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ by a modified linear system $\tilde{\boldsymbol{A}}\boldsymbol{x} = \tilde{\boldsymbol{b}}$, where this second system has to fulfill two properties [7, 8]:

- Solving $\tilde{\boldsymbol{A}}\boldsymbol{x} = \tilde{\boldsymbol{b}}$ should be (more) easy and

- $\tilde{\boldsymbol{A}}\boldsymbol{x} = \tilde{\boldsymbol{b}}$ and $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ must have the same solution, i.e., $\tilde{\boldsymbol{A}}^{-1}\tilde{\boldsymbol{b}} = \boldsymbol{A}^{-1}\boldsymbol{b}$.

Here, we demonstrate preconditioning in context of the simplest algorithm for generating Krylov subspaces — the Lanczos algorithm. Since this algorithm only works for symmetric systems, we have to retain the symmetry in $\tilde{\boldsymbol{A}}$. Therefore we make the ansatz $\tilde{\boldsymbol{A}} = \boldsymbol{L}^{-1}\boldsymbol{A}\boldsymbol{L}^{-T}$ with a nonsingular matrix $\boldsymbol{L}$, where $\boldsymbol{M} := \boldsymbol{L}\boldsymbol{L}^{T}$ shall in some sense be near to $\boldsymbol{A}$. To ensure the equivalence of the preconditioned and non-preconditioned system, we have to define $\tilde{\boldsymbol{b}} = \boldsymbol{L}^{-1}\boldsymbol{b}$ and $\tilde{\boldsymbol{x}} = \boldsymbol{L}^{T}\boldsymbol{x}$ to get

$$\tilde{\boldsymbol{A}}\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{b}}$$
$$\Leftrightarrow \quad \boldsymbol{L}^{-1}\boldsymbol{A}\boldsymbol{L}^{-T}\boldsymbol{L}^{T}\boldsymbol{x} = \boldsymbol{L}^{-1}\boldsymbol{b}$$
$$\Leftrightarrow \quad \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}.$$

Simply replacing $\boldsymbol{A}$ by $\tilde{\boldsymbol{A}} = \boldsymbol{L}^{-1}\boldsymbol{A}\boldsymbol{L}^{-T}$ and $\boldsymbol{b}$ by $\tilde{\boldsymbol{b}} = \boldsymbol{L}^{-1}\boldsymbol{b}$ and for technical reasons also renaming $\boldsymbol{v}$ by $\boldsymbol{z}$ in the Lanczos algorithm (compare Algorithm 1.2) leads to

$$
\begin{array}{ll}
1 & \text{Given } \boldsymbol{x}_0,\ \tilde{\boldsymbol{x}}_0 = \boldsymbol{L}^{T}\boldsymbol{x}_0 \\
2 & \tilde{\boldsymbol{z}}_1 = \boldsymbol{L}^{-1}\boldsymbol{b} - \boldsymbol{L}^{-1}\boldsymbol{A}\boldsymbol{L}^{-T}\boldsymbol{L}^{T}\boldsymbol{x}_0 = \boldsymbol{L}^{-1}(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0) \\
3 & \textbf{for } m = 1, 2, \ldots \\
4 & \quad \beta_{m-1} = \|\tilde{\boldsymbol{z}}_m\| \\
5 & \quad \boldsymbol{z}_m = \tilde{\boldsymbol{z}}_m / \beta_{m-1} \\
6 & \quad \tilde{\boldsymbol{z}}_{m+1} = \boldsymbol{L}^{-1}\boldsymbol{A}\boldsymbol{L}^{-T}\boldsymbol{z}_m \\
7 & \quad \textbf{if } m > 1 \\
8 & \quad\quad \tilde{\boldsymbol{z}}_{m+1} = \tilde{\boldsymbol{z}}_{m+1} - \beta_{m-1}\boldsymbol{z}_{m-1} \\
9 & \quad \alpha_m = \langle \tilde{\boldsymbol{z}}_{m+1} \mid \boldsymbol{z}_m \rangle \\
10 & \quad \tilde{\boldsymbol{z}}_{m+1} = \tilde{\boldsymbol{z}}_{m+1} - \alpha_m \boldsymbol{z}_m
\end{array}
$$

From line 2 and 5 we see, comparing with Algorithm 1.2,

$$\boldsymbol{v}_m = \boldsymbol{L}\boldsymbol{z}_m, \quad \text{and} \quad \tilde{\boldsymbol{v}}_m = \boldsymbol{L}\tilde{\boldsymbol{z}}_m.$$

To avoid the explicit use of $\boldsymbol{L}^{-1}$ and $\boldsymbol{L}^{-T}$ we introduce two auxiliary vectors $\boldsymbol{p}_m$ and $\tilde{\boldsymbol{p}}_m$, defined by

$$\boldsymbol{p}_m = \boldsymbol{M}^{-1}\boldsymbol{v}_m = \boldsymbol{L}^{-T}\boldsymbol{L}^{-1}\boldsymbol{v}_m, \quad \text{and} \quad \tilde{\boldsymbol{p}}_m = \boldsymbol{M}^{-1}\tilde{\boldsymbol{v}}_m.$$

Now we substitute the vectors $\boldsymbol{z}_m$ and $\tilde{\boldsymbol{z}}_m$ by expressions with $\boldsymbol{v}_m$, $\tilde{\boldsymbol{v}}_m$, $\boldsymbol{p}_m$, and $\tilde{\boldsymbol{p}}_m$. With line 4 we get

$$\begin{aligned}
\beta_{m-1} = \|\tilde{\boldsymbol{z}}_m\| &= \langle \boldsymbol{L}^{-1}\tilde{\boldsymbol{v}}_m \mid \boldsymbol{L}^{-1}\tilde{\boldsymbol{v}}_m \rangle^{1/2} \\
&= \langle \tilde{\boldsymbol{v}}_m \mid \boldsymbol{L}^{-T}\boldsymbol{L}^{-1}\tilde{\boldsymbol{v}}_m \rangle^{1/2} \\
&= \langle \tilde{\boldsymbol{v}}_m \mid \tilde{\boldsymbol{p}}_m \rangle^{1/2}.
\end{aligned}$$

Using $\tilde{\boldsymbol{z}}_{m+1} = \boldsymbol{L}^{-1}\tilde{\boldsymbol{v}}_{m+1}$ and $\boldsymbol{L}^{-T}\boldsymbol{z}_m = \boldsymbol{p}_m$ we get in line 6

$$\tilde{\boldsymbol{z}}_{m+1} = \boldsymbol{L}^{-1}\boldsymbol{A}\boldsymbol{p}_m \quad \Leftrightarrow \quad \boldsymbol{L}\tilde{\boldsymbol{z}}_{m+1} = \boldsymbol{A}\boldsymbol{p}_m \quad \Leftrightarrow \quad \tilde{\boldsymbol{v}}_{m+1} = \boldsymbol{A}\boldsymbol{p}_m.$$

Multiplying lines 8 and 10 with $\boldsymbol{L}$ from left yields $\tilde{\boldsymbol{v}}_{m+1} = \tilde{\boldsymbol{v}}_{m+1} - \beta_{m-1}\boldsymbol{v}_{m-1}$ and $\tilde{\boldsymbol{v}}_{m+1} = \tilde{\boldsymbol{v}}_{m+1} - \alpha_m\boldsymbol{v}_m$. With line 9 we have

$$\alpha_m = \langle \tilde{\boldsymbol{z}}_{m+1} \mid \boldsymbol{z}_m \rangle = \langle \boldsymbol{L}^{-1}\tilde{\boldsymbol{v}}_{m+1} \mid \boldsymbol{L}^T\boldsymbol{p}_m \rangle = \langle \tilde{\boldsymbol{v}}_{m+1} \mid \boldsymbol{p}_m \rangle.$$

We collect our results in Algorithm 1.4 (left)

Given $\boldsymbol{x}_0$	Given $\boldsymbol{x}_0$
$\tilde{\boldsymbol{v}}_1 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$	$\tilde{\boldsymbol{v}}_1 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
for $m = 1, 2, \ldots$	**for** $m = 1, 2, \ldots$
$\quad \tilde{\boldsymbol{p}}_m = \boldsymbol{M}^{-1}\tilde{\boldsymbol{v}}_m$	
$\quad \beta_{m-1} = \langle \tilde{\boldsymbol{v}}_m \mid \tilde{\boldsymbol{p}}_m \rangle^{1/2}$	$\quad \beta_{m-1} = \|\tilde{\boldsymbol{v}}_m\|$
$\quad \boldsymbol{v}_m = \tilde{\boldsymbol{v}}_m/\beta_{m-1}$	$\quad \boldsymbol{v}_m = \tilde{\boldsymbol{v}}_m/\beta_{m-1}$
$\quad \boldsymbol{p}_m = \tilde{\boldsymbol{p}}_m/\beta_{m-1}$	
$\quad \tilde{\boldsymbol{v}}_{m+1} = \boldsymbol{A}\boldsymbol{p}_m$	$\quad \tilde{\boldsymbol{v}}_{m+1} = \boldsymbol{A}\boldsymbol{v}_m$
$\quad$ **if** $m > 1$	$\quad$ **if** $m > 1$
$\quad\quad \tilde{\boldsymbol{v}}_{m+1} = \tilde{\boldsymbol{v}}_{m+1} - \beta_{m-1}\boldsymbol{v}_{m-1}$	$\quad\quad \tilde{\boldsymbol{v}}_{m+1} = \tilde{\boldsymbol{v}}_{m+1} - \beta_{m-1}\boldsymbol{v}_{m-1}$
$\quad \alpha_m = \langle \tilde{\boldsymbol{v}}_{m+1} \mid \boldsymbol{p}_m \rangle$	$\quad \alpha_m = \langle \tilde{\boldsymbol{v}}_{m+1} \mid \boldsymbol{v}_m \rangle$
$\quad \tilde{\boldsymbol{v}}_{m+1} = \tilde{\boldsymbol{v}}_{m+1} - \alpha_m\boldsymbol{v}_m$	$\quad \tilde{\boldsymbol{v}}_{m+1} = \tilde{\boldsymbol{v}}_{m+1} - \alpha_m\boldsymbol{v}_m$

Algorithm 1.4: A (symmetrically) preconditioned Lanczos algorithm with preconditioner $\boldsymbol{M} := \boldsymbol{L}\boldsymbol{L}^T$ (left) and its non-preconditioned variant (right).

As we can see from Algorithm 1.4, the essential modification is the computation of the solution of $\boldsymbol{M}\tilde{\boldsymbol{p}}_m = \tilde{\boldsymbol{v}}_m$ with a matrix $\boldsymbol{M}$ similar to $\boldsymbol{A}$. In other words, we need an approximate solution of $\boldsymbol{A}\tilde{\boldsymbol{p}}_m = \tilde{\boldsymbol{v}}_m$. At a first glance it seems to make no sense to solve $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ by repeatedly solving $\boldsymbol{A}\tilde{\boldsymbol{p}}_m = \tilde{\boldsymbol{v}}_m$ but the emphasis is on *approximate solution*, that is, we only need a fast approximation even though it is a rough one.

In this sense, every linear system solver can be applied as a preconditioner. Due to the nature of these solvers, preconditioners can be divided roughly into three categories:

- Preconditioners based on simple iterative solvers, e.g., Jacobi, Gauß-Seidel, or SOR preconditioners.

- Preconditioners based on direct solvers, modified for fast but approximative solving, e.g., incomplete Cholesky or incomplete LU (ILU) or modified variants [11].

- Problem specific preconditioners, either designed for a broad class of underlying problems or even for one specific matrix or problem. For example, there are preconditioners for elliptic PDE's, namely multigrid or domain decomposition preconditioners [26], or special preconditioners as the diffusion synthetic acceleration preconditioner (DSA), solely designed for the transport equation (see [88]).

In the following two sections, we give an introductory overview about the first two categories. Since this work focuses on generic linear system solving, we won't consider the special preconditioners, described in the last category.

1.4.1 Splitting Techniques

Historically, the first class of iterative solvers for linear systems of equations was based on so called splitting techniques [55, 127, 135]. There we split the matrix $\boldsymbol{A}$ in a sum of two matrices, say $\boldsymbol{B}$ and $\boldsymbol{A} - \boldsymbol{B}$ and then write $\boldsymbol{B}\boldsymbol{x} = (\boldsymbol{B} - \boldsymbol{A})\boldsymbol{x} + \boldsymbol{b}$ instead of $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. If $\boldsymbol{B}$ is nonsingular, we obtain the following fixed point formulation

$$\boldsymbol{x} = \boldsymbol{B}^{-1}(\boldsymbol{B} - \boldsymbol{A})\boldsymbol{x} + \boldsymbol{B}^{-1}\boldsymbol{b}.$$

Substituting the left hand side $\boldsymbol{x}$ by $\boldsymbol{x}_{k+1}$ and the right hand side $\boldsymbol{x}$ by $\boldsymbol{x}_k$, we get the iteration scheme

$$\boldsymbol{x}_{k+1} = \boldsymbol{B}^{-1}(\boldsymbol{B} - \boldsymbol{A})\boldsymbol{x}_k + \boldsymbol{B}^{-1}\boldsymbol{b} = \boldsymbol{x}_k - \boldsymbol{B}^{-1}\boldsymbol{A}\boldsymbol{x}_k + \boldsymbol{B}^{-1}\boldsymbol{b} \qquad (1.17)$$

which is convergent if and only if the spectral radius of $\boldsymbol{B}^{-1}(\boldsymbol{B} - \boldsymbol{A})$ is less than one. In the case of convergence, i.e., with $\boldsymbol{x}^* = \lim_{k \to \infty}(\boldsymbol{x}_k)$ we have $\boldsymbol{B}^{-1}\boldsymbol{A}\boldsymbol{x}^* + \boldsymbol{B}^{-1}\boldsymbol{b} = 0$ or $\boldsymbol{A}\boldsymbol{x}^* = \boldsymbol{b}$.

To utilize the iteration scheme (1.17) as a preconditioner, we only perform one iteration step. The main effort is solving the linear system $\boldsymbol{B}\boldsymbol{z} = (\boldsymbol{B} - \boldsymbol{A})\boldsymbol{x}$ for $\boldsymbol{z}$, that is, we should chose $\boldsymbol{B}$ to assure that this solution is easily computable. According to different choices of $\boldsymbol{B}$ we have different algorithms.

- *Jacobi Preconditioners*

For the Jacobi algorithm, we choose $\boldsymbol{B} = \mathrm{diag}(\boldsymbol{A})$ (see Figure 1.3). This leads to the simple preconditioner

$$z_i = x_i - \frac{1}{a_{i,i}} \sum_{j=1}^{n} a_{i,j} x_j + \frac{b_i}{a_{i,i}}$$

$$= \frac{1}{a_{i,i}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{i,j} x_j \right), \quad \text{for } i = 1, \dots, n.$$
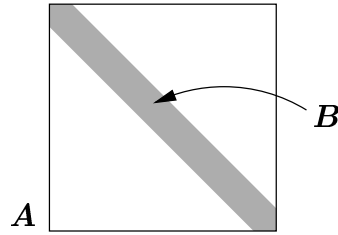
Figure 1.3: Splitting scheme for the Jacobi iteration. $\boldsymbol{B} = \mathrm{diag}(\boldsymbol{A})$.

• *Gauß-Seidel Preconditioners*

Here we set $\boldsymbol{B} = \mathrm{lowerTriangle}(\boldsymbol{A})$ (see Figure 1.4), This leads to the Gauß-Seidel algorithm which is related to the solution of a triangular system

$$z_i = x_i - \frac{1}{a_{i,i}} \left(\sum_{j=1}^{i-1} a_{i,j} z_j + \sum_{j=i}^{n} a_{i,j} x_j \right) + \frac{b_i}{a_{i,i}}$$

$$= \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} z_j - \sum_{j=i+1}^{n} a_{i,j} x_j \right), \quad \text{for } i = 1, \dots, n.$$

Figure 1.4: Splitting scheme for the Gauß-Seidel iteration. $\boldsymbol{B} = \mathrm{lowerTriangle}(\boldsymbol{A})$.

• *Relaxation Methods*

For both, the Jacobi and the Gauß-Seidel algorithm, one can scale the matrix $\boldsymbol{B}$ by a so called relaxation parameter ω. This leads to the Jacobi relaxation preconditioner

$$z_i = x_i - \frac{\omega}{a_{i,i}} \sum_{j=1}^{n} a_{i,j} x_j + \omega \frac{b_i}{a_{i,i}}$$

$$= (1 - \omega) x_i - \frac{\omega}{a_{i,i}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{i,j} x_j \right), \quad \text{for } i = 1, \dots, n.$$

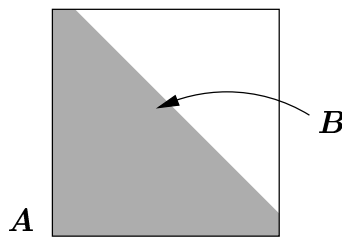or the Gauß-Seidel relaxation preconditioner, also known as SOR (successive over relaxation) preconditioner

$$z_i = x_i - \frac{\omega}{a_{i,i}} \left(\sum_{j=1}^{i-1} a_{i,j} z_j + \sum_{j=i}^{n} a_{i,j} x_j \right) + \omega \frac{b_i}{a_{i,i}}$$

$$= (1 - \omega) x_i - \frac{\omega}{a_{i,i}} \left(\sum_{j=1}^{i-1} a_{i,j} z_j + \sum_{j=i+1}^{n} a_{i,j} x_j \right), \quad \text{for } i = 1, \ldots, n.$$

These algorithms can be interpreted as a weighted sum of the non-preconditioned vector $\boldsymbol{x}$ and the *update* vector $\boldsymbol{z} - \boldsymbol{x}$. To ensure positive weights we must choose $\omega \in (0, 2)$. A *good* relaxation parameter can improve the preconditioner significantly but in general it is hard to determine an optimal relaxation parameter ω. However, there are several works (see e.g. [2, 135]) dealing with this question depending on special properties of the matrix $\boldsymbol{A}$.

1.4.2 Incomplete Decompositions

This important class of generic preconditioners is based on direct solvers, i.e., on a multiplicative decomposition, say $\boldsymbol{L} \cdot \boldsymbol{U}$ of $\boldsymbol{A}$. Favorably, we deal with triangular factors $\boldsymbol{L}$ and $\boldsymbol{U}$ as delivered, for example, by the LU-decomposition. Since solving $\boldsymbol{L}\boldsymbol{U}\boldsymbol{z} = \boldsymbol{x}$ for $\boldsymbol{z}$ is generally neither really fast nor very approximative, we actually do not compute the entire factors $\boldsymbol{L}$ and $\boldsymbol{U}$ but only incomplete factors $\tilde{\boldsymbol{L}}$ and $\tilde{\boldsymbol{U}}$. That means, we compute only a subset of the elements of these triangular matrices. There are basically two strategies how to decide, whether an element of the complete triangular factor is to be taken up into the sparse triangular factor or whether it can be dropped:

- Compute an element $l_{i,j}$ of $\tilde{\boldsymbol{L}}$ (or $u_{i,j}$ of $\tilde{\boldsymbol{U}}$, respectively) depending on memory management considerations. Usually we compute an element at place (i, j) if and only if $a_{i,j} \neq 0$. In this case we can use the storage scheme of $\boldsymbol{A}$ to store $\tilde{\boldsymbol{L}}$ and $\tilde{\boldsymbol{U}}$ in.

- Compute an element $l_{i,j}$ of $\tilde{\boldsymbol{L}}$ (or $u_{i,j}$ of $\tilde{\boldsymbol{U}}$, respectively) depending on its *importance*. Usually this importance is measured in the following sense. The sparse triangular factors are computed columnwise. An element is dropped if it is smaller than a given *drop-tolerance* times the norm of the corresponding column of $\boldsymbol{A}$.

  There exist various modifications [11], one, for example, tries to save some of the dropped information by adding the dropped elements to the diagonal element of the upper diagonal factor to retain the column-norms of $\boldsymbol{A}$.

The LU-decomposition works for arbitrary nonsingular matrices $\boldsymbol{A}$. However, there are some variants exploiting special properties of $\boldsymbol{A}$ such as symmetry or positive definiteness. Since we make extensive use of these preconditioners (see Section 4.3.3, we describe some important LU variants here.

• LDM^T Decomposition of General Matrices

For this variant, we actually do not need a special structure in A. In the usual LU factorization L tends to be well-conditioned whereas the condition number of A moves into U [41]. Here we factorize A into a three-matrices product LDM^T where D is diagonal and L and M both are unit lower triangular. Subsequently we distribute D among L and M, i.e., we define

$$D_1 := \sqrt{|D|}, \quad D_2 := \text{sign}(D)\sqrt{|D|}, \quad \hat{L} := LD_1, \quad \hat{U} := MD_2.$$

This leads to a $\hat{L}\hat{U}^T$ factorization and heuristically, $\hat{L}$ and $\hat{U}$ have a more or less equal condition number $\sqrt{\text{cond}(A)}$. This modification is important for verifying an approximate solution of a linear system as described in Section 4.3.3. There we need the smallest singular value of A which we estimate by $\sigma_{\min}(\hat{L}) \cdot \sigma_{\min}(\hat{U})$. Since these singular values are the square root of the eigenvalues of $\hat{L}\hat{L}^T$ and $\hat{U}\hat{U}^T$ we have to compute eigenvalues of matrices with condition numbers $\text{cond}(\hat{L})^2$ respectively $\text{cond}(\hat{U})^2$. Without the balancing, i.e. with $\text{cond}(U) \approx \text{cond}(A)$, this would limit us to linear systems with moderate condition numbers less than $\epsilon^{-1/2}$.

To obtain sparse or incomplete triangular factors, obviously all modifications described for the LU-factorization can be applied.

• LDL^T Decomposition of Symmetric Matrices

Here we suppose A to be symmetric, then we have redundancy in the LDM^T algorithm since in this case $L = M$. This can be seen by multiplying $A = LDM^T$ with M^{-1} from left and M^{-T} from right. This yields

$$M^{-1}AM^{-T} = M^{-1}LDM^TM^{-T} = M^{-1}LD.$$

The left hand side is symmetric and the right hand side is lower triangular and thus $M^{-1}LD$ is diagonal. Since D is nonsingular, this implies $M^{-1}L$ is also diagonal. But $M^{-1}L$ is unit lower triangular and so $M^{-1}L = I$ (see [41]). Thus we can omit computing M. Again distributing D yields

$$D_1 := \sqrt{|D|}, \quad D_2 := \text{sign}(D)\sqrt{|D|}, \quad \hat{L} := LD_1, \quad \hat{U} := LD_2.$$

In this symmetric case we even have

$$\text{cond}(\hat{L}) = \text{cond}(L|D|^{1/2}) = \text{cond}(L|D|^{1/2}\text{sign}(D)) = \text{cond}(\hat{U}).$$

• Cholesky or LL^T Decomposition of S.P.D. Matrices

Moreover, if A is symmetric positive definite (s.p.d.), i.e., if $x^TAx > 0$ for all $x \neq o$ then we have

$$0 < (L^{-T}e_i)^TA(L^{-T}e_i) = e_i^TL^{-1}AL^{-T}e_i = e_i^TD\,e_i = d_{i,i}.$$

Thus all elements of D are positive which enables us to define

$$\hat{D} := \sqrt{D}, \quad \hat{L} := LD.$$

Together we get $A = \hat{L}\hat{L}^T$, that is $\hat{L}$ is the Cholesky factor of A.

● *Pivoting and Reordering Algorithms*

Suppose $\boldsymbol{L}\boldsymbol{U}^T$ is a triangular factorization of $\boldsymbol{A}$, computed in finite precision. Then we have $\boldsymbol{L}\boldsymbol{U}^T = \tilde{\boldsymbol{A}} \approx \boldsymbol{A}$. For the error matrix $\tilde{\boldsymbol{A}} - \boldsymbol{A}$ we have (see [104])

$$|\tilde{\boldsymbol{A}} - \boldsymbol{A}| \leq 3(n-1)\epsilon \cdot (|\boldsymbol{A}| + |\boldsymbol{L}||\boldsymbol{U}|) + O(\epsilon^2).$$

Consequently, this error might be very large if we encounter a small pivot during factorizing $\boldsymbol{A}$ because this leads to large elements in $\boldsymbol{L}$ and $\boldsymbol{U}$. We stress that small pivots are not necessarily due to ill-conditioning as the example

$$\boldsymbol{A} = \begin{pmatrix} \epsilon & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1/\epsilon & 1 \end{pmatrix} \begin{pmatrix} \epsilon & 1 \\ 0 & -1/\epsilon \end{pmatrix} = \boldsymbol{L}\boldsymbol{U}^T$$

demonstrates. To avoid large elements in $\boldsymbol{L}$ and $\boldsymbol{U}$ we must allow some kind of pivoting [30], i.e., permuting of the rows of $\boldsymbol{A}$:

$$\boldsymbol{P}\boldsymbol{A} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \epsilon & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \epsilon & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \epsilon & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \boldsymbol{L}\boldsymbol{U}^T.$$

Usually there are two pivoting strategies: column pivoting and complete pivoting. Both limit the norm of $\boldsymbol{L}$ and $\boldsymbol{U}$, but unfortunately, they destroy a possibly given band structure. In particular we have the following situation. Suppose $\boldsymbol{A}$ to have lower bandwidth p and upper bandwidth q. Without pivoting, the original band-widths remain unchanged, with column pivoting, $\boldsymbol{U}$ has bandwidth $p+q$, while $\boldsymbol{L}$'s band-structure is completely lost, and finally with complete pivoting we loose the structure of $\boldsymbol{L}$ and $\boldsymbol{U}$. Thus, dealing with sparse matrices, we have to trade off between loosing accuracy and saving memory.

Usually, pivoting is done in each step of an LU factorization. However, there are some useful strategies how to compute permutation matrices $\boldsymbol{P}$ and $\boldsymbol{Q}$ in advance, such that an LU factorization of $\boldsymbol{P}\boldsymbol{A}\boldsymbol{Q}$ has *advantageous* properties [23,31]. Advantageous in this sense means smaller bandwidths or less nonzero elements, for example. One of the most successful algorithms is the so called 'reverse Cuthill-McKee' algorithm. This powerful graph theoretic algorithm often leads to a dramatical reduction of the numbers of nonzeros in $\boldsymbol{L}$ and $\boldsymbol{U}$ and therefore to a large speedup in solving and particularly in verifying a sparse linear system of equations.

1.5 Krylov Type Linear System Solver

In this section we first give an overview about various Krylov type linear system solver [44, 60]. Since our improvements in convergence, speed, and accuracy as well as our verification methods do not depend on the particular method, we only describe some of the most important variants in more detail. These are CG (Section 1.5.2), BiCG (Section 1.5.3), CGS (Section 1.5.4), and GMRES (Section 1.5.5). Additionally, we present the basic ideas of residual norm smoothing (BiCGStab, QMRCGStab) and quasi minimization (QMR, TFQMR), see Section 1.5.6.

For each described method, we give a preconditioned algorithm formulated in a pseudo programming language, taken from [9]

symm.pos.def.

CG
Hestenes, Stiefel '52 [59]
Lanczos | Galerkin

symm.indef.

SYMMLQ
Paige, Saunders '75 [97]
Lanczos | —

arbitrary

CGNE
Paige, Saunders '79 [98]
Lanczos | Galerkin

MINRES
Paige, Saunders '75 [97]
Lanczos | norm.min.

CGNR
Paige, Saunders '82 [98]
Lanczos | norm.min.

GMRES
Saad, Schulz '86 [115]
Arnoldi | norm.min.

BiCG
Fletcher '75 [35]
Bi-Lanczos | Petrov-Galerkin

QMR
Freund, Nachtigal '91 [91]
Bi-Lanczos | quasi norm.min.

CGS
Sonnefeld '89 [124]
Bi-Lanczos | Petrov-Galerkin

TFQMR
Freund '93 [37]
Bi-Lanczos | quasi norm.min.

BiCGStab
van de Vorst '92 [134]
Bi-Lanczos | —

QMRCGStab
Chan et al. '94 [18]
Bi-Lanczos | —

Figure 1.5: Krylov type linear system solver

1.5.1 Overview

The oldest and probably best known Krylov type method is the **C**onjugate **G**radient (CG) method, developed by Hestenes and Stiefel in 1952 [59]. It was designed for solving systems of linear equations with symmetric positive definite coefficient matrices. Possibly because matrix dimensions were small at this time and CG was considered as a direct solver, there was no much attention to this algorithm. This changed in the middle of the 70's, where the iterative character of CG was spotted mainly by Paige and Saunders (see Section 1.5.2).

In 1975, the first remarkable variants of CG were developed: MINRES and SYMMLQ by Paige and Saunders [97] and BiCG by Fletcher [35]. Since CG is based on a Lanczos algorithm for generating orthonormal bases of the Krylov spaces and a subsequent LDL^T factorization of the symmetric tridiagonal matrix T, it is potentially unstable if A, and consequently T is indefinite (T is unitarily similar to A), see Section 1.5.2.

MINRES avoids this LDL^T factorization switching to the norm **min**imizing condition for the **res**idual vectors (see Definition 1.2). This yields to a least squares problem which does not depend on the definiteness of A [32].

SYMMLQ solves the projected system with system matrix T via an LQ factorization instead of LDL^T, but does not minimize anything. However, it keeps the residuals orthogonal to all previous ones.

While these two variants retained the Lanczos process and were therefore bound to symmetric matrices A, Fletcher generalized CG by switching to the Bi-Lanczos algorithm in combination with the Petrov-Galerkin condition. This leads to the **bi-c**onjugate **g**radient (BiCG) algorithm, which works for arbitrary square (and nonsingular) matrices A. Unfortunately, BiCG needs matrix-vector products with the transposed system matrix, which is often a problem for large sparse matrices (compare Section 6.1.2). Due to this lack, the development of Bi-Lanczos based algorithms stagnated for several years.

The next two variants, again based on CG, avoiding the assumption of symmetry of A. One obvious trick is to apply CG to the **n**ormal **e**quations (CGNE) [98], i.e., to $A^T A x = A^T b$. While the convergence rate of CG depends on the condition number of the system matrix (see Section 1.3) it now depends on the square of cond(A) and thus might be relatively slow.

Several proposals have been made to improve the numerical stability of this method. The best known is by Paige and Saunders [98] and is based upon applying the Lanczos method to the auxiliary $2n$ by $2n$ system

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

A clever execution of this scheme delivers the LDL^T factorization of the tridiagonal matrix that would have been computed by carrying out the Lanczos procedure with $A^T A$ but without squaring the condition number.

Applying this ideas to MINRES leads to a CG like algorithm applied to the **n**ormal equations and minimizing the **r**esidual norm. The resulting algorithm was called CGNR [98].

Another important extension of the MINRES algorithm, called GMRES (**g**eneralized **m**inimal **res**iduals), was developed in 1986 by Saad and Schulz [115]. They avoided the need of symmetry by interchanging the underlying Lanczos algorithm with the Arnoldi algorithm. The disadvantage of this approach is that it needs increasing time and memory with each iteration due to the Arnoldi method which orthogonalizes every new Krylov basis vector against all previous ones. Several proposals have been made to get this mathematically excellent algorithm computationally more attractive. The best known is the restart technique GMRES(m), restarting GMRES every m iterations with the best approximation computed so far as the new starting vector. Beside this computational penalty, however, it is the only Krylov algorithm for arbitrary matrices with a norm minimizing property of the generated residuals (see Section 1.5.5).

Three years later, Sonnefeld [124] improved the meanwhile 14 years old BiCG algorithm to work without access to the transposed of A. Substantially, this improvement was based on replacing scalar products like $\langle Ap \mid A^T p \rangle$ with $\langle A^2 p \mid p \rangle$ and therefore was called CGS (**c**onjugate **g**radients **s**quared). This squaring can be

interpreted as performing two *minimization* steps at once while computing only one search direction. Sometimes we can observe a doubled convergence rate but since the second 'minimization' step uses the old (and maybe completely wrong) search direction, we often have a quite irregular convergence behavior (see Section 1.5.4).

In 1992, van de Vorst [134] introduced an additional parameter into the Petrov-Galerkin condition and used this parameter to smooth this irregular behavior of the residual norms in CGS. Due to this **stab**ilizing parameter, he called his algorithm BiCGStab, again reminding on the underlying Bi-Lanczos procedure (see Section 1.5.6).

Since norm minimizing of the residuals for arbitrary matrices depends strongly on the Arnoldi algorithm (GMRES), there seemed to be no possibility to develop a GMRES like algorithm with short recurrences. However, in 1991, Freund and Nachtigal [91] managed to bound the residual norm with a product of two norms, where one of them can be minimized even by using a Bi-Lanczos procedure for generating the needed Krylov spaces and the other can be bound independently of $\boldsymbol{A}$. Minimizing only the first of these two norms, they introduced a **q**uasi **m**inimization of the **r**esidual norms (QMR) based on short recurrence formulas (see Section 1.5.6).

Similar to the step from BiCG to CGS, Freund [37] improves this algorithm 1993 to work without transposed matrix-vector products. The resulting procedure was called **t**ranspose **f**ree QMR — TFQMR. Compared with QMR, this algorithm again shows a more irregular behavior in the computed residual norms for the same reason as the CGS algorithm does. Applying the ideas of van de Vorst, Chan *et al.* [18] stabilized TFQMR and developed his so called QMRCGStab algorithm.

1.5.2   Conjugate Gradients (CG)

There are various ways to derive CG. Usually one starts with an obvious steepest descent approach to minimize the function

$$\phi(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{x}^T\boldsymbol{b}.$$

Minimizing ϕ is equivalent to finding the zero of its gradient $\nabla\phi(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}$ if $\boldsymbol{A}$ is positive definite (note: for the second derivate of ϕ we have $\nabla^2\phi \equiv \boldsymbol{A}$).

The resulting algorithm often shows a prohibitively slow convergence rate and heavy oscillating residuals. Modifying this steepest descent algorithm to get conjugate search directions avoids these pitfalls and leads to the Conjugate Gradient algorithm. Using this approach is fairly intuitive and has a good geometrical interpretation. Unfortunately, this interpretation gets lost for most of the other more advanced Krylov algorithms. Therefore we try to give a uniform derivation based on the generating process of the used Krylov spaces (Section 1.2) and based on the conditions to choose the current iterate from the Krylov space (Section 1.1), compare [41]. The main attention thereby is on deriving the important short update formulas.

Suppose $\boldsymbol{A}$ to be positive definite. After m steps of the Lanczos algorithm (compare Section 1.2.2) we obtain the factorization

$$\boldsymbol{A}\boldsymbol{V}_m = \boldsymbol{V}_m\boldsymbol{T}_m + \boldsymbol{v}_{m+1} \cdot h_{m+1,m}\boldsymbol{e}_m^T.$$

Representing $\boldsymbol{x}_m \in \mathcal{V}_m$ as $\boldsymbol{x}_m = \boldsymbol{x}_0 + \boldsymbol{V}_m \boldsymbol{\xi}_m$, the Galerkin condition $\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m \perp \mathcal{V}_m$ now writes as

$$
\begin{aligned}
\boldsymbol{V}_m^T(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m) \;=\; \boldsymbol{V}_m^T(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0 - \boldsymbol{A}\boldsymbol{V}_m\boldsymbol{\xi}_m) \;&=\; 0 \\
\Leftrightarrow \quad \boldsymbol{V}_m^T \boldsymbol{A} \boldsymbol{V}_m \boldsymbol{\xi}_m \;&=\; \boldsymbol{V}_m^T \boldsymbol{r}_0 \\
\Leftrightarrow \quad \boldsymbol{T}_m \boldsymbol{\xi}_m \;&=\; \|\boldsymbol{r}_0\| \boldsymbol{e}_1.
\end{aligned}
$$

With this approach, computing the mth iterated approximation $\boldsymbol{x}_m$ becomes equivalent to solving a positive definite tridiagonal system with system matrix

$$
\boldsymbol{T}_m = \begin{pmatrix}
\alpha_1 & \beta_1 & & & \\
\beta_1 & \alpha_2 & \beta_2 & & \\
& \ddots & \ddots & \ddots & \\
& & \beta_{m-2} & \alpha_{m-1} & \beta_{m-1} \\
& & & \beta_{m-1} & \alpha_m
\end{pmatrix},
$$

which may be solved via the $\boldsymbol{LDL}^T$ factorization. In particular, by setting

$$
\boldsymbol{L}_m = \begin{pmatrix}
1 & 0 & \cdots & \cdots & 0 \\
l_1 & 1 & & & \vdots \\
0 & l_2 & \ddots & & \vdots \\
\vdots & & \ddots & 1 & 0 \\
0 & \cdots & 0 & l_{m-1} & 1
\end{pmatrix}
\quad \text{and} \quad
\boldsymbol{D}_m = \begin{pmatrix}
d_1 & 0 & \cdots & \cdots & 0 \\
0 & d_2 & & & \vdots \\
\vdots & & \ddots & & \vdots \\
\vdots & & & d_{m-1} & 0 \\
0 & \cdots & \cdots & 0 & d_m
\end{pmatrix}
$$

we find by comparing entries in $\boldsymbol{T}_m = \boldsymbol{L}_m \boldsymbol{D}_m \boldsymbol{L}_m^T$ that $d_1 = \alpha_1$, $l_{j-1} = \beta_{j-1}/d_{j-1}$, and $d_j = \alpha_j - \beta_{j-1}l_{j-1}$ for $j = 2, \ldots, m$. Note that since the computation of l_{j-1} and d_j depends solely on d_{j-1}, we can update $(\boldsymbol{L}_m, \boldsymbol{D}_m)$ from $(\boldsymbol{L}_{m-1}, \boldsymbol{D}_{m-1})$ by computing

$$
\begin{aligned}
l_{m-1} &= \beta_{m-1}/d_{m-1} \\
d_m &= \alpha_m - \beta_{m-1}l_{m-1}.
\end{aligned}
$$

Defining $\tilde{\boldsymbol{P}}_m \in \mathbb{R}^{n \times m}$ and $\boldsymbol{y}_m \in \mathbb{R}^m$ by the equations

$$
\tilde{\boldsymbol{P}}_m \boldsymbol{L}_m^T = \boldsymbol{V}_m \quad \text{and} \quad \boldsymbol{L}_m \boldsymbol{D}_m \boldsymbol{y}_m = \boldsymbol{V}_m^T \boldsymbol{r}_0 \tag{1.18}
$$

we get

$$
\begin{aligned}
\boldsymbol{x}_m &= \boldsymbol{x}_0 + \boldsymbol{V}_m \cdot \boldsymbol{T}_m^{-1} \boldsymbol{V}_m^T \boldsymbol{r}_0 = \boldsymbol{x}_0 + \boldsymbol{V}_m (\boldsymbol{L}_m \boldsymbol{D}_m \boldsymbol{L}_m^T)^{-1} \boldsymbol{V}_m^T \boldsymbol{r}_0 \\
&= \boldsymbol{x}_0 + \underbrace{\boldsymbol{V}_m \boldsymbol{L}_m^{-T}}_{\tilde{\boldsymbol{P}}_m} \cdot (\boldsymbol{L}_m \boldsymbol{D}_m)^{-1} \cdot \underbrace{\boldsymbol{V}_m^T \boldsymbol{r}_0}_{\boldsymbol{L}_m \boldsymbol{D}_m \boldsymbol{y}_m} \\
&= \boldsymbol{x}_0 + \tilde{\boldsymbol{P}}_m \boldsymbol{y}_m.
\end{aligned}
$$

Due to the simple structure of $\boldsymbol{L}_m$ and $\boldsymbol{D}_m$ we get the following short update formulas for $\tilde{\boldsymbol{P}}_m$ and $\boldsymbol{y}_m$:

$$
\begin{aligned}
\tilde{\boldsymbol{P}}_m &= (\tilde{\boldsymbol{P}}_{m-1} | \tilde{\boldsymbol{p}}_m) \quad \text{with} \quad \tilde{\boldsymbol{p}}_m = \boldsymbol{v}_m - l_{m-1}\tilde{\boldsymbol{p}}_{m-1} \\
\boldsymbol{y}_m &= (\boldsymbol{y}_{m-1} | y_m)^T \quad \text{with} \quad y_m = (\boldsymbol{v}_m^T \boldsymbol{r}_0 - l_{m-1}d_{m-1}y_{m-1})/d_{m-1}
\end{aligned}
$$

Given $\boldsymbol{x}_0$
$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
$\beta_0 = \|\boldsymbol{r}_0\|_2$
$\boldsymbol{q}_0 = \boldsymbol{0}$
for $m = 1, 2, \ldots$
   $\boldsymbol{v}_m = \boldsymbol{r}_{m-1}/\beta_{m-1}$
   $\alpha_m = \boldsymbol{v}_m^T \boldsymbol{A} \boldsymbol{v}_m$
   $\boldsymbol{r}_m = (\boldsymbol{A} - \alpha_m \boldsymbol{I})\boldsymbol{v}_m - \beta_{m-1}\boldsymbol{v}_{m-1}$
   $\beta_m = \|\boldsymbol{r}_m\|_2$
   if $m = 1$
      $d_1 = \alpha_1$
      $y_1 = \beta_0/\alpha_1$
      $\tilde{\boldsymbol{p}}_1 = \boldsymbol{v}_1$
      $\boldsymbol{x}_1 = y_1 \boldsymbol{v}_1$
   else
      $l_{m-1} = \beta_{m-1}/d_{m-1}$
      $d_m = \alpha_m - \beta_{m-1}l_{m-1}$
      $y_m = -l_{m-1}d_{m-1}y_{m-1}/d_m$
      $\tilde{\boldsymbol{p}}_m = \boldsymbol{v}_m - l_{m-1}\tilde{\boldsymbol{p}}_{m-1}$
      $\boldsymbol{x}_m = \boldsymbol{x}_{m-1} + y_m\tilde{\boldsymbol{p}}_m$

Algorithm 1.5: Conjugate Gradient algorithm derived from a Lanczos process with Petrov condition.

and thus

$$\boldsymbol{x}_m = \boldsymbol{x}_0 + \tilde{\boldsymbol{P}}_m \boldsymbol{y}_m = \boldsymbol{x}_0 + \tilde{\boldsymbol{P}}_{m-1}\boldsymbol{y}_{m-1} + \tilde{\boldsymbol{p}}_m y_m = \boldsymbol{x}_{m-1} + \tilde{\boldsymbol{p}}_m y_m.$$

Doing a lot of algebraic substitutions and transformations (compare [41]) one can prove that Algorithm 1.5 is equivalent to the CG Algorithm 1.6 (with $\boldsymbol{M} = \boldsymbol{I}$). However, we can see that both algorithms require one matrix vector product per iteration ($\boldsymbol{A}\boldsymbol{v}_m$ respectively $\boldsymbol{A}\boldsymbol{p}_m$) and update their approximate solution both with a short recurrence.

1.5.3   Bi-Conjugate Gradients (BiCG)

The Conjugate Gradient method is not suitable for nonsymmetric systems because the residual vectors cannot be made orthogonal with short recurrences (for a proof of this see Faber and Manteuffel [33]). The GMRES method (see Section 1.5.5) retains orthogonality of the residuals by using long recurrences, at the cost of a larger storage demand. The Bi-Conjugate Gradient (BiCG) method takes another approach, replacing the orthogonal sequence of residuals by two mutually orthogonal sequences, at the price of no longer providing a minimization.

BiCG is based on a Bi-Lanczos process and a Petrov-Galerkin condition. That is, two sequences of Krylov subspace basis vectors are generated: $\boldsymbol{V}_m = (\boldsymbol{v}_1 | \cdots | \boldsymbol{v}_m)$ and $\boldsymbol{W}_m = (\boldsymbol{w}_1 | \cdots | \boldsymbol{w}_m)$.

Given $\boldsymbol{x}_0$
$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
for $m = 1, 2, \ldots$
    solve $\boldsymbol{M}\boldsymbol{z}_{m-1} = \boldsymbol{r}_{m-1}$
    $\rho_{m-1} = \langle \boldsymbol{r}_{m-1} \mid \boldsymbol{z}_{m-1} \rangle$
    if $m = 1$
      $\boldsymbol{p}_1 = \boldsymbol{z}_0$
    else
      $\beta_{m-1} = \rho_{m-1}/\rho_{m-2}$
      $\boldsymbol{p}_m = \boldsymbol{z}_{m-1} + \beta_{m-1}\boldsymbol{p}_{m-1}$
    $\boldsymbol{q}_m = \boldsymbol{A}\boldsymbol{p}_m$
    $\alpha_m = \rho_{m-1}/\langle \boldsymbol{p}_m \mid \boldsymbol{q}_m \rangle$
    $\boldsymbol{x}_m = \boldsymbol{x}_{m-1} + \alpha_m \boldsymbol{p}_m$
    $\boldsymbol{r}_m = \boldsymbol{r}_{m-1} - \alpha_m \boldsymbol{q}_m$

Algorithm 1.6: Preconditioned Conjugate Gradient algorithm (CG).

Representing $\boldsymbol{x}_m \in \mathcal{V}_m$ as $\boldsymbol{x}_m = \boldsymbol{x}_0 + \boldsymbol{V}_m \boldsymbol{\xi}_m$, the Petrov-Galerkin condition $\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m \perp \mathcal{W}_m$ now writes as

$$\boldsymbol{W}_m^T(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m) = \boldsymbol{W}_m^T(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0 - \boldsymbol{A}\boldsymbol{V}_m\boldsymbol{\xi}_m) = 0$$
$$\Leftrightarrow \quad \boldsymbol{W}_m^T\boldsymbol{A}\boldsymbol{V}_m\boldsymbol{\xi}_m = \boldsymbol{W}_m^T\boldsymbol{r}_0,$$
$$\Leftrightarrow \quad \boldsymbol{T}_m\boldsymbol{\xi}_m = \|\boldsymbol{r}_0\|\boldsymbol{e}_1$$

Contrary to the Lanczos algorithm, the matrix

$$\boldsymbol{T}_m = \boldsymbol{W}_m^T\boldsymbol{A}\boldsymbol{V}_m = \begin{pmatrix} \alpha_1 & \beta_1 & & & \\ \gamma_1 & \alpha_2 & \beta_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \gamma_{m-2} & \alpha_{m-1} & \beta_{m-1} \\ & & & \gamma_{m-1} & \alpha_m \end{pmatrix}$$

is no more symmetric but still tridiagonal. Thus, instead of $\boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^T$ we have to apply a LU factorization to solve this tridiagonal system:

$$\boldsymbol{L}_m = \begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ l_1 & 1 & & & \vdots \\ 0 & l_2 & \ddots & & \vdots \\ \vdots & & \ddots & 1 & 0 \\ 0 & \cdots & 0 & l_{m-1} & 1 \end{pmatrix} \quad \text{and} \quad \boldsymbol{U}_m = \begin{pmatrix} u_{1,1} & u_{1,2} & 0 & \cdots & & 0 \\ 0 & u_{2,2} & u_{2,3} & & & \vdots \\ \vdots & & \ddots & \ddots & & 0 \\ \vdots & & & & u_{m-1,m-1} & u_{m-1,m} \\ 0 & \cdots & \cdots & & 0 & u_{m,m} \end{pmatrix}$$

Similar to CG, we have simple update formulas for $\boldsymbol{L}_m$ and $\boldsymbol{U}_m$. With $\tilde{\boldsymbol{P}}_m \in \mathbb{R}^{n \times m}$ and $\boldsymbol{y}_m \in \mathbb{R}^m$ defined by

$$\tilde{\boldsymbol{P}}_m\boldsymbol{U}_m = \boldsymbol{V}_m \quad \text{and} \quad \boldsymbol{L}_m\boldsymbol{y}_m = \boldsymbol{V}_m^T\boldsymbol{r}_0 \tag{1.19}$$

we get

$$
\begin{aligned}
\boldsymbol{x}_m &= \boldsymbol{x}_0 + \boldsymbol{V}_m \cdot \boldsymbol{T}_m^{-1} \boldsymbol{V}_m^T \boldsymbol{r}_0 = \boldsymbol{x}_0 + \boldsymbol{V}_m (\boldsymbol{L}_m \boldsymbol{U}_m)^{-1} \boldsymbol{V}_m^T \boldsymbol{r}_0 \\
&= \boldsymbol{x}_0 + \underbrace{\boldsymbol{V}_m \boldsymbol{U}_m^{-1}}_{\tilde{\boldsymbol{P}}_m} \cdot \boldsymbol{L}_m^{-1} \cdot \underbrace{\boldsymbol{V}_m^T \boldsymbol{r}_0}_{\boldsymbol{L}_m \boldsymbol{y}_m} \\
&= \boldsymbol{x}_0 + \tilde{\boldsymbol{P}}_m \boldsymbol{y}_m .
\end{aligned}
$$

Due to the particular structure of $\boldsymbol{L}_m$ and $\boldsymbol{U}_m$ we get the following short update formulas for $\tilde{\boldsymbol{P}}_m$ and $\boldsymbol{y}_m$:

$$
\begin{aligned}
\tilde{\boldsymbol{P}}_m &= (\tilde{\boldsymbol{P}}_{m-1} | \tilde{\boldsymbol{p}}_m) \quad \text{with} \quad \tilde{\boldsymbol{p}}_m = (\boldsymbol{v}_m - u_{m-1,m} \tilde{\boldsymbol{p}}_{m-1})/u_{m,m} \\
\boldsymbol{y}_m &= (\boldsymbol{y}_{m-1} | y_m)^T \quad \text{with} \quad y_m = \boldsymbol{v}_m^T \boldsymbol{r}_0 - l_{m-1} y_{m-1}
\end{aligned}
$$

and thus

$$
\begin{aligned}
\boldsymbol{x}_m &= \boldsymbol{x}_0 + \tilde{\boldsymbol{P}}_m \boldsymbol{y}_m = \boldsymbol{x}_0 + \tilde{\boldsymbol{P}}_{m-1} \boldsymbol{y}_{m-1} + \tilde{\boldsymbol{p}}_m y_m \\
&= \boldsymbol{x}_{m-1} + \tilde{\boldsymbol{p}}_m y_m
\end{aligned}
$$

and

$$
\begin{aligned}
\boldsymbol{r}_m &= \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_{m-1} - \boldsymbol{A}\tilde{\boldsymbol{p}}_m y_m \\
&= \boldsymbol{r}_{m-1} - y_m \boldsymbol{A}\tilde{\boldsymbol{p}}_m .
\end{aligned}
$$

Similar we get the residuals and search directions of the dual problem $\boldsymbol{A}^T \boldsymbol{x}^{dual} = \boldsymbol{b}$:

$$
\begin{aligned}
\tilde{\boldsymbol{p}}_m^{dual} &= \boldsymbol{w}_m - l_m \tilde{\boldsymbol{p}}_{m-1}^{dual} \\
y_m^{dual} &= \boldsymbol{w}_m^T \boldsymbol{r}_0^{dual} - u_{m-1,m}/u_{m,m} \cdot y_{m-1} \\
\boldsymbol{r}_m^{dual} &= \boldsymbol{r}_{m-1}^{dual} - y_m^{dual} \boldsymbol{A}^T \tilde{\boldsymbol{p}}_m^{dual} .
\end{aligned}
$$

Together we have short recurrence formulas for all quantities. Again, we do not present all of the algebraic substitutions (see [89]) but only show the BiCG algorithm as it is usually formulated in Algorithm 1.7.

1.5.4   Conjugate Gradient Squared (CGS)

Let $\mathcal{P}_m$ be the set of polynomials with maximum degree m. Then we can write

$$
\begin{aligned}
\boldsymbol{r}_m &= \phi_m(\boldsymbol{A})\boldsymbol{r}_0 , \quad \boldsymbol{r}_m^{dual} = \phi_m(\boldsymbol{A}^T)\boldsymbol{r}_0^{dual} \\
\boldsymbol{p}_m &= \psi_m(\boldsymbol{A})\boldsymbol{r}_0 , \quad \boldsymbol{p}_m^{dual} = \psi_m(\boldsymbol{A}^T)\boldsymbol{r}_0^{dual}
\end{aligned}
$$

with $\phi_m, \psi_m \in \mathcal{P}_m$. To satisfy the definitions in the BiCG algorithm, we have to define these polynomials via the following recurrence relations (see [89]):

$$
\begin{aligned}
\psi_m(\lambda) &= \phi_m(\lambda) + \beta_{m-1} \psi_{m-1}(\lambda) \\
\phi_{m+1}(\lambda) &= \phi_m(\lambda) - \alpha_m \lambda \psi_m(\lambda)
\end{aligned}
$$

with $\psi_0(\lambda) = \phi_0(\lambda) \equiv 1$. The basic idea of the CGS algorithm is exploiting the fact

$$
\langle \pi(\boldsymbol{A})\boldsymbol{r}_0 \mid \pi(\boldsymbol{A}^T)\boldsymbol{r}_0 \rangle = \langle \pi^2(\boldsymbol{A})\boldsymbol{r}_0 \mid \boldsymbol{r}_0 \rangle ,
$$

Given $\boldsymbol{x}_0$
$\boldsymbol{r}_0 = \boldsymbol{r}_0^{dual} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
$\boldsymbol{p}_1 = \boldsymbol{p}_1^{dual} = \boldsymbol{r}_0$
for $m = 1, 2, \ldots$
    solve $\boldsymbol{M}\boldsymbol{z}_{m-1} = \boldsymbol{r}_{m-1}$
    solve $\boldsymbol{M}^T\boldsymbol{z}_{m-1}^{dual} = \boldsymbol{r}_{m-1}^{dual}$
    $\rho_{m-1} = \langle \boldsymbol{r}_{m-1}^{dual} \mid \boldsymbol{z}_{m-1} \rangle$
    if $\rho_{m-1} = 0$ method fails
    if $m = 1$
        $\boldsymbol{p}_1 = \boldsymbol{z}_0$
        $\boldsymbol{p}_1^{dual} = \boldsymbol{z}_0^{dual}$
    else
        $\beta_{m-1} = \rho_{m-1}/\rho_{m-2}$
        $\boldsymbol{p}_m = \boldsymbol{z}_{m-1} + \beta_{m-1}\boldsymbol{p}_{m-1}$
        $\boldsymbol{p}_m^{dual} = \boldsymbol{z}_{m-1}^{dual} + \beta_{m-1}\boldsymbol{p}_{m-1}^{dual}$
    $\boldsymbol{q}_m = \boldsymbol{A}\boldsymbol{p}_m$
    $\boldsymbol{q}_m^{dual} = \boldsymbol{A}^T\boldsymbol{p}_m^{dual}$
    $\alpha_m = \rho_{m-1}/\langle \boldsymbol{p}_m^{dual} \mid \boldsymbol{q}_m \rangle$
    $\boldsymbol{x}_m = \boldsymbol{x}_{m-1} + \alpha_m\boldsymbol{p}_m$
    $\boldsymbol{r}_m = \boldsymbol{r}_{m-1} - \alpha_m\boldsymbol{q}_m$
    $\boldsymbol{r}_m^{dual} = \boldsymbol{r}_{m-1}^{dual} - \alpha_m\boldsymbol{q}_m^{dual}$

Algorithm 1.7: Preconditioned Bi-Conjugate Gradient algorithm (BiCG).

which holds for all polynomials $\pi \in \mathcal{P}_m$.

Defining $\hat{\boldsymbol{r}}_m := \phi_m^2(\boldsymbol{A})\boldsymbol{r}_0$ and $\hat{\boldsymbol{p}}_m := \psi_m^2(\boldsymbol{A})\boldsymbol{r}_0$ yields

$$
\begin{aligned}
\alpha_m &= \frac{\langle \phi_m(\boldsymbol{A})\boldsymbol{r}_0 \mid \phi_m(\boldsymbol{A}^T)\boldsymbol{r}_0 \rangle}{\langle \boldsymbol{A}\psi_m(\boldsymbol{A})\boldsymbol{r}_0 \mid \psi_m(\boldsymbol{A}^T)\boldsymbol{r}_0 \rangle} = \frac{\langle \phi_m^2(\boldsymbol{A})\boldsymbol{r}_0 \mid \boldsymbol{r}_0 \rangle}{\langle \boldsymbol{A}\psi_m^2(\boldsymbol{A})\boldsymbol{r}_0 \mid \boldsymbol{r}_0 \rangle} \\
&= \frac{\langle \hat{\boldsymbol{r}}_m \mid \boldsymbol{r}_0 \rangle}{\langle \boldsymbol{A}\hat{\boldsymbol{p}}_m \mid \boldsymbol{r}_0 \rangle}
\end{aligned}
$$

and

$$
\beta_m = \frac{\langle \phi_{m+1}(\boldsymbol{A})\boldsymbol{r}_0 \mid \phi_{m+1}(\boldsymbol{A}^T)\boldsymbol{r}_0 \rangle}{\langle \phi_m(\boldsymbol{A})\boldsymbol{r}_0 \mid \phi_m(\boldsymbol{A}^T)\boldsymbol{r}_0 \rangle} = \frac{\langle \hat{\boldsymbol{r}}_{m+1} \mid \boldsymbol{r}_0 \rangle}{\langle \hat{\boldsymbol{r}}_m \mid \boldsymbol{r}_0 \rangle}
$$

Applying the recurrence relations we get

$$
\psi_m^2(\lambda) = \phi_m^2(\lambda) + 2\beta_{m-1}\phi_m(\lambda)\psi_{m-1}(\lambda) + \beta_{m-1}^2\psi_{m-1}^2(\lambda),
$$
$$
\phi_{m+1}^2(\lambda) = \phi_m^2(\lambda) - \alpha_m\lambda\big(2\phi_m^2(\lambda) + 2\beta_{m-1}\phi_m(\lambda)\psi_{m-1}(\lambda) - \alpha_m\lambda\psi_{m-1}^2(\lambda)\big), \text{ and}
$$
$$
\phi_{m+1}(\lambda)\psi_m(\lambda) = \phi_m^2(\lambda) + \beta_{m-1}\phi_m(\lambda)\psi_{m-1}(\lambda) - \alpha_m\lambda\psi_{m-1}^2(\lambda).
$$

Now we introduce two auxiliary vectors $\hat{\boldsymbol{q}}_m$ and $\hat{\boldsymbol{u}}_m$ as

$$
\hat{\boldsymbol{q}}_m := \phi_{m+1}(\boldsymbol{A})\psi_m(\boldsymbol{A})\boldsymbol{r}_0
$$

and

$$\begin{aligned}
\hat{\boldsymbol{u}}_m &:= \phi_m(\boldsymbol{A})\psi_m(\boldsymbol{A})\boldsymbol{r}_0 \\
&= \phi_m^2(\boldsymbol{A})\boldsymbol{r}_0 + \beta_{m-1}\phi_m(\boldsymbol{A})\psi_{m-1}(\boldsymbol{A})\boldsymbol{r}_0 \\
&= \hat{\boldsymbol{r}}_m + \beta_{m-1}\hat{\boldsymbol{q}}_{m-1}.
\end{aligned}$$

This yields

$$\begin{aligned}
\hat{\boldsymbol{q}}_m &:= \phi_{m+1}^2(\boldsymbol{A})\boldsymbol{r}_0 + \beta_{m-1}\phi_m(\boldsymbol{A})\psi_{m-1}(\boldsymbol{A})\boldsymbol{r}_0 - \alpha_m\boldsymbol{A}\psi_{m-1}^2(\boldsymbol{A}) \\
&= \underbrace{\hat{\boldsymbol{r}}_m + \beta_{m-1}\hat{\boldsymbol{q}}_{m-1}}_{\hat{\boldsymbol{u}}_m} - \alpha_m\boldsymbol{A}\hat{\boldsymbol{p}}_m.
\end{aligned}$$

Using these auxiliary vectors we obtain the desired short recurrences for $\hat{\boldsymbol{r}}_{m+1}$ and $\hat{\boldsymbol{p}}_{m+1}$:

$$\begin{aligned}
\hat{\boldsymbol{r}}_{m+1} &= \phi_{m+1}^2(\boldsymbol{A})\boldsymbol{r}_0 \\
&= \phi_m^2(\boldsymbol{A}) - \alpha_m\boldsymbol{A}\big(2\phi_m^2(\boldsymbol{A}) + 2\beta_{m-1}\phi_m(\boldsymbol{A})\psi_{m-1}(\boldsymbol{A}) - \alpha_m\boldsymbol{A}\psi_{m-1}^2(\boldsymbol{A})\big) \\
&= \hat{\boldsymbol{r}}_m - \alpha_m\boldsymbol{A}\big(\underbrace{\hat{\boldsymbol{r}}_m + \beta_{m-1}\hat{\boldsymbol{q}}_{m-1}}_{\hat{\boldsymbol{u}}_m} + \underbrace{\hat{\boldsymbol{r}}_m + \beta_{m-1}\hat{\boldsymbol{q}}_{m-1} - \alpha_m\boldsymbol{A}\hat{\boldsymbol{p}}_m}_{\hat{\boldsymbol{q}}_m}\big)
\end{aligned}$$

and

$$\begin{aligned}
\hat{\boldsymbol{p}}_{m+1} &= \psi_{m+1}^2(\boldsymbol{A})\boldsymbol{r}_0 \\
&= \phi_{m+1}^2(\boldsymbol{A}) + 2\beta_m\phi_{m+1}(\boldsymbol{A})\psi_m(\boldsymbol{A}) + \beta_m^2\psi_m^2(\boldsymbol{A}), \\
&= \hat{\boldsymbol{r}}_{m+1} + 2\beta_m\hat{\boldsymbol{q}}_m + \beta_m^2\hat{\boldsymbol{p}}_m.
\end{aligned}$$

Omitting the hats ($\hat{}$) and doing some algebraic reformulation to save memory and computing time, we get the CGS algorithm as shown in Algorithm 1.8.

1.5.5   Generalized Minimal Residuals (GMRES)

GMRES is based on the Arnoldi algorithm to generate $\mathcal{V}_m$ and a Petrov-Galerkin condition with $\mathcal{W}_m = \boldsymbol{A}\mathcal{V}_m$ to choose $\boldsymbol{x}_m \in \mathcal{V}_m$. That is $\boldsymbol{x}_m = \boldsymbol{x}_0 + \boldsymbol{V}_m\boldsymbol{\xi}_m$ has to satisfy

$$\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m \perp \boldsymbol{W}_m = \boldsymbol{A}\boldsymbol{V}_m. \tag{1.20}$$

With this particular $\mathcal{W}_m$, the Petrov-Galerkin condition is equivalent to a norm minimizing condition. Therefore, $\boldsymbol{\xi}_m \in \mathbb{R}^m$ minimizes the function

$$\phi_m(\boldsymbol{\xi}_m) := \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m\|_2 = \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0 - \boldsymbol{A}\boldsymbol{V}_m\boldsymbol{\xi}_m\|_2 = \|\boldsymbol{r}_0 - \boldsymbol{A}\boldsymbol{V}_m\boldsymbol{\xi}_m\|_2$$

if and only if $\boldsymbol{x}_m = \boldsymbol{x}_0 + \boldsymbol{V}_m\boldsymbol{\xi}_m$ satisfies (1.20).

With the Arnoldi factorization (equation (1.7) on page 11) we get

$$\begin{aligned}
\phi_m(\boldsymbol{\xi}_m) &= \|\boldsymbol{r}_0 - \boldsymbol{A}\boldsymbol{V}_m\boldsymbol{\xi}_m\|_2 \\
&= \|\,\|\boldsymbol{r}_0\|_2\boldsymbol{v}_1 - (\boldsymbol{V}_m\boldsymbol{H}_m + h_{m+1,m}\cdot\boldsymbol{v}_{m+1}\boldsymbol{e}_m^T)\boldsymbol{\xi}_m\|_2 \\
&= \|\,\|\boldsymbol{r}_0\|_2\boldsymbol{V}_{m+1}\boldsymbol{e}_1 - \boldsymbol{V}_{m+1}\tilde{\boldsymbol{H}}_m\boldsymbol{\xi}_m\|_2 \tag{1.21} \\
&= \|\boldsymbol{V}_{m+1}(\|\boldsymbol{r}_0\|_2\boldsymbol{e}_1 - \tilde{\boldsymbol{H}}_m\boldsymbol{\xi}_m)\|_2 \\
&= \|\,\|\boldsymbol{r}_0\|_2\boldsymbol{e}_1 - \tilde{\boldsymbol{H}}_m\boldsymbol{\xi}_m\|_2 \tag{1.22}
\end{aligned}$$

Given $\boldsymbol{x}_0$
$\boldsymbol{r}_0 = \boldsymbol{r}_0^{dual} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
choose $\tilde{\boldsymbol{r}}$, for example $\tilde{\boldsymbol{r}} = \boldsymbol{r}_0$
for $m = 1, 2, \ldots$
$\quad \rho_{m-1} = \langle \tilde{\boldsymbol{r}} \mid \boldsymbol{r}_{m-1} \rangle$
$\quad$ **if** $\rho_{m-1} = 0$ method fails
$\quad$ **if** $m = 1$
$\quad\quad \boldsymbol{u}_1 = \boldsymbol{r}_0$
$\quad\quad \boldsymbol{p}_1 = \boldsymbol{u}_1$
$\quad$ **else**
$\quad\quad \beta_{m-1} = \rho_{m-1}/\rho_{m-2}$
$\quad\quad \boldsymbol{u}_m = \boldsymbol{r}_{m-1} + \beta_{m-1}\boldsymbol{q}_{m-1}$
$\quad\quad \boldsymbol{p}_m = \boldsymbol{u}_m + \beta_{m-1}(\boldsymbol{q}_{m-1} + \beta_{m-1}\boldsymbol{p}_{m-1})$
$\quad$ solve $\boldsymbol{M}\boldsymbol{z}_m = \boldsymbol{p}_m$
$\quad \hat{\boldsymbol{z}}_m = \boldsymbol{A}\boldsymbol{z}_m$
$\quad \alpha_m = \rho_{m-1}/\langle \hat{\boldsymbol{z}}_m \mid \tilde{\boldsymbol{r}} \rangle$
$\quad \boldsymbol{q}_m = \boldsymbol{u}_m - \alpha_m \hat{\boldsymbol{z}}_m \quad$ solve $\boldsymbol{M}\boldsymbol{z}_m = \boldsymbol{u}_m + \boldsymbol{q}_m$
$\quad \boldsymbol{x}_m = \boldsymbol{x}_{m-1} + \alpha_m \boldsymbol{z}_m$
$\quad \hat{\boldsymbol{z}}_m = \boldsymbol{A}\boldsymbol{z}_m$
$\quad \boldsymbol{r}_m = \boldsymbol{r}_{m-1} - \alpha_m \hat{\boldsymbol{z}}_m$

Algorithm 1.8: Preconditioned Conjugate Gradient Squared algorithm (CGS).

where

$$\tilde{\boldsymbol{H}}_m = \begin{pmatrix} \boldsymbol{H}_m \\ \boldsymbol{e}_m^T \cdot h_{m+1,m} \end{pmatrix} = \left(\begin{array}{ccccc} h_{1,1} & h_{1,2} & \cdots & h_{1,m-1} & h_{1,m} \\ h_{2,1} & h_{2,2} & & \vdots & \vdots \\ & h_{3,2} & & \vdots & \vdots \\ & & \ddots & \vdots & \vdots \\ & & & h_{m,m-1} & h_{m,m} \\ \hline 0 & \cdots & & 0 & h_{m+1,m} \end{array} \right) \in \mathrm{I\!R}^{(m+1)\times m}$$

Thus, minimizing ϕ_m is equivalent to solving the least squares problem

$$\min_{\boldsymbol{\xi}_m \in \mathrm{I\!R}^m} \| \, \|\boldsymbol{r}_0\|_2 \boldsymbol{e}_1 - \tilde{\boldsymbol{H}}_m \boldsymbol{\xi}_m \|_2.$$

A standard method for solving least squares problems is to factorize the $m + 1$ by m matrix $\tilde{\boldsymbol{H}}_m$ into an upper triangular $m + 1$ by m matrix $\boldsymbol{R}_m$ (with last row $\boldsymbol{0}$) and an $m + 1$ by $m + 1$ unitary matrix $\boldsymbol{Q}_m$. Exploiting the special structure of $\tilde{\boldsymbol{H}}_m$ this generalized QR factorization can efficiently be computed with Givens rotations. The solution $\boldsymbol{\xi}_m$ is then obtained by solving the upper triangular m by m subsystem of

$$\boldsymbol{R}_m \boldsymbol{\xi}_m = \|\boldsymbol{r}_0\|_2 \boldsymbol{Q}_m \boldsymbol{e}_1. \tag{1.23}$$

Suppose we have such a generalized QR factorization of $\tilde{\boldsymbol{H}}_m$, i.e., we have m

Givens rotations $\boldsymbol{G}_i$ with

$$\underbrace{\boldsymbol{G}_m \boldsymbol{G}_{m-1} \cdots \boldsymbol{G}_1}_{\boldsymbol{Q}_m} \tilde{\boldsymbol{H}}_m = \boldsymbol{R}_m = \begin{pmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,m} \\ 0 & r_{2,2} & & \vdots \\ & \ddots & \ddots & \vdots \\ & & \ddots & r_{m,m} \\ & & & 0 \end{pmatrix}.$$

Applying $\boldsymbol{Q}_m$ in the next step to $\tilde{\boldsymbol{H}}_{m+1}$ yields

$$\boldsymbol{G}_m \boldsymbol{G}_{m-1} \cdots \boldsymbol{G}_1 \cdot \tilde{\boldsymbol{H}}_{m+1} = \left(\begin{array}{cccc|c} r_{1,1} & r_{1,2} & \cdots & r_{1,m} & r_{1,m+1} \\ 0 & r_{2,2} & & \vdots & \vdots \\ & \ddots & \ddots & \vdots & \vdots \\ & & \ddots & r_{m,m} & r_{m,m+1} \\ & & & 0 & \tilde{r}_{m+1,m+1} \\ \hline 0 & & \cdots & 0 & \tilde{r}_{m+2,m+1} \end{array} \right).$$

Thus we just have to apply one more Givens rotation $\boldsymbol{G}_{m+1}$ to eliminate $\tilde{r}_{m+2,m+1}$:

$$\boldsymbol{G}_{m+1} := \begin{pmatrix} \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0}^T & \tilde{\boldsymbol{G}}_{m+1} \end{pmatrix} \quad \text{with} \quad \tilde{\boldsymbol{G}}_{m+1} \begin{pmatrix} \tilde{r}_{m+1,m+1} \\ \tilde{r}_{m+2,m+1} \end{pmatrix} = \begin{pmatrix} r_{m+1,m+1} \\ 0 \end{pmatrix}$$

Collecting our results we can update $\boldsymbol{R}_{m+1}$ from $\boldsymbol{R}_m$ by applying $\boldsymbol{Q}_m$ to the last column of $\tilde{\boldsymbol{H}}_{m+1}$, computing $\boldsymbol{G}_{m+1}$, and setting

$$r_{m+1,m+1} := \begin{cases} \text{sign}(r_{m+1,m+1}) \sqrt{\tilde{r}_{m+1,m+1}^2 + \tilde{r}_{m+2,m+1}^2} & \text{if } \tilde{r}_{m+1,m+1} \neq 0 \\ \tilde{r}_{m+2,m+1} & \text{if } \tilde{r}_{m+1,m+1} = 0 \end{cases}$$

and

$$r_{m+2,m+1} := 0.$$

The right hand side vector in (1.23) is computed by applying all Givens rotations to the first unit vector $\boldsymbol{e}_1$. For the residual norm, we have

$$\|\boldsymbol{r}_m\|_2 = \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m\|_2 = \| \, \|\boldsymbol{r}_0\|_2 \boldsymbol{e}_1 - \boldsymbol{Q}_m \boldsymbol{R}_m \boldsymbol{\xi}_m\|_2 = \| \, \|\boldsymbol{r}_0\|_2 \boldsymbol{Q}_m^T \boldsymbol{e}_1 - \boldsymbol{R}_m \boldsymbol{\xi}_m\|_2.$$

The first m elements of $\|\boldsymbol{r}_0\|_2 \boldsymbol{Q}_m^T \boldsymbol{e}_1 - \boldsymbol{R}_m \boldsymbol{\xi}_m$ are 0, because the m by m upper triangular subsystem in (1.23) is nonsingular and therefore the least squares problem is solved with error 0. Therefore, $\|\boldsymbol{r}_0\|_2 \boldsymbol{Q}_m^T \boldsymbol{e}_1 - \boldsymbol{R}_m \boldsymbol{\xi}_m = (0, \ldots, 0, \|\boldsymbol{r}_0\|_2 \cdot \boldsymbol{e}_{m+1}^T \boldsymbol{Q}_m^T \boldsymbol{e}_1)^T$ and thus $\|\boldsymbol{r}_m\|_2 = \|\boldsymbol{r}_0\|_2 \cdot |\boldsymbol{e}_1^T \boldsymbol{Q}_m \boldsymbol{e}_{m+1}|$.

This enables us to check the residual norm quickly and only to compute the approximate solution $\boldsymbol{x}_m$ if $\|\boldsymbol{r}_m\|_2$ is sufficiently small.

Since we have to store all Krylov basis vectors anyway, it might be advantageous to replace the Gram-Schmidt orthogonalization by the robust Householder process [140]. However, Rozložník showed, that GMRES with modified Gram-Schmidt is backward stable [50, 105] and thus the higher computational effort is not necessary.

Given $\boldsymbol{x}_0$
for $j = 1, 2, \ldots$
    $\tilde{\boldsymbol{v}}_1 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
    solve $\boldsymbol{M}\boldsymbol{z}_1 = \tilde{\boldsymbol{v}}_1$
    $\boldsymbol{v}_1 = \boldsymbol{z}_1/\|\boldsymbol{z}_1\|$
    $\boldsymbol{s}_1 = \|\boldsymbol{z}_1\|\boldsymbol{e}_1$
    for $m = 1, \ldots, M$
        $\tilde{\boldsymbol{v}}_{m+1} = \boldsymbol{A}\boldsymbol{v}_m$
        solve $\boldsymbol{M}\boldsymbol{z}_{m+1} = \tilde{\boldsymbol{v}}_{m+1}$
        for $k = 1, \ldots, m$
            $h_{k,m} = \langle \boldsymbol{z}_{m+1} \mid \boldsymbol{v}_k \rangle$
            $\boldsymbol{z}_{m+1} = \boldsymbol{z}_{m+1} - h_{k,m}\boldsymbol{v}_k$
        $h_{m+1,m} = \|\boldsymbol{z}_{m+1}\|$
        $\boldsymbol{v}_{m+1} = \boldsymbol{z}_{m+1}/h_{m+1,m}$
        apply $\boldsymbol{G}_1, \ldots, \boldsymbol{G}_{m-1}$ to $(h_{1,m} \ldots h_{m+1,m})^T$
        compute $\boldsymbol{G}_m$
        $\boldsymbol{s}_{m+1} = \boldsymbol{G}_m\boldsymbol{s}_m$
        if $s_{m+1,m+1}$ is small enough
            compute $\boldsymbol{x}_{m+1}$
            quit
    compute $\boldsymbol{x}_{m+1}$
    if convergence
        quit
    else
        $\boldsymbol{x}_0 = \boldsymbol{x}_{m+1}$

Algorithm 1.9: Preconditioned Generalized Minimal Residual algorithm with restart (GMRES(m)).

1.5.6 Stabilized Variants and Quasi Minimization

● *Smoothing Residual Norms*

Inspecting the run of the curve of CGS residual norms, we often observe an irregular, oscillating behavior. This is due to the fact that CGS does two 'minimization' steps at once, but computes only once the search direction. Therefore CGS sometimes overshoots locally. To smooth these oscillations, van de Vorst [133, 134] coupled CGS with a repeatedly applied GMRES(1) iteration. This leads to a local minimization of the residual norms and therefore to a considerably smoother convergence behavior. Chan *et. al.* applied this idea to TFQMR and developed the so called QMRCGStab algorithm (see below).

● *Quasi Minimization*

There are various quasi minimization methods which try to retain the advantages of GMRES but only need short recurrences from Bi-Lanczos like algorithms. The most important are QMR (**Q**uasi **M**inimal **R**esiduals) which combines BiCG with

GMRES, TFQMR (**T**ranspose **F**ree QMR) which is based on CGS and GMRES, and QMRCGStab (BiCGStab with GMRES).

We illustrate only the basic ideas of quasi minimization at the example of QMR. For further references see [18, 37, 91].

As in BiCG the mth iterate has the form $\boldsymbol{x}_m = \boldsymbol{x}_0 + \boldsymbol{V}_m \boldsymbol{\xi}_m$ and we have

$$\boldsymbol{A}\boldsymbol{V}_m = \boldsymbol{V}_{m+1}\tilde{\boldsymbol{T}}_m$$

where $\tilde{\boldsymbol{T}}_m \in \mathrm{I\!R}^{m+1 \times m}$ is tridiagonal. It follows that if $\boldsymbol{v}_1 = \boldsymbol{r}_0/\|\boldsymbol{r}_0\|_2$, then

$$
\begin{aligned}
\phi_m(\boldsymbol{\xi}_m) &:= \|\boldsymbol{b} - \boldsymbol{A}(\boldsymbol{x}_0 + \boldsymbol{V}_m\boldsymbol{\xi}_m)\|_2 \\
&= \|\boldsymbol{r}_0 - \boldsymbol{A}\boldsymbol{V}_m\boldsymbol{\xi}_m\|_2 \\
&= \|\boldsymbol{r}_0 - \boldsymbol{V}_{m+1}\tilde{\boldsymbol{T}}_m\boldsymbol{\xi}_m\|_2 \\
&= \|\boldsymbol{V}_{m+1}(\|\boldsymbol{r}_0\|_2\boldsymbol{e}_1 - \tilde{\boldsymbol{T}}_m\boldsymbol{\xi}_m)\|_2.
\end{aligned}
$$

Unfortunately, $\boldsymbol{V}_{m+1}$ is not orthogonal and thus we cannot conclude $\phi_m(\boldsymbol{\xi}_m) = \|\|\boldsymbol{r}_0\|_2\boldsymbol{e}_1 - \tilde{\boldsymbol{T}}_m\boldsymbol{\xi}_m\|_2$ as we did in GMRES. However, introducing the scaling matrix $\boldsymbol{S}_{m+1} = \mathrm{diag}(\boldsymbol{V}_{m+1})$ we have

$$\phi_m(\boldsymbol{\xi}_m) \leq \|\boldsymbol{V}_{m+1}\boldsymbol{S}_{m+1}^{-1}\|_2 \cdot \|\boldsymbol{S}_{m+1}(\|\boldsymbol{r}_0\|_2\boldsymbol{e}_1 - \tilde{\boldsymbol{T}}_m\boldsymbol{\xi}_m)\|_2 \qquad (1.24)$$

with $\|\boldsymbol{V}_{m+1}\boldsymbol{S}_{m+1}^{-1}\|_2 \leq \sqrt{m+1}$. In the QMR algorithm we simply neglect this first factor in (1.24) and only minimize the second.

Applying these ideas to CGS/BiCGStab leads to the TFQMR respectively QMR-CGStab algorithm.

2

Krylov Methods and Floating-Point Arithmetic

> ❝ *Accuracy and precision are the same for the scalar computation* $c = a * b$, *but accuracy can be much worse than precision in the solution of a linear system of equations, for example.* ❞
>
> Nicholas J. Higham, 1996

> ❝ *It makes me nervous to fly on airplanes since I know they are designed using floating-point arithmetic* ❞
>
> Alston S. Householder

Preconditioned Krylov subspace solvers are frequently used for solving large sparse linear systems. There are many advantageous properties concerning convergence rates and error estimates but unfortunately, if we implement such a solver on a computer, we often observe an unexpected and even contrary behavior (see e.g. [48, 65, 128]). The basic reason for this is that standard computer arithmetic is based on floating-point numbers, i.e., numbers with a finite precision. In Section 2.1 we give a short overview about floating-point arithmetics, Section 2.3.1 describes possible failures during preconditioning and Section 2.3.2 gives some deeper insight into the effect of finite precision arithmetic to Krylov methods.

2.1   Floating-Point Arithmetics
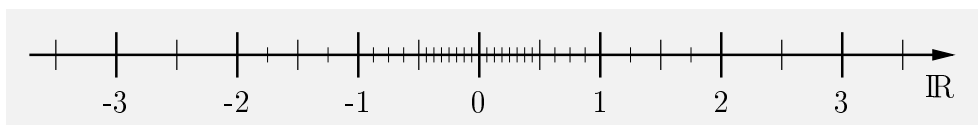
2.1.1   Floating-Point Numbers

Nearly all important computer architectures used today are providing so called floating-point numbers for numerical computations. The floating-point number space $\mathcal{F} = \mathcal{F}(b, l, e_{\min}, e_{\max})$ is a finite subset of $\mathbb{R}$, characterized by its *base b*, the *length* of its mantissa l, the *minimal exponent* $e_{\min}$, and *maximal exponent* $e_{\max}$.

A floating-point number itself consists of three parts: a *sign s* ($+$ or $-$), a fixed length *mantissa* $0.m_1 m_2 \ldots m_l$ with digits $m_i \in \{1, \ldots, b-1\}$, and an *exponent e* with $e_{\min} \le e \le e_{\max}$, given to the base b. These define the floating-point number $s\, m \cdot b^e$. To get unique representations, the first mantissa digit m_1 is required to be nonzero (otherwise, e.g., we had $0.01b^e = 0.10b^{e-1}$). Numbers with this property are called *normalized*. For $e = e_{\min}$ it is not necessary to require $m_1 \neq 0$ in order to retain the uniqueness of the representation. That is, we may allow non-normalized numbers[1] between 0 and $\pm 0.1b^{e_{\min}}$.

Table 2.1 and Figure 2.1 illustrate the floating-point system $\mathcal{F}(2, 3, -1, 2)$.

		normalized mantissas			
		0.100_2	0.101_2	0.110_2	0.111_2
exponent	3	2.0	2.5	3.0	3.5
	2	1.0	1.25	1.5	1.75
	0	0.5	0.625	0.75	0.875
	-1	0.25	0.3125	0.375	0.4375
		denormalized mantissas			
		0.000_2	0.001_2	0.010_2	0.011_2
	-1	0.0	0.0625	0.125	0.1875

Table 2.1: The floating-point system $\mathcal{F}(2, 3, -1, 2)$ with denormalized numbers

Figure 2.1: The floating-point system $\mathcal{F}(2, 3, -1, 2)$ with denormalized numbers

The two most common floating-point systems are defined in the IEEE-754 standard [4]. The first is called 'single precision' which is a $\mathcal{F}(2, 24, -125, 128)$ system and the second is called 'double precision' which is $\mathcal{F}(2, 53, -1021, 1024)^2$. Both systems provide the special numbers -0, $\pm\texttt{inf}$ (*infinity*), and $\texttt{nan}$ (*not-a-number*).

---

[1]Having non-normalized numbers also ensures the existence of unique additive inverses in $\mathcal{F}$.

[2]Note that this definition differs a little from standard due to our definition of normalized numbers.

2.1.2   Roundings

Obviously, $(\mathcal{F}, \circ)$ with $\circ \in \{+, -, *, /\}$ is not closed, i.e., with $a, b \in \mathcal{F}$ we generally don't have $a \circ b \in \mathcal{F}$. Therefore we need a *rounding* after each operation, to map $a \circ b$ back into the floating-point screen $\mathcal{F}$.

Usually, one is interested to obtain the nearest number out of $\mathcal{F}$. In this case we define $\Box(a \circ b) =: x \in \mathcal{F}$ with[3] $|x - (a \circ b)| = \min_{\xi \in \mathcal{F}}\{|\xi - (a \circ b)|\}$. Unfortunately, with this rounding we neither know the exact size nor the sign of the error we made by substituting the mathematically correct $a \circ b$ with the computer representable $\Box(a \circ b)$. However, at least for the relative error we have

$$\frac{|\Box(a \circ b) - a \circ b|}{|a \circ b|} \leq \frac{b^{-l-1}}{2} := \epsilon/2 \quad \text{if } a \circ b \neq 0.$$

This ϵ is called *machine precision*. It is the smallest possible relative error in $\mathcal{F}$, i.e., $(1 - \min_{x \in \mathcal{F}}\{x > 1\})$.

To retain at least the sign of the error, we may use directed roundings:

$$\triangledown(a \circ b) = \max_{\xi \in \mathcal{F}}\{\xi \leq a \circ b\}, \qquad \triangle(a \circ b) = \min_{\xi \in \mathcal{F}}\{\xi \geq a \circ b\}$$
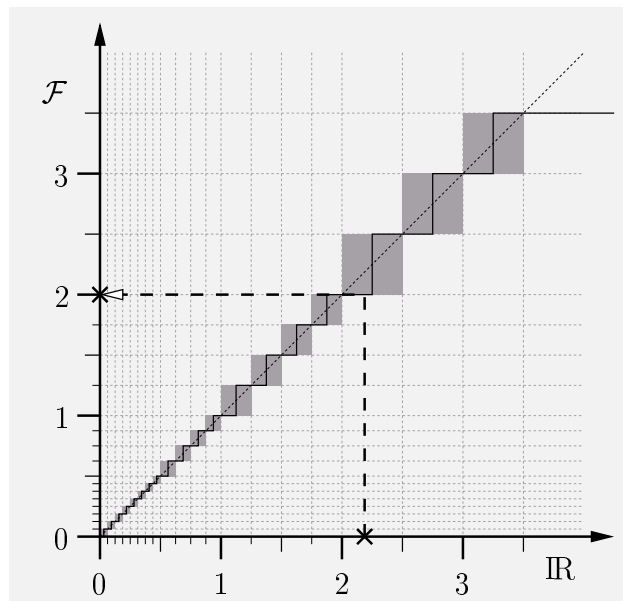
Figure 2.2: Round to nearest in $\mathcal{F}(2, 3, -1, 2)$ with denormalized numbers

Figure 2.2 illustrates 'round to nearest' in our example floating-point screen $\mathcal{F}(2, 3, -1, 2)$. Since a rounding can only take values in $\mathcal{F}$, it is completely defined by its saltus in each of the grey boxes ([75]). That is the smallest possible rounding is $\triangledown$, with the saltus on the right edge in each box and $\triangle$ is the largest one, with the saltus on the left edge.

---

[3]If $a \circ b$ lies exactly in the middle between two floating-point numbers, then the one with even mantissa, i.e. with $m_l = 0$, is chosen ('round to nearest even').

2.2   Finite Precision Behavior of Lanczos Procedures

Rounding errors greatly affect the behavior of the Lanczos iteration [12, 64, 122, 143]. The basic difficulty with finite precision arithmetic is the loss of orthogonality among the generated basis vectors of the Krylov subspaces. The central results about Lanczos error analysis are mostly based on the fundamental and excellent work of Paige, see e.g. [93, 94].

Implementing the Lanczos procedure with modified Gram-Schmidt orthogonalization in finite precision arithmetic we have to take rounding errors into consideration. That is, each value, computed in finite precision, may differ from the exact one and thus would need a different notation. However, for simplicity we retain the notation unchanged in this section and explicitly point out where we use exact quantities.

Paige shows that in finite precision equation (1.11) on page 13 has to be extended to

$$\boldsymbol{A V}_m = \boldsymbol{V}_m \boldsymbol{T}_m + \boldsymbol{v}_{m+1} \beta_m \boldsymbol{e}_m^T + \boldsymbol{F}_m, \qquad (2.1)$$

where $\boldsymbol{F}_m$ contains the rounding errors.

We define

$$\epsilon_0 := 2(n+4)\epsilon \quad \text{and} \quad \epsilon_1 := 2(7 + nnz \cdot \| \, |\boldsymbol{A}| \, \|_2 / \|\boldsymbol{A}\|_2)\epsilon$$

where ϵ denotes the machine precision and nnz is the maximum number of nonzero elements in any row of $\boldsymbol{A}$. Under the assumptions that

$$\epsilon_0 < \frac{1}{12}, \qquad m(3\epsilon_0 + \epsilon_1) < 1,$$

and ignoring higher order terms in ϵ, Paige shows that

$$\|\boldsymbol{F}_m\|_2 \le \sqrt{m}\epsilon_1 \|\boldsymbol{A}\|_2$$

or, a little bit more sloppy, $\|\boldsymbol{F}_m\|_2$ is approximately of size $\epsilon\|\boldsymbol{A}\|_2$ [100, 119]. This means, the tri-diagonalization holds up to machine precision. Unfortunately, the situation becomes much more difficult concerning the orthogonality condition. This fact is stated in the following theorem.

Theorem 2.1 *Suppose* $(\boldsymbol{S}_m, \boldsymbol{\Lambda}_m)$ *with* $\boldsymbol{\Lambda}_m := \text{diag}(\lambda_1, \ldots, \lambda_m)$ *is the exact spectral factorization of the computed* $\boldsymbol{T}_m$ *(see equation. (2.1)), i.e., the* $(\boldsymbol{z}_j, \lambda_j)$ *are the computed Ritz-pairs of* $\boldsymbol{A}$ *if we define* $\boldsymbol{z}_j := \boldsymbol{V}_m \boldsymbol{s}_j$. *Then, we have*

$$\angle(\boldsymbol{z}_j, \boldsymbol{v}_{m+1}) \approx Arccos\left(\frac{\epsilon\|\boldsymbol{A}\|_2}{| \, \|\boldsymbol{A}\boldsymbol{z}_j - \lambda_j \boldsymbol{z}_j\|_2 - \epsilon\|\boldsymbol{A}\|_2 \, |} \right).$$

Notice that $\boldsymbol{v}_{m+1}$ should stay orthogonal to $\mathcal{K}_m = \text{span}(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m)$ and $\boldsymbol{z}_j \in \mathcal{K}_m$, because it is a linear combination of the columns of $\boldsymbol{V}_m$ — the basis vectors. So $\boldsymbol{v}_{m+1}$ should stay orthogonal to $\boldsymbol{z}_j$, too. If $(\boldsymbol{z}_j, \lambda_j)$ is a good approximation to any eigenpair of $\boldsymbol{A}$, i.e., $\|\boldsymbol{A}\boldsymbol{z}_j - \lambda_j \boldsymbol{z}_j\|_2$ is small, then $\angle(\boldsymbol{z}_j, \boldsymbol{v}_{m+1}) \approx \text{Arccos}(1) = 0$. This means, ironically, if a Ritz-pair is converging (and this is exactly what we want in eigenvalue computations) we lose our orthogonality in $\boldsymbol{V}_m$ completely. Nevertheless

we can easily delay this loss of orthogonality using a higher precision arithmetic, i.e., a small machine ϵ.

Proof: Separating the last column of (2.1) by multiplying it with $\boldsymbol{e}_m$ from the right we get

$$\beta_m \boldsymbol{v}_{m+1} = \boldsymbol{A}\boldsymbol{v}_m - \alpha_m \boldsymbol{v}_m - \beta_{m-1}\boldsymbol{v}_{m-1} - \boldsymbol{f}_m.$$

Now multiplication with $\boldsymbol{V}_m^T$ from left yields

$$\beta_m \underbrace{\boldsymbol{V}_m^T \boldsymbol{v}_{m+1}}_{=:\boldsymbol{q}_{m+1}} = \underbrace{\boldsymbol{V}_m^T \boldsymbol{A}}_{\boldsymbol{T}_m^T \boldsymbol{V}_m^T + \beta_m \boldsymbol{e}_m \boldsymbol{v}_{m+1}^T + \boldsymbol{F}_m^T} \boldsymbol{v}_m - \alpha_m \boldsymbol{V}_m^T \boldsymbol{v}_m - \beta_{m-1}\boldsymbol{V}_m^T \boldsymbol{v}_{m-1} - \boldsymbol{V}_m^T \boldsymbol{f}_m$$

$$\Leftrightarrow \qquad \beta_m \boldsymbol{q}_{m+1} = \boldsymbol{T}_m \overset{\leftrightarrow}{\boldsymbol{q}}_m - \alpha_m \overset{\leftrightarrow}{\boldsymbol{q}}_m - \beta_{m-1}\overset{\leftrightarrow}{\boldsymbol{q}}_{m-1} + \beta_m \boldsymbol{e}_m \boldsymbol{v}_{m+1}^T \boldsymbol{v}_m$$
$$+ \underbrace{\boldsymbol{F}_m^T \boldsymbol{v}_m - \boldsymbol{V}_m^T \boldsymbol{f}_m}_{=:\boldsymbol{g}_m}$$

$$\Leftrightarrow \qquad \beta_m \boldsymbol{q}_{m+1}\boldsymbol{e}_m^T = \boldsymbol{T}_m \overset{\leftrightarrow}{\boldsymbol{Q}}_m - \overset{\leftrightarrow}{\boldsymbol{Q}}_m \boldsymbol{T}_m + \overset{\leftrightarrow}{\boldsymbol{G}}_m. \tag{2.2}$$

Where, to facilitate additions of vectors with different lengths and the collection of vectors with different lengths as columns of a matrix, we introduced the $\overset{\leftrightarrow}{\cdot}$-operator which extends a vector to the appropriate size by adding some zero-elements at the bottom.

Let now the vectors $\boldsymbol{s}_j$, $\boldsymbol{z}_j$, and the scalars λ_j be defined in the same way as in the theorem. Multiplying (2.2) with $\boldsymbol{s}_j^T$ from left and $\boldsymbol{s}_j$ from right for any $j \leq m$ yields

$$\beta_m \underbrace{\boldsymbol{s}_j^T \boldsymbol{V}_m^T}_{\boldsymbol{z}_j^T} \boldsymbol{v}_{m+1} \underbrace{\boldsymbol{e}_m^T \boldsymbol{s}_j}_{s_{mj}} = \underbrace{\boldsymbol{s}_j^T \boldsymbol{T}_m}_{\lambda_j \boldsymbol{s}_j^T}\overset{\leftrightarrow}{\boldsymbol{Q}}_m \boldsymbol{s}_j - \boldsymbol{s}_j^T \overset{\leftrightarrow}{\boldsymbol{Q}}_m \underbrace{\boldsymbol{T}_m \boldsymbol{s}_j}_{\lambda_j \boldsymbol{s}_j} + \boldsymbol{s}_j^T \overset{\leftrightarrow}{\boldsymbol{G}}_m \boldsymbol{s}_j$$

$$\Leftrightarrow \qquad \boldsymbol{z}_j^T \boldsymbol{v}_{m+1} = \frac{\boldsymbol{s}_j^T \overset{\leftrightarrow}{\boldsymbol{G}}_m \boldsymbol{s}_j}{\beta_m s_{mj}}. \tag{2.3}$$

The numerator of this fraction is approximately of size $\epsilon \|\boldsymbol{A}\|_2 \|\boldsymbol{z}_j\|_2$. In order to estimate the denominator we look at the *quality* of the Ritz-pair $(\lambda_j, \boldsymbol{z}_j)$:

$$\|\boldsymbol{A}\boldsymbol{z}_j - \lambda_j \boldsymbol{z}_j\|_2 = \|\underbrace{\boldsymbol{A}\boldsymbol{V}_m}_{\boldsymbol{V}_m \boldsymbol{T}_m + \beta_m \boldsymbol{v}_{m+1}\boldsymbol{e}_m^T + \boldsymbol{F}_m}\boldsymbol{s}_j - \lambda_j \boldsymbol{V}_m \boldsymbol{s}_j\|_2$$
$$\leq \|\beta_m \boldsymbol{v}_{m+1}s_{mj}\|_2 + \|\boldsymbol{F}_m\|_2$$
$$\approx |\beta_m s_{mj}| + \epsilon \|\boldsymbol{A}\|_2. \tag{2.4}$$

Collecting our results in (2.3) and (2.4) we finally obtain

$$\angle(\boldsymbol{z}_j, \boldsymbol{v}_{m+1}) = \text{Arccos}\left(\frac{|\boldsymbol{z}_j^T \boldsymbol{v}_{m+1}|}{\|\boldsymbol{z}_j\|_2 \|\boldsymbol{v}_{m+1}\|_2}\right) \approx \text{Arccos}\left(\frac{\epsilon \|\boldsymbol{A}\|_2}{|\,\|\boldsymbol{A}\boldsymbol{z}_j - \lambda_j \boldsymbol{z}_j\|_2 - \epsilon \|\boldsymbol{A}\|_2|}\right)$$

$$\blacksquare$$

An obvious way to retain the orthogonality among the Lanczos vectors is to orthogonalize each newly computed vector to all its predecessors. In fact, this is

almost equivalent to applying the Arnoldi procedure to a symmetric matrix. With this approach, the computed Lanczos vectors are orthogonal up to machine precision but we have to pay a high price concerning computing time and storage demands.

A more carefully inspection of Theorem 2.1 shows that re-orthogonalizing against all previous Lanczos vectors is not necessary [120]. The *bad guys* are only the (almost) converged Ritz vectors z (with Ritz value λ), since $\angle(z, v_k)$ tends to zero only if $\|Az - \lambda z\|_2$ is sufficiently small. Exploiting this fact leads to so called selective orthogonalization [101] where only *good* Ritz vectors are stored and used for re-orthogonalizing of newly computed Lanczos vectors. In this sense, a Ritz pair (z, λ) is called *good* if it satisfies

$$\|Az - \lambda z\|_2 \leq \sqrt{\epsilon}\|A\|_2.$$

An even more refined reorthogonalization strategy called partial reorthogonalization can be found in [121].

2.3   Examples

2.3.1   Preconditioning

Preconditioning is very important in solving linear systems because well conditioned systems are much easier and particularly faster to solve. However, preconditioning in finite precision can cause a drastically perturbed solution. Thus an important question is: *How does preconditioning affect the solution of a linear system?*

Since in modern Krylov subspace solvers preconditioning is no separate step but an inherent part to the solver itself (see Algorithm 1.2 on page 14), it cannot be distinguished which part of the deviation between the solutions of a preconditioned and a non-preconditioned system is caused by the preconditioning itself and which part is induced by various other error sources.

To give an idea of the magnitude of the perturbation that can be caused by preconditioning we consider the following example. We use a Jacobi preconditioner, which is so easy to apply to an entire linear system, that it is often used to transform A and b in advance. Afterwards a non-preconditioning solver is applied. This means we scale A to get a unit diagonal (we just perform one division for each element of A and b). This operation is done in IEEE double-precision. In order to identify the error caused by this scaling operation, we apply a verifying solver to the scaled system. Here we solve systems GK4.16(n) which result from a 5-point discretization of a fourth order ODE, see (6.1) on page 89. The resulting verified solutions of the scaled systems are shown in Figure 2.3.

For comparison we also plotted the verified solutions of the non-preconditioned systems (compare [34]). They differ so little that they appear as a single line (x). Additionally, we drew the solution of the underlying continuous problem. The discretization errors are so small that this curve is also not distinguishable from the non-preconditioned solutions (x).

This example demonstrates that preconditioning in this traditional way may introduce unacceptably large errors which can be significantly larger than the discretization error of an underlying continuous problem (which is less than 10^{-6} in our examples). It is possible to avoid these problems by doing verified preconditioning
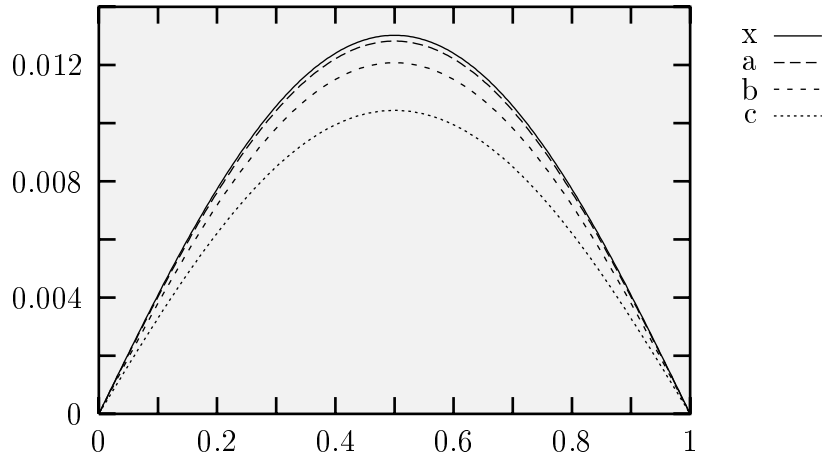
Figure 2.3: The effect of preconditioning on ill-conditioned linear systems. The curves represent the solutions of the preconditioned systems a: GK(8191), b: GK(12287), and c: GK(16383) with cond $\approx 10^{14}$, 10^{15}, and 10^{16}, resp. The exact solutions (compare [34]) of the non-preconditioned systems differ so little, that they appear as a single line (x).

but this leads to an interval matrix and therefore to an often unacceptable computational effort to solve these systems. Another, maybe more practical way is to use a more accurate arithmetic to avoid significant propagation of the various errors.

2.3.2 Convergence

One well known Krylov subspace method is the Conjugate Gradient algorithm. It can be interpreted as a Lanczos procedure and a subsequent $\boldsymbol{LDL}^T$-factorization to solve the tridiagonal system $\boldsymbol{T}\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{b}}$ (see Section 1.5.2) [41]. Then the residuals turn out to be scaled versions of the Krylov basis vectors and hence should stay orthogonal to their preceding residuals.

In Figure 2.4 the Euclidean norms of the residual and the error in each step are plotted during solving a GK(1023) system. Additionally we show the level of orthogonality of the new residual-vector $\boldsymbol{r}_{m+1}$ to the previous ones: $\max_{k=1}^{m}\{\langle \boldsymbol{r}_k \mid \boldsymbol{r}_{m+1}\rangle/(\|\boldsymbol{r}_k\|_2\|\boldsymbol{r}_{m+1}\|_2)\}$. As we can see, there is no convergence at all up to step $m = 1.5n$ and particularly no convergence at the theoretically guaranteed step $m = n$. One reason is easy to identify (see Theorem 2.1 or [51, 128, 129]): the basis of the Krylov subspace loses its orthogonality completely at $m \approx 400$ and the *basis*-vectors may even become linearly dependent. So CG can't minimize the residual in the entire $\mathbb{R}^m$ but only in a smaller subspace.

Further we can observe that the error norm runs into saturation at a level of approximately 10^{-6}. This matches with the well known rule of thumb saying that we may lose up to $\log(\text{cond}(\boldsymbol{A}))$ (≈ 10 in this case) digits from the 16 decimal digits we have in IEEE double precision.
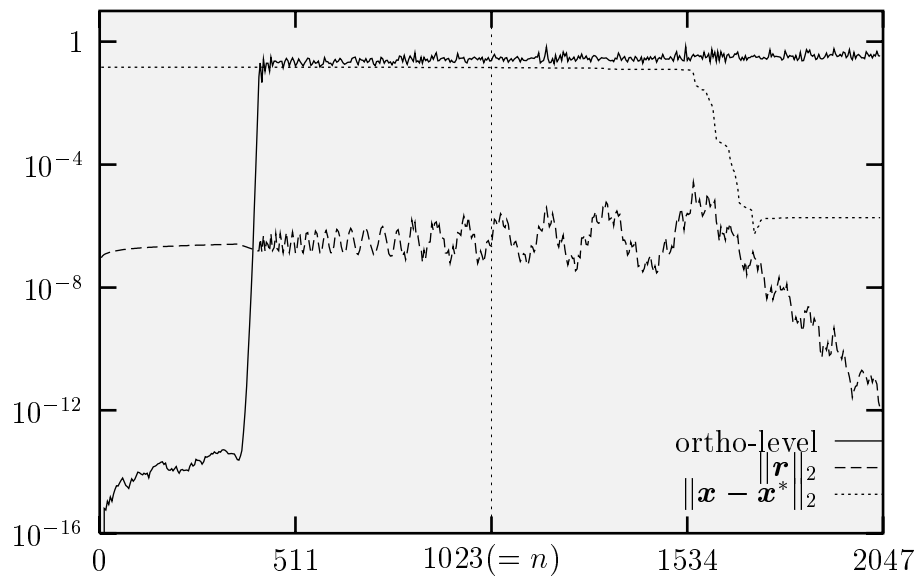
Figure 2.4: The Euclidean norms of the residual (dashed) and the error (dotted), and the level of orthogonality (solid) during solving the $\mathsf{GK}(1023)$ system. (ortho-level $= \max_{k=1}^{m}\{\langle \boldsymbol{r}_k \mid \boldsymbol{r}_{m+1}\rangle / (\|\boldsymbol{r}_k\|_2 \|\boldsymbol{r}_{m+1}\|_2)\}$)

3

Improved Arithmetics

In many numerical algorithms there is a large gap between the theoretical, i.e., mathematical, behavior on the one hand and the finite precision behavior on the other hand. In cases where the accuracy of a result is insufficient or no results can be obtained at all due to poorly conditioned problems, it is desirable to have a *better* arithmetic. Particularly, iterative algorithms sometimes even speed up because a higher precision arithmetic produces fewer errors that have to be minimized by the algorithm and therefore often iterations can be saved. In fact, it is nearly always possible to save iterations, but since the computing time per iteration increases with higher precision, we only sometimes really save time. However, we should always take into consideration that the higher computing time per iteration is often due to missing hardware support. For example the *exact scalar product* (Section 3.1), suitably supported in hardware, can be computed at least as fast as the standard scalar product (see [74]).

To narrow the gap between exact and finite precision arithmetic, often some minor arithmetic improvements suffice to get the desired results. One uses a more precise arithmetic (see Section 3.2). A second possibility is to leave the data type but control the rounding errors introduced by arithmetic operations performed on these numbers as described in Section 3.3. Further on we may pick out some frequently

used operations or functions and improve their accuracy. In particular, we focus on the scalar product (Section 3.1) which is a fundamental operation in numerical linear algebra. We start with this *exact scalar product* because this concept is needed for one of the higher precision arithmetics we describe in Section 3.2.

3.1   The Exact Scalar Product

Since scalar products occur very frequently and are important basic operations in numerical linear algebra, it is advantageous to perform this operation with the same precision as the other basic operations like addition or subtraction. Usually, scalar products are implemented by use of ordinary multiplication and addition and we have to beware of a lot of roundoff errors and their amplification due to cancellation. This is not necessary as has been shown by Kulisch [67, 74, 75].

The basic idea is first to multiply the floating-point vector elements exactly, that means we have to store the result in a floating-point format with double mantissa length and doubled exponent range[1]. Secondly we have to accumulate these products without any roundoff errors, see Figure 3.1. One possibility to achieve this is by use
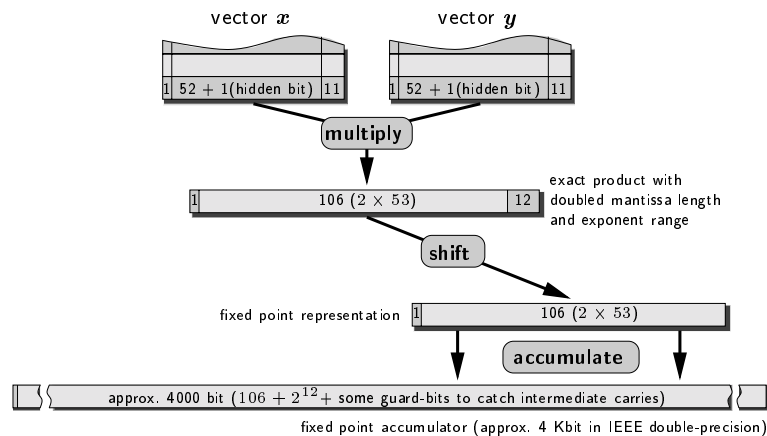
Figure 3.1: The basic idea of the exact scalar product, implemented by means of a long accumulator.

of a fixed point accumulator that covers the doubled floating-point number range plus some extra bits for intermediate overflows. At a first glance one might think that this accumulator must be very large, but in fact for the IEEE double-precision format, a little more than half a kilobyte is sufficient: 106 mantissa bits (for a zero exponent) plus 2^{11} binary digits for all possible left shifts and the same for right shifts plus one sign bit and some guard bits.

In Karlsruhe we built this operation in hardware as a numerical co-processor called XPA-3233 (eXtended Precision Arithmetic on a 32 bit PCI bus with 33 MHz clock speed), see [10]. On the XPA-3233 we use 67×64 bit words of storage. With this 92 guard bits even a tera-flop computer would need more than a hundred million years to cause an

---

[1] That is a sign bit, 2×53 mantissa bits and $11 + 1$ bits for the exponent, i.e., a total of 119 bits for the IEEE double-precision format.

overflow[2]. After accumulation of all products we usually have to round the result into the floating-point format which is symbolized with the $\Box$-operator. That is, the entire scalar product operation can be computed with just one rounding at all and therefore we have the much sharper bound

$$\left| \sum_{i=1}^{n} x_i y_i - \Box \left(\sum_{i=1}^{n} x_i y_i \right) \right| \leq \epsilon \left| \sum_{i=1}^{n} x_i y_i \right|$$

for the relative error than we usually have for scalar products if we use an ordinary floating-point arithmetic with a rounding after each multiplication and accumulation (see [61, 90])

$$\left| \sum_{i=1}^{n} x_i y_i - \boxed{\sum_{i=1}^{n}} x_i \boxdot y_i \right| \leq \frac{n\epsilon}{1 - n\epsilon} \sum_{i=1}^{n} |x_i y_i|,$$

which can be arbitrarily bad if $\sum_i |x_i y_i| \gg |\sum_i x_i y_i|$.

3.2   Multiple Precision

There are various tools and libraries providing numbers with a higher precision than the build-in data types (usually IEEE double precision) [13, 103, 123]. We can subdivide them in two fundamental types. The first implements exact numbers, i.e., numbers with infinite precision while the latter offers higher but finite precision. Because of storage and computing time requirements, we only focus on the latter multiple precision numbers. They are subdivided ac-

Figure 3.2: Higher precision data types.

cording to different implementation techniques in so called *staggered* numbers which are basically a sum of ordinary floating-point numbers (Section 3.2.1) and numbers with a contiguous mantissa, i.e., long floating point numbers (Section 3.2.2). The latter are mostly implemented using an integer field for the mantissa and some additional memory for the exponent and sign.
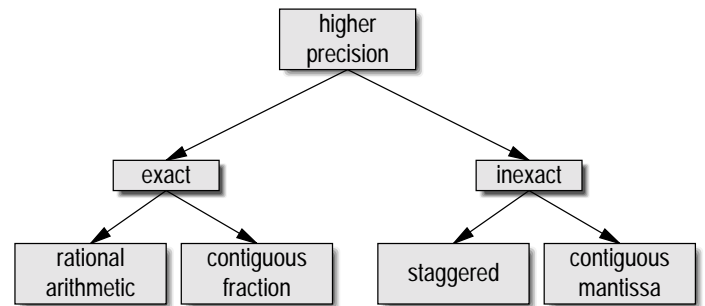
3.2.1   Staggered Precision Numbers

In this section we introduce the basic ideas of the so called staggered arithmetic. We give the definition of staggered precision numbers and show how the basic arithmetic operations can be performed by use of the exact scalar product (see Section 3.1). Instancing the square root as an example, we illustrate how elementary functions can be realized for staggered precision numbers.

Definition 3.1 *Given l floating-point numbers* $x^{(1)}, \ldots, x^{(l)}$ *we define a staggered precision number (or short: staggered number)* x *with staggered length l by*

$$x := \sum_{k=1}^{l} x^{(k)}.$$

---

[2]In fact, the XPA3233 only provides 90 guard bits, because 2 bits have a special meaning.

To ensure maximum precision we are interested in *non-overlapping* staggered numbers, that is, staggered numbers with $|x^{(1)}| > \ldots > |x^{(l)}|$ and the exponents of two successive summands $x^{(k)}$, $x^{(k+1)}$ differ at least by the mantissa length m of the floating-point system. In this case, the staggered number x represents a high precision number with at least $m \cdot l$ mantissa digits but with the same exponent range as the underlying floating-point system.

Let us assume, for example, we have a floating-point system with 3 decimal mantissa digits. Then with $x^{(1)} = 1.65 \cdot 10^3$, $x^{(2)} = 3.94 \cdot 10^0$, $x^{(3)} = 5.75 \cdot 10^{-5}$, and $x^{(4)} = 2.24 \cdot 10^{-8}$ the staggered number x of length 4:

$$x = 1.65 \cdot 10^3 + 3.94 \cdot 10^0 + 5.75 \cdot 10^{-5} + 2.24 \cdot 10^{-8} = 1.6539400575224 \cdot 10^3$$

represents a higher precision floating-point number with a minimum of 12 (14 in this case) decimal digits

- *Basic Arithmetic Operations*

Designing the basic arithmetic operations for staggered numbers, we have to decide which precision, i.e., which staggered length the result should have. Trying always to represent the exact result soon leads to very large numbers and fails already for the division where we usually get infinitely many digits. Therefore we define the staggered length of the resulting staggered number as the maximum of the staggered lengths of the operands.

Having the exact scalar product available, the algorithms for addition, subtraction and even multiplication are really simple. We just accumulate the result in a long accumulator and subsequently we round out the summands of the resulting staggered number as described in Algorithm 3.1. Further on we refer to this algorithm by the notation $z = round\_to\_staggered(accu)$ where z is a staggered number and $accu$ is a long accumulator.

> Given a long accumulator $accu$
> $l = staggered\_length\_of(z)$
> **for** $k = 1, \ldots, l$
> $\quad z^{(k)} = round\_towards\_zero(accu)$
> $\quad accu = accu - z^{(k)}$

Algorithm 3.1: Successively rounding out a long accumulator $accu$ into the summands of a staggered number z.

As an example for a basic arithmetic operation we state the subtraction. Given two staggered numbers $x := \sum_{k=1}^{l_1} x^{(k)}$ and $y := \sum_{k=1}^{l_2} y^{(k)}$ we compute $z := \sum_{k=1}^{\max\{l_1, l_2\}} z^{(k)} := x - y$ as showed in Algorithm 3.2.

- *Elementary Functions*

Since in this thesis, we only need the square root function in addition to the basic arithmetic operations, we only describe this function. However, with the basic arithmetic operations at hand, it is also possible to compute other elementary functions.

Given two staggered numbers x and y
$l_1 = staggered\_length\_of(x)$; $l_2 = staggered\_length\_of(y)$
$accu = 0$
for $k = 1, \ldots, l_1$
    $accu = accu + x^{(k)}$
for $k = 1, \ldots, l_2$
    $accu = accu - y^{(k)}$
$set\_length(z, \max\{l_1, l_2\})$ $z = round\_to\_staggered(accu)$

Algorithm 3.2: Subtraction of staggered numbers x and y. The intermediate result is stored in the long accumulator $accu$.

To compute the inverse of a given elementary function it is often helpful to use a Newton iteration. In the case of square roots $y = \sqrt{x}$ we set $f(y) = y^2 - x$ and approximate the zero y^* of f, i.e., the square root of x by a sequence y_i defined by

$$y_0 = \sqrt{\square \left(\sum_{k=1}^{l} x^{(k)} \right)}, \qquad y_i = y_{i-1} - \frac{f(y_{i-1})}{f'(y_{i-1})} = \frac{1}{2} \left(\frac{x}{y_{i-1}} + y_{i-1} \right), \quad i = 1, 2, \ldots$$

This algorithm is also known as Heron algorithm. Since the Newton iteration converges quadratically in a neighborhood of y^*, $\sqrt{l}$ iterations should suffice to obtain enough correct digits for y.

The staggered technique is implemented for example in the XSC languages [69] and is massively based on the availability of the exact scalar product. The special case where the number of ordinary floating-point numbers used to define a staggered number is fixed to 2 can also be coded by use of some arithmetic tricks without the exact scalar product. These tricks are mainly based on ideas by Dekker and Kahan [25]. A fast implementation of this latter technique in C++ is the `doubledouble` library, see [80].

- *Vector Operations*

Since we use the exact scalar product anyway for computing the products and additions in scalar products of staggered vectors, we can easily implement a exact scalar product for staggered vectors. Avoiding the intermediate roundings, the exact staggered scalar product is even faster than the ordinary scalar product. We describe this exact inner multiplication of staggered vectors in Algorithm 3.3.

3.2.2 Contiguous Mantissas

The contiguous mantissa type numbers can be addressed as long floating-point numbers. Usually, there is an integer variable containing the sign and exponent and an array of integers storing the mantissa. This is illustrated in Figure 3.3

In comparison to the staggered numbers, there is no possibility to exploit gaps, i.e., zeroes in the mantissa, but we have a more compact representation since we only store one exponent for the entire number. There exists an excellent implementation in C/assembler, the Gnu Multi-Precision library (GNU MP or GMP) [46] which

Given two staggered vectors $\boldsymbol{x} = \left(x_i\right)_{i=1}^n$ and $\boldsymbol{y} = \left(y_i\right)_{i=1}^n$

$l_1 = staggered\_length\_of(\boldsymbol{x})$

$l_2 = staggered\_length\_of(\boldsymbol{y})$

$accu = 0$

for $i = 1, \ldots, n$

   for $j = 1, \ldots, l_1$

      for $k = 1, \ldots, l_2$

         $accu = accu + x_i^{(j)} y_i^{(k)}$

$z = round\_to\_staggered(accu)$

Algorithm 3.3: Exact scalar product of staggered vectors $\boldsymbol{x}$ and $\boldsymbol{y}$. The intermediate result is stored in the long accumulator *accu*.
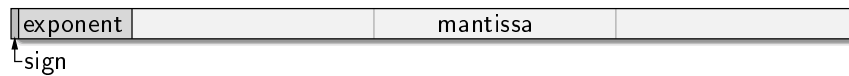
Figure 3.3: A *long* floating-point number with a contiguous mantissa.

aims to provide the fastest multiple precision library. For this purpose there are carefully optimized assembler routines for nearly all important architectures and a generic C formulation for non-standard computers.
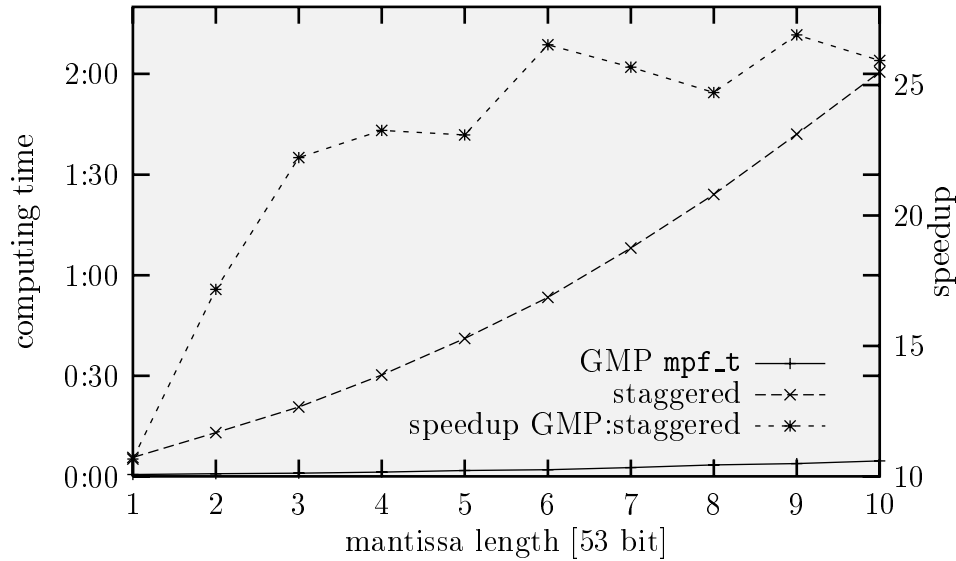
This results in a very high performance multiple precision arithmetic. Figure 3.4 shows the computing time for 100 000 evaluations of a typical expression in staggered precision arithmetic (C-XSC) and in a contiguous mantissa arithmetic (GMP). The speedup of GMP against staggered arithmetic levels off at approximately 25 which is mainly due to two reasons: First, the software simulation of the exact scalar product uses integer arithmetic, i.e., the floating-point summands of a staggered number have to be decomposed and composed repeatedly. This is avoided for the contiguous mantissa type numbers, because they use their own number format. Secondly this software simulation is relatively slow anyway since it is written in C instead of assembler as is the GMP arithmetic.

Since there was no object oriented interface for `gmp`, I implemented one with a complete set of overloaded operators. This interface is called `gmp++` and is included in **vk** (see Chapter 6). It enables us to use multiple precision numbers for generic algorithms (compare Section 5.1).

3.3   Interval Arithmetic

In order to get reliable results, e.g., for error bounds of linear systems, it is not sufficient to reduce the rounding errors. We have to control this arithmetic uncertainty completely [3]. As we have seen in Section 2.1, each basic operation — even if it has floating-point operands — may have a result which is not representable exactly in the given floating point format. Thus each basic operation involves a rounding back into the floating-point screen and therefore causes an error. Clearly, there is no necessity to use *random* roundings and if we are interested in reliable bounds to the exact result, we simply have to return two floating-point numbers: One, which is guaranteed to be larger than the exact result, e.g., it's upwardly rounded value and

Figure 3.4: Computing time for $100\,000$ evaluations of a typical expression in staggered precision arithmetic (C-XSC) and in a contiguous mantissa arithmetic (GMP).

one which is smaller, e.g., it's downwardly rounded value. This idea automatically leads to an interval data type as defined in the following definition.

Definition 3.2 *Let $\underline{x} \le \bar{x}$, then we call the set $[x] := [\underline{x}, \bar{x}] := \{\xi \in \mathbb{R} \mid \underline{x} \le \xi \le \bar{x}\}$ an interval.*

Note: even if we have such an interval on a computer, that is, $\underline{x}$ and $\bar{x}$ are floating-point numbers, the interval $[\underline{x}, \bar{x}]$ contains *all real* numbers between $\underline{x}$ and $\bar{x}$ and not only floating-point numbers.

Since intervals are elements of the power-set of $\mathbb{R}$, the basic arithmetic operations are defined by restriction of the power-set operations

$$[\underline{x}, \bar{x}] \circ [\underline{y}, \bar{y}] := \{\xi \circ \eta \mid \underline{x} \le \xi \le \bar{x} \wedge \underline{y} \le \eta \le \bar{y}\} \text{ with } \circ \in \{+, -, \times, /\} \\ (\text{and } 0 \notin [\underline{y}, \bar{y}] \text{ for } \circ = /) \tag{3.1}$$

Exploiting monotonicity properties, this infinitely many operations needed to compute $[\underline{x}, \bar{x}] \circ [\underline{y}, \bar{y}]$ in (3.1) reduce to the computation of only a few operations. For example the addition of intervals can be computed by $[\underline{x}, \bar{x}] + [\underline{y}, \bar{y}] = [\underline{x} + \underline{y}, \bar{x} + \bar{y}]$. Unfortunately, the right hand side of this equation, i.e., the bounds of the resulting interval may not be representable in our floating-point format. To make sure that we enclose the exact solution interval on the computer, we again have to replace $\underline{x} + \underline{y}$ by a smaller floating-point number and $\bar{x} + \bar{y}$ by a larger one [73, 76]. Usually this lower respectively upper floating-point bounds are obtained by selecting the correct directed rounding mode (compare Section 2.1). On a computer we define the addition of intervals via

$$[\underline{x}, \bar{x}] + [\underline{y}, \bar{y}] := [\nabla(\underline{x} + \underline{y}), \triangle(\bar{x} + \bar{y})], \tag{3.2}$$

where ∇ and $\triangle$ denotes the downward respectively upward directed rounding. Sometimes we use the shorter notation

$$[\underline{x}, \bar{x}] + [\underline{y}, \bar{y}] := \diamond([\underline{x} + \underline{y}, \bar{x} + \bar{y}]).$$

There are various libraries providing interval data types, including the XSC languages [57, 58, 69], `Profil/BIAS` [71, 72] and Intlab [111] and newly there are commercial Fortran/C/C++ compilers from Sun-Microsystems [131] providing an interval data type. Using other libraries, one must carefully inspect if correct rounding modes are used.

There is also a interval-staggered library which provides higher precision intervals in Pascal-XSC [82].

• *Excursion: Enclosing Floating-Point Computations*

In particular, the exact scalar product is very useful if we have to compute an enclosure of a linear expression only with floating-point arguments. For example, if we have a floating-point matrix $\boldsymbol{A}$, a floating-point right hand side vector $\boldsymbol{b}$, and floating-point approximate solution $\tilde{\boldsymbol{x}}$ and we want to compute an enclosure of the residual vector, we can either cast all floating-point numbers to intervals (via the $\diamond$-operator) and then use standard interval arithmetic ($\diamond, \diamond$), see Algorithm 3.4a, or we can use the exact scalar product to compute the exact residual. Finally, this exact but long number has to be rounded to a (very tight) interval with floating-point bounds and relative diameter less than or equal to ϵ, see Algorithm 3.4b.

$$
\begin{array}{ll}
\textbf{for } i = 1, \ldots, n \\
\quad r_i = \diamond b_i \\
\quad \textbf{for } j = 1, \ldots, n \\
\quad\quad r_i = r_i \diamond a_{i,j} \diamond \tilde{x}_j
\end{array}
\qquad
\begin{array}{ll}
\textbf{for } i = 1, \ldots, n \\
\quad accu = b_i \\
\quad \textbf{for } j = 1, \ldots, n \\
\quad\quad accu = accu - a_{i,j} \cdot \tilde{x}_j \quad /\text{* exact */} \\
\quad r_i = \diamond accu
\end{array}
$$

a) with interval arithmetic   b) with the exact scalar product

Algorithm 3.4: Computation of an enclosure of the residual with two different techniques.

With approach a), the relative error in component i can only be bounded by

$$\frac{(n+1)\epsilon}{1 - (n+1)\epsilon} \cdot \left(|r_i| + \sum_{j=1}^{n} |a_{i,j}\tilde{x}_j| \right),$$

which might be arbitrary bad, if $|r_i| + \sum_{j=1}^{n} |a_{i,j}\tilde{x}_j| \gg \left| r_i - \sum_{j=1}^{n} a_{i,j}\tilde{x}_j \right|$.

4

Error Bounds for Solutions of Linear Systems

> ❝ *No method of solving a computational problem is really available to a user until it is completely described in an algebraic computing language and made completely reliable.* ❞
>
> George E. Forsythe, 1967

This chapter gives an overview about the most important verification methods for linear systems of equations. Section 4.1 shows straight forward extensions of point-algorithms to interval algorithms, while Section 4.2 focuses on verification algorithms based on fixed point theorems [76, 86, 108]. The latter class of algorithms is much more general than the first one, but suffers in two important ways from the underlying interval arithmetic. First, due to the software simulated interval arithmetic, they are relatively slow and secondly, one has to pay either with a big computational overhead or a significant loss of accuracy because of the so called wrapping effect (see [92]).

In Section 4.3 we present a different class of verification algorithms. Instead of delivering an enclosure of each solution component, they only compute a rigorous bound for the error norm. This might be disadvantageous if the solution components have highly different magnitudes. However, since most of the computation

can be done approximately (i.e., in ordinary floating-point arithmetic) with only a subsequent verification procedure, the resulting algorithms often are much faster than the previous ones.

4.1   Interval Extensions of Point Algorithms

The most obvious way to obtain a verifying algorithm is using a given point-algorithm and replacing every floating-point operation by the corresponding interval operation. It has been shown that, for example, the interval version of Gaussian elimination is executable in this way for diagonally dominant matrices or M-matrices. For general matrices the intervals tend to grow in diameter rapidly and it may soon happen that a pivot column solely consists of intervals containing zero. In this case, the algorithm terminates prematurely without computing an enclosure of the solution. However, if $\boldsymbol{A}$ has special properties, the Interval Gauß algorithm (IGA) may even be capable to produce optimal enclosures, that is, the interval vector $[\boldsymbol{x}]$ is the smallest n-dimensional box enclosing the solution set [87]. See Paragraph *Solutions of symmetric tridiagonal systems* in Section 4.3.3 for an example with particular tridiagonal matrices.

4.2   Enclosures via Fixed Point Methods

Suppose we have an approximate inverse $\boldsymbol{R}$ for $\boldsymbol{A}$ then we can define a sequence of vectors $(\boldsymbol{x}_k)_{k\in\mathbb{N}}$ by

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{R}(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_k) = \boldsymbol{R}\boldsymbol{b} + (\boldsymbol{I} - \boldsymbol{R}\boldsymbol{A})\boldsymbol{x}_k.$$

This vector sequence converges for every $\boldsymbol{x}$ if and only if the spectral radius of $\boldsymbol{I} - \boldsymbol{R}\boldsymbol{A}$ is less than one.

If $\mathcal{X}$ is a non-empty, convex, and compact subset of $\mathbb{R}^n$ then by Brouwer's fixed point theorem

$$\mathcal{X} \supseteq \mathcal{X} + \boldsymbol{R}(\boldsymbol{b} - \boldsymbol{A}\mathcal{X}) \quad \text{implies} \quad \exists\, \boldsymbol{x} \in \mathcal{X} : \boldsymbol{R}(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}) = \boldsymbol{o}.$$

Using an interval vector $[\boldsymbol{x}]$ as a special non-empty, convex, and compact subset of $\mathbb{R}^n$, we generally have $\mathrm{diam}([\boldsymbol{x}] + \boldsymbol{R}(\boldsymbol{b} - \boldsymbol{A}[\boldsymbol{x}])) > \mathrm{diam}([\boldsymbol{x}])$ and thus $[\boldsymbol{x}]$ will never be a superset of $[\boldsymbol{x}] + \boldsymbol{R}(\boldsymbol{b} - \boldsymbol{A}[\boldsymbol{x}])$. Moreover, only if $\boldsymbol{R}$ is nonsingular, then we guarantee that $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ is solvable, i.e., that $\boldsymbol{A}$ is nonsingular, too. These two problems are solved by the next theorem (compare [106]).

Theorem 4.1 *Let $\boldsymbol{A}$, $\boldsymbol{R} \in \mathbb{R}^{n \times n}$ and $\boldsymbol{b} \in \mathbb{R}^n$. Suppose for the interval vector $[\boldsymbol{x}]$ holds*

$$\boldsymbol{R}\boldsymbol{b} + (\boldsymbol{I} - \boldsymbol{R}\boldsymbol{A})[\boldsymbol{x}] \subseteq [\overset{\circ}{\boldsymbol{x}}] \tag{4.1}$$

then $\boldsymbol{A}$ and $\boldsymbol{R}$ are nonsingular and there is exactly one $\boldsymbol{x} \in [\boldsymbol{x}]$ satisfying $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$.

See [106] for a proof.

In a floating-point system (with denormalized mantissas), numbers are much narrower around zero. Therefore it is always a good idea to work with the error,

i.e., the difference of the exact solution $\boldsymbol{x}^*$ and the approximate solution $\tilde{\boldsymbol{x}}$ since this is hopefully close to zero. Applying Theorem 4.1 to $\tilde{\boldsymbol{x}} + [\boldsymbol{x}]$, condition (4.1) now reads

$$
\begin{aligned}
\boldsymbol{R}\boldsymbol{b} + (\boldsymbol{I} - \boldsymbol{R}\boldsymbol{A})(\tilde{\boldsymbol{x}} + [\boldsymbol{x}]) &\subseteq (\tilde{\boldsymbol{x}} + \overset{\circ}{[\boldsymbol{x}]}) \\
\Leftrightarrow \quad \boldsymbol{R}\boldsymbol{b} + \tilde{\boldsymbol{x}} - \boldsymbol{R}\boldsymbol{A}\tilde{\boldsymbol{x}} + (\boldsymbol{I} - \boldsymbol{R}\boldsymbol{A})[\boldsymbol{x}] &\subseteq \tilde{\boldsymbol{x}} + \overset{\circ}{[\boldsymbol{x}]} \quad\quad (4.2) \\
\Leftrightarrow \quad \boldsymbol{R}(\boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{x}}) + (\boldsymbol{I} - \boldsymbol{R}\boldsymbol{A})[\boldsymbol{x}] &\subseteq \overset{\circ}{[\boldsymbol{x}]}
\end{aligned}
$$

If condition (4.2) does not hold one may initiate the following iteration process

$$
\begin{aligned}
[\boldsymbol{x}]^{(0)} &:= [\boldsymbol{x}], \\
\widetilde{[\boldsymbol{x}]}^{(k)} &:= \boldsymbol{R}(\boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{x}}) + (\boldsymbol{I} - \boldsymbol{R}\boldsymbol{A})[\boldsymbol{x}]^{(k-1)} \\
[\boldsymbol{x}]^{(k)} &:= [1 - \epsilon, 1 + \epsilon] \cdot \widetilde{[\boldsymbol{x}]}^{(k)} \quad\quad \text{for k=1,2,\dots.}
\end{aligned}
$$

Then for some $k \in \mathbb{N}$ and $[\boldsymbol{x}]$ the inclusion $[\boldsymbol{x}]^{(k)} \subseteq \overset{\circ}{[\boldsymbol{x}]}^{(k-1)}$ holds. The question is for which $[\boldsymbol{x}]$ and which k we will obtain $[\boldsymbol{x}]^{(k)} \subseteq \overset{\circ}{[\boldsymbol{x}]}^{(k-1)}$. The answer is given by the following theorem.

Theorem 4.2 *Let $\boldsymbol{A}$, $\boldsymbol{R} \in \mathbb{R}^{n \times n}$ and $\boldsymbol{b} \in \mathbb{R}^n$. Then the following two statements are equivalent:*

1. *For each interval n-vector $[\boldsymbol{x}]$ with*

$$
\text{diam}([\boldsymbol{x}]) > \frac{2}{1 - \rho(|\boldsymbol{I} - \boldsymbol{R}\boldsymbol{A}|)} \cdot |\boldsymbol{R}(\boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{x}})|
$$

   there exists a $k \in \mathbb{N}$ with $[\boldsymbol{x}]^{(k)} \subseteq \overset{\circ}{[\boldsymbol{x}]}^{(k-1)}$

2. *$\rho(|\boldsymbol{I} - \boldsymbol{R}\boldsymbol{A}|) < 1$*

See [107] for a proof.

Obviously, we do not need $\boldsymbol{R}$ explicitly and it is sufficient to have a, e.g., triangular factorization $\boldsymbol{L}\boldsymbol{U}$ of $\boldsymbol{A}$ to compute $\boldsymbol{R}(\boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{x}})$ and $\boldsymbol{R}\boldsymbol{A}[\boldsymbol{x}]$ via forward and backward substitution. Note that this will often produve large overestimations for $\boldsymbol{R}\boldsymbol{A}[\boldsymbol{x}]$ due to the wrapping effect which occurs in solving triangular systems. To avoid this problem, one can either use a coordinate transformation technique as described by R. Lohner in [81] or one can substitute intervals by zonotopes which might cover the shape of the solution more appropriate. However, these triangular factors may exploit a possible sparsity in $\boldsymbol{A}$ (e.g., banded Cholesky) to make this algorithm applicable to larger matrices, compare [81].

4.3   Error Bounds via Perturbation Theory

Usually, stopping criteria for iterative solvers of linear systems are based on the norm of the residual $\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{x}}$. Since $\|\tilde{\boldsymbol{x}} - \boldsymbol{x}^*\|_2 \leq \|\boldsymbol{A}^{-1}\|_2 \cdot \|\boldsymbol{r}\|_2$, this gives a rough idea about the distance to the exact solution $\boldsymbol{x}^* = \boldsymbol{A}^{-1}\boldsymbol{b}$ if we have some information about $\|\boldsymbol{A}^{-1}\|_2$ or the condition of $\boldsymbol{A}$. Sometimes 'cheap' condition estimators [56]

are used to estimate $\mathrm{cond}_2(\boldsymbol{A})$ but this approach gives only error estimates instead of bounds. Our first task will be to compute a verified upper bound of $\|\tilde{\boldsymbol{x}} - \boldsymbol{x}^*\|_2$. Then we try to improve this worst case bound by taking into account some more knowledge about the spectrum of $\boldsymbol{A}$.

4.3.1   Basic Error Bounds

This section is essentially based on ideas of S. Rump [109, 110]. With $\|\boldsymbol{A}\|_2 \geq \sigma_{\min}(\boldsymbol{A})$ we have $\|\boldsymbol{A}^{-1}\|_2 \leq \sigma_{\min}^{-1}(\boldsymbol{A})$ and therefore

$$\|\tilde{\boldsymbol{x}} - \boldsymbol{x}^*\|_2 \leq \sigma_{\min}^{-1}(\boldsymbol{A}) \cdot \|\boldsymbol{r}\|_2.$$

A well known method to compute the smallest singular value of a matrix $\boldsymbol{A}$ is the inverse power method (with shift 0) (see [61]). Therefore it is necessary to have a factorization, say $(\boldsymbol{L}, \boldsymbol{U})$ of $\boldsymbol{A}$ that enables us to compute $(\boldsymbol{L}\boldsymbol{U})^{-1}\boldsymbol{z}$ for arbitrary $\boldsymbol{z}$ easily. Mostly, $\boldsymbol{L}\boldsymbol{U} = \boldsymbol{A}$ doesn't hold exactly but $\boldsymbol{L}\boldsymbol{U} = \tilde{\boldsymbol{A}} \approx \boldsymbol{A}$ is sufficient and it is often possible to get $\|\Delta\boldsymbol{A}\| = \|\tilde{\boldsymbol{A}} - \boldsymbol{A}\|$ fairly small. The next theorem clarifies how $\|\tilde{\boldsymbol{x}} - \boldsymbol{x}^*\|$ depends on the smallest singular value $\sigma_{\min}(\tilde{\boldsymbol{A}})$, $\Delta\boldsymbol{A}$ and $\|\boldsymbol{r}\|$.

Theorem 4.3 *Let $\boldsymbol{A} \in \mathrm{I\!R}^{n \times n}$, $\boldsymbol{b} \in \mathrm{I\!R}^n$ be given as well as a nonsingular $\tilde{\boldsymbol{A}} \in \mathrm{I\!R}^{n \times n}$ and $\tilde{\boldsymbol{x}} \in \mathrm{I\!R}^n$. Define $\Delta\boldsymbol{A} := \tilde{\boldsymbol{A}} - \boldsymbol{A}$, $\boldsymbol{r} := \boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{x}}$ and suppose*

$$\sigma_{min}(\tilde{\boldsymbol{A}}) > n^{1/2} \cdot \|\Delta\boldsymbol{A}\|_\infty.$$

Then $\boldsymbol{A}$ is nonsingular and for $\boldsymbol{x}^* := \boldsymbol{A}^{-1}\boldsymbol{b}$ holds

$$\|\boldsymbol{x}^* - \tilde{\boldsymbol{x}}\|_\infty \leq \frac{n^{1/2} \cdot \|\boldsymbol{r}\|_\infty}{\sigma_{min}(\tilde{\boldsymbol{A}}) - n^{1/2} \cdot \|\Delta\boldsymbol{A}\|_\infty}.$$

Proof: Since $\|\tilde{\boldsymbol{A}}^{-1}\Delta\boldsymbol{A}\|_2 \leq \sigma_{\min}(\tilde{\boldsymbol{A}})^{-1} \cdot \|\Delta\boldsymbol{A}\|_2 \leq n^{1/2} \cdot \sigma_{\min}(\tilde{\boldsymbol{A}})^{-1} \cdot \|\Delta\boldsymbol{A}\|_\infty < 1$ the matrix $\boldsymbol{I} - \tilde{\boldsymbol{A}}^{-1}\Delta\boldsymbol{A} = \tilde{\boldsymbol{A}}^{-1}\boldsymbol{A}$ and hence $\boldsymbol{A}$ itself is invertible. Now

$$(\boldsymbol{I} - \tilde{\boldsymbol{A}}^{-1}\Delta\boldsymbol{A})(\boldsymbol{x}^* - \tilde{\boldsymbol{x}}) = \tilde{\boldsymbol{A}}^{-1}\boldsymbol{A}(\boldsymbol{x}^* - \tilde{\boldsymbol{x}}) = \tilde{\boldsymbol{A}}^{-1} \cdot \boldsymbol{r}$$

and therefore

$$\|\boldsymbol{x}^* - \tilde{\boldsymbol{x}}\| = \|((\boldsymbol{I} - \tilde{\boldsymbol{A}}^{-1}\Delta\boldsymbol{A})^{-1} \cdot \tilde{\boldsymbol{A}}^{-1} \cdot \boldsymbol{r})\| \leq \|\tilde{\boldsymbol{A}}^{-1} \cdot \boldsymbol{r}\| \cdot \|(\boldsymbol{I} - \tilde{\boldsymbol{A}}^{-1}\Delta\boldsymbol{A})^{-1}\|. \quad (4.3)$$

Using $\|(\boldsymbol{I} - \boldsymbol{B})^{-1}\|_\infty \leq (1 - \|\boldsymbol{B}\|_\infty)^{-1}$ for convergent $\boldsymbol{B}$ (i.e., $\|\boldsymbol{B}\|_\infty < 1$) we get

$$\|\boldsymbol{x}^* - \tilde{\boldsymbol{x}}\|_\infty \leq \frac{\|\tilde{\boldsymbol{A}}^{-1}\|_\infty \cdot \|\boldsymbol{r}\|_\infty}{1 - \|\tilde{\boldsymbol{A}}^{-1} \cdot \Delta\boldsymbol{A}\|_\infty} \leq \frac{\|\tilde{\boldsymbol{A}}^{-1}\|_\infty \cdot \|\boldsymbol{r}\|_\infty}{1 - \|\tilde{\boldsymbol{A}}^{-1}\|_\infty \cdot \|\Delta\boldsymbol{A}\|_\infty}$$

and applying

$$\|\boldsymbol{B}\|_\infty \leq n^{1/2} \cdot \|\boldsymbol{B}\|_2 \leq n^{1/2} \cdot \sigma_{\min}(\boldsymbol{B}) \quad (4.4)$$

yields

$$\|\boldsymbol{x}^* - \tilde{\boldsymbol{x}}\|_\infty \leq \frac{n^{1/2} \cdot \sigma_{\min}(\tilde{\boldsymbol{A}})^{-1} \cdot \|\boldsymbol{r}\|_\infty}{1 - n^{1/2} \cdot \sigma_{\min}(\tilde{\boldsymbol{A}})^{-1} \cdot \|\Delta\boldsymbol{A}\|_\infty}.$$

■

Of course for sparse matrices this 2-norm bound (4.4) is a rough overestimation of the ∞-norm. Suppose $\boldsymbol{B}$ to have at most m nonzero elements per row and let $\beta := \max_{i,j}\{|\boldsymbol{B}_{i,j}|\}$ then both $\|\boldsymbol{B}\|_1$ and $\|\boldsymbol{B}\|_\infty$ are bounded by $m \cdot \beta$. Using $\|\boldsymbol{B}\|_2^2 \le \|\boldsymbol{B}\|_1 \cdot \|\boldsymbol{B}\|_\infty$ we get

$$\|\boldsymbol{B}\|_2 \le (\|\boldsymbol{B}\|_1 \cdot \|\boldsymbol{B}\|_\infty)^{1/2} \le m \cdot \beta.$$

Theorem 4.4 *Let $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, $\boldsymbol{b} \in \mathbb{R}^n$ be given as well as a nonsingular $\tilde{\boldsymbol{A}} \in \mathbb{R}^{n\times n}$ and $\tilde{\boldsymbol{x}} \in \mathbb{R}^n$. Define $\Delta\boldsymbol{A} := \tilde{\boldsymbol{A}} - \boldsymbol{A}$, $\boldsymbol{r} := \boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{x}}$ and suppose $\sigma_{min}(\tilde{\boldsymbol{A}}) > (\|\Delta\boldsymbol{A}\|_1 \cdot \|\Delta\boldsymbol{A}\|_\infty)^{1/2}$.*
Then $\boldsymbol{A}$ is nonsingular and for $\boldsymbol{x}^* := \boldsymbol{A}^{-1}\boldsymbol{b}$ holds

$$\|\boldsymbol{x}^* - \tilde{\boldsymbol{x}}\|_\infty \le \|\boldsymbol{x}^* - \tilde{\boldsymbol{x}}\|_2 \le \frac{\|\boldsymbol{r}\|_2}{\sigma_{min}(\tilde{\boldsymbol{A}}) - (\|\Delta\boldsymbol{A}\|_1 \cdot \|\Delta\boldsymbol{A}\|_\infty)^{1/2}}.$$

Proof: Starting with equation (4.3) but using the 2-norm instead of the ∞-norm we get

$$\|\boldsymbol{x}^* - \tilde{\boldsymbol{x}}\|_2 \le \frac{\|\tilde{\boldsymbol{A}}^{-1}\|_2 \cdot \|\boldsymbol{r}\|_2}{1 - \|\tilde{\boldsymbol{A}}^{-1}\|_2 \cdot \|\Delta\boldsymbol{A}\|_2} \le \frac{\|\boldsymbol{r}\|_2}{\sigma_{\min}(\tilde{\boldsymbol{A}}) - (\|\Delta\boldsymbol{A}\|_1 \cdot \|\Delta\boldsymbol{A}\|_\infty)^{1/2}}.$$

■

In practical computations it is a difficult task to get the smallest singular value of an arbitrary matrix $\boldsymbol{A}$ or at least a reliable lower bound of $\sigma_{\min}(\boldsymbol{A})$. But if we have an approximate decomposition, say $(\boldsymbol{L}, \boldsymbol{U})$ with $\tilde{\boldsymbol{A}} = \boldsymbol{L}\boldsymbol{U}$ we can apply inverse power iteration to $\boldsymbol{L}\boldsymbol{U}$ to compute $\sigma_{\min}(\tilde{\boldsymbol{A}})$. Of course, if we can compute a lower bound of $\sigma_{\min}(\boldsymbol{A})$ directly then by setting $\tilde{\boldsymbol{A}} := \boldsymbol{A}$ we get $\Delta\boldsymbol{A} = \boldsymbol{0}$ and thus

$$\|\boldsymbol{x}^* - \tilde{\boldsymbol{x}}\|_\infty \le \|\boldsymbol{x}^* - \tilde{\boldsymbol{x}}\|_2 \le \sigma_{\min}(\boldsymbol{A})^{-1} \cdot \|\boldsymbol{r}\|_2.$$

4.3.2 Improved Error Bounds

The ideas of this section are mostly due to Dahlquist [24] who stated an interesting connection between Lanczos procedures and Gaussian quadrature rules. Later on, these ideas were sophisticated in [43, 45].

We start with the well known relation between the error norm and the residual norm

$$\|\boldsymbol{x}^* - \tilde{\boldsymbol{x}}\|_2 = \|\boldsymbol{A}^{-1}\boldsymbol{b} - \tilde{\boldsymbol{x}}\|_2 = \|\boldsymbol{A}^{-1}(\boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{x}})\|_2 = \|\boldsymbol{A}^{-1}\boldsymbol{r}\|_2. \qquad (4.5)$$

Suppose $\boldsymbol{A}$ to be symmetric positive definite then we have real positive eigenvalues $\lambda_1, \ldots, \lambda_n$ (non-increasingly ordered) and an orthonormal basis of eigenvectors $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_n$. Using $\boldsymbol{\Lambda} := \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$ and $\boldsymbol{Q} := (\boldsymbol{q}_1 \mid \cdots \mid \boldsymbol{q}_n)$ yields

$$
\begin{aligned}
\|\boldsymbol{x}^* - \tilde{\boldsymbol{x}}\|_2 &= \|\boldsymbol{Q}^T(\boldsymbol{x}^* - \tilde{\boldsymbol{x}})\|_2 = \|\boldsymbol{Q}^T\boldsymbol{A}^{-1}\boldsymbol{r}\|_2 \\
&= \|\boldsymbol{Q}^T\boldsymbol{A}^{-1}\boldsymbol{Q} \cdot \boldsymbol{Q}^T\boldsymbol{r}\|_2 = \|\boldsymbol{\Lambda}^{-1} \cdot \boldsymbol{Q}^T\boldsymbol{r}\|_2 \qquad (4.6) \\
&\le \|\boldsymbol{\Lambda}^{-1}\|_2 \cdot \|\boldsymbol{Q}^T\boldsymbol{r}\|_2 = \lambda_{\min}^{-1} \cdot \|\boldsymbol{r}\|_2.
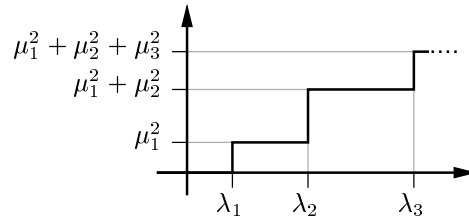\end{aligned}
$$

This estimate is a worst case bound which does not make any use of the extent to which each of the eigenvectors $\boldsymbol{q}_i$ is actually present in $\boldsymbol{r}$, i.e., of the size of the elements of $\boldsymbol{\mu} := \boldsymbol{Q}^T \boldsymbol{r}$. With (4.5) and (4.6) we see

$$
\begin{aligned}
\|\boldsymbol{x}^* - \tilde{\boldsymbol{x}}\|_2^2 = \|\boldsymbol{A}^{-1}\boldsymbol{r}\|_2^2 &= \langle \boldsymbol{A}^{-1}\boldsymbol{r} \mid \boldsymbol{A}^{-1}\boldsymbol{r} \rangle \\
&= \langle \boldsymbol{\Lambda}^{-1}\boldsymbol{Q}^T\boldsymbol{r} \mid \boldsymbol{\Lambda}^{-1}\boldsymbol{Q}^T\boldsymbol{r} \rangle \\
&= \langle \boldsymbol{\mu} \mid \boldsymbol{\Lambda}^{-2}\boldsymbol{\mu} \rangle \\
&= \sum_{i=1}^n \lambda_i^{-2} \cdot \mu_i^2 .
\end{aligned}
$$

In general the relation $\langle \boldsymbol{r} \mid \boldsymbol{A}^{-2}\boldsymbol{r} \rangle = \sum_{i=1}^n \lambda_i^{-2}\mu_i^2$ holds for any analytic function f and the sum can be considered as a Riemann-Stieltjes integral, see e.g. [42]:

$$
\langle \boldsymbol{r} \mid f(\boldsymbol{A})\boldsymbol{r} \rangle = \sum_{i=1}^n f(\lambda_i) \cdot \mu_i^2 = \int_a^b f(\lambda)d\mu \tag{4.7}
$$

with the piecewise constant, positive, and increasing measure μ defined via

$$
\mu(\lambda) = \begin{cases} 0 & \text{for } \lambda < \lambda_1 \\ \sum_{j=1}^i \mu_j^2 & \text{for } \lambda_i \le \lambda < \lambda_{i+1} \\ \sum_{j=1}^n \mu_j^2 & \text{for } \lambda_n \le \lambda \end{cases} .
$$

Note that the interval $[a, b]$ must contain the spectrum of $\boldsymbol{A}$, in particular $a \le \lambda_{\min}$ must hold.

Unfortunately we do not know the eigenvalues and eigenvectors of $\boldsymbol{A}$ and hence we cannot evaluate (4.7) directly. However, one way to obtain bounds for Riemann-Stieltjes integrals is to use Gauß and Gauß-Radau quadrature formulas.

Evaluating (4.7) with a Gauß quadrature rule with m nodes $(\xi_1, \ldots, \xi_m)$ and corresponding weights $(\omega_1, \ldots, \omega_m)$ we get

$$
\int_a^b f(\lambda)d\mu = \underbrace{\sum_{j=1}^m \omega_j f(\xi_j)}_{I_{\text{Gauß}}^{(m)}} + \underbrace{\frac{f^{(2m)}(\eta)}{(2m)!} \cdot \int_a^b \prod_{j=1}^m (\lambda - \xi_j)^2 d\mu}_{R_{\text{Gauß}}^{(m)}}, \quad \eta \in (a, b).
$$

with an integral approximation $I_{\text{Gauß}}^{(m)}$ and the remainder $R_{\text{Gauß}}^{(m)}$. Note that for $f(\cdot) = \cdot^{-2}$ we get

$$
R_{\text{Gauß}}^{(m)} = \frac{(-1)^{2m}(2m+1)!\lambda^{-(2m+2)}}{(2m)!} \cdot \int_a^b \prod_{j=1}^m (\lambda - \xi_j)^2 d\mu \quad \ge \quad 0
$$

and therefore $I_{\text{Gauß}}^{(m)} \le \int_a^b f(\lambda)d\mu$.

To obtain an upper bound, we apply a Gauß-Radau rule to (4.7) with $m + 1$ nodes $(\xi_1, \ldots, \xi_{m+1})$ with $\xi_{m+1} := a$ and corresponding weights $(\omega_1, \ldots, \omega_{m+1})$

$$\int_a^b f(\lambda)d\mu = \underbrace{\sum_{j=1}^{m+1} \omega_j f(\xi_j)}_{I_{\text{Radau}}^{(m+1)}} + \underbrace{\frac{f^{(2m+1)}(\eta)}{(2m+1)!} \cdot \int_a^b (\lambda - a)\prod_{j=1}^m (\lambda - \xi_j)^2 d\mu}_{R_{\text{Radau}}^{(m+1)}}, \quad \eta \in (a, b).$$

In this case we have

$$R_{\text{Radau}}^{(m+1)} = \frac{(-1)^{2m+1}(2m+2)!\lambda^{-(2m+3)}}{(2m+1)!} \cdot \int_a^b (\lambda - a)\prod_{j=1}^m (\lambda - \xi_j)^2 d\mu \quad \leq \quad 0$$

due to the fact that $(\lambda - a)$ is nonnegative because a is a lower bound of the spectrum of $\boldsymbol{A}$. Thus $I_{\text{Radau}}^{(m+1)} \geq \int_a^b f(\lambda)d\mu$ and together we have

$$I_{\text{Gauß}}^{(m)} \quad \leq \quad \|\boldsymbol{x}^* - \tilde{\boldsymbol{x}}\|_2 \quad \leq \quad I_{\text{Radau}}^{(m+1)}.$$

Let us briefly recall how the nodes and weights are obtained in Gauß like quadrature rules. By use of the scalar product $\langle \phi | \psi \rangle_\mu := \int_a^b \phi \cdot \psi \, d\mu$, it is possible to define an orthonormal sequence of polynomials $\phi_1, \phi_2, \ldots$ with $\deg(\phi_j) = j$. This set of orthonormal polynomials satisfies a three term recurrence relationship:

$$\beta_j \phi_j(\lambda) = (\lambda - \alpha_j)\phi_{j-1}(\lambda) - \beta_{j-1}\phi_{j-2}(\lambda), \quad j = 1, \ldots, m$$
$$\phi_{-1}(\lambda) \equiv 0, \quad \phi_0(\lambda) \equiv 1 \tag{4.8}$$

if $\int_a^b d\mu = 1$ (by scaling $\boldsymbol{r}$ to $\|\boldsymbol{r}\|_2 = 1$). In Matrix form this can be written as

$$(\boldsymbol{J}^{(m)} - \lambda\boldsymbol{I})\boldsymbol{\phi}(\lambda) = -\beta_m\phi_m(\lambda)\boldsymbol{e}_m \tag{4.9}$$

where $\boldsymbol{\phi}(\lambda) = (\phi_0(\lambda), \ldots, \phi_{n-1}(\lambda))$ and

$$\boldsymbol{J}^{(m)} = \begin{pmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{m-2} & \alpha_{m-1} & \beta_{m-1} \\ & & & \beta_{m-1} & \alpha_m \end{pmatrix}.$$

From (4.8), we can see that these α's and β's are exactly the same coefficients as computed by the Lanczos algorithm (Algorithm 1.2). Multiplying the recurrence relation with $\phi_{j-1}(\lambda)$ and using the orthogonality constraints we get $\langle \phi_{j-1}(\lambda) \mid (\lambda - \alpha_j)\phi_{j-1}(\lambda)\rangle_\mu = 0$ and therefore

$$\alpha_j = \langle \phi_{j-1}(\lambda) \mid \lambda\phi_{j-1}(\lambda)\rangle_\mu = \int_a^b \lambda\phi_{j-1}^2(\lambda)\,d\mu$$

$$= \sum_{i=1}^n \lambda_i\phi_{j-1}^2(\lambda_i)\mu_i^2 = \sum_{i=1}^n \phi_{j-1}(\lambda_i)\mu_i \cdot \lambda_i\phi_{j-1}(\lambda_i)\mu_i$$

$$= \langle \phi_{j-1}(\boldsymbol{A}) \mid \boldsymbol{A} \cdot \phi_{j-1}(\boldsymbol{A})\rangle = \langle \boldsymbol{v}_{j-1} \mid \boldsymbol{A}\boldsymbol{v}_{j-1}\rangle.$$

Using $\|\phi_j(\boldsymbol{A})\|_2 = 1$, we get $\beta_j = \|(\boldsymbol{A} - \alpha_m\boldsymbol{I})\boldsymbol{v}_{j-1} - \beta_{j-1}\boldsymbol{v}_{j-2}\|_2$ in correspondence with Algorithm 1.2.

The eigenvalues of $\boldsymbol{J}^{(m)}$ (which are the zeroes of ϕ_m) are the nodes of the Gauß quadrature rule. The weights are the squares of the first elements of the normalized eigenvectors of $\boldsymbol{J}^{(m)}$.

In order to obtain the Gauß-Radau rule, we have to extend the matrix $\boldsymbol{J}^{(m)}$ in such a way that it has one prescribed eigenvalue $\xi_{m+1} = a$, i.e., we wish to construct ϕ_{m+1} such that $\phi_{m+1}(a) = 0$. From the recurrence relation (4.8), we have

$$0 \stackrel{!}{=} \beta_{m+1}\phi_{m+1}(a) = (a - \alpha_{m+1})\phi_m(a) - \beta_m\phi_{m-1}(a).$$

This gives

$$\alpha_{m+1} = a - \beta_m\frac{\phi_{m-1}(a)}{\phi_m(a)}$$

and evaluating (4.9) at $\lambda = a$ yields

$$(\boldsymbol{J}^{(m)} - a\boldsymbol{I})\boldsymbol{\phi}(a) = -\beta_m\phi_m(a) \cdot \boldsymbol{e}_m \quad \Leftrightarrow \quad (\boldsymbol{J}^{(m)} - a\boldsymbol{I})\boldsymbol{\delta} = \beta_m^2\boldsymbol{e}_m \qquad (4.10)$$

with $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_m)^T$ and

$$\delta_j = -\beta_m\frac{\phi_{j-1}(a)}{\phi_m(a)}, \quad j = 1, \ldots, m.$$

From these relations we can compute the tridiagonal matrix of the Gauß-Radau rule $\hat{\boldsymbol{J}}^{(m+1)}$ by first solving the tridiagonal system (4.10) and then using the last element of $\boldsymbol{\delta}$ to define $\hat{\boldsymbol{J}}^{(m+1)}$ via

$$\hat{\boldsymbol{J}}^{(m+1)} = \begin{pmatrix} \alpha_1 & \beta_1 & & & & \\ \beta_1 & \alpha_2 & \beta_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \beta_{m-2} & \alpha_{m-1} & \beta_{m-1} & \\ & & & \beta_{m-1} & \alpha_m & \beta_m \\ \hline & & & & \beta_m & a + \delta_m \end{pmatrix}$$

Note that we need not compute the eigenvalues and eigenvectors of these tridiagonal matrices $\boldsymbol{J}^{(m)}$ and $\hat{\boldsymbol{J}}^{(m+1)}$.

Theorem 4.5 *Let $(\xi_1, \ldots, \xi_k)$ and $(\omega_1, \ldots, \omega_k)$ be the nodes and weights of an k-point Gauß like quadrature rule and $\boldsymbol{J} = \text{tridiag}(\beta_j, \alpha_j, \beta_{j+1})$ with α_j and β_j being the coefficients from the corresponding three term recurrence. Further let f be an analytic function, then*

$$\sum_{j=1}^m \omega_j f(\xi_j) = \langle \boldsymbol{e}_1 \mid f(\boldsymbol{J})\boldsymbol{e}_1 \rangle.$$

Proof: As shown for example in [126], the weights ω_j can be computed as

$$\omega_j = \left(\frac{y_{1,j}}{\phi_0(\xi_j)}\right)^2, \quad j = 1, \ldots, m,$$

where $y_{1,j}$ is the first element of the jth eigenvector of $\boldsymbol{J}$. Since $\phi_0(\lambda) \equiv 1$, we have $\omega_j = (y_{1,j})^2 = (\boldsymbol{e}_1^T \boldsymbol{y}_j)^2$. Using $\boldsymbol{Y} := (\boldsymbol{y}_1 | \dots | \boldsymbol{y}_m)$ and $\boldsymbol{\Xi} = \text{diag}(\xi_1, \dots, \xi_m)$ we get

$$\sum_{j=1}^m \omega_j f(\xi_j) = \sum_{j=1}^m \boldsymbol{e}_1^T \boldsymbol{y}_j f(\xi_j) (\boldsymbol{y}_j)^T \boldsymbol{e}_1$$

$$= \left\langle \boldsymbol{e}_1 \, \middle| \, \sum_{j=1}^m \boldsymbol{y}_j f(\xi_j) (\boldsymbol{y}_j)^T \cdot \boldsymbol{e}_1 \right\rangle$$

$$= \left\langle \boldsymbol{e}_1 \, \middle| \, \boldsymbol{Y} f(\boldsymbol{\Xi}) \boldsymbol{Y}^T \cdot \boldsymbol{e}_1 \right\rangle$$

$$= \left\langle \boldsymbol{e}_1 \, \middle| \, f(\boldsymbol{J}) \, \boldsymbol{e}_1 \right\rangle.$$

∎

Since we had to scale our initial residual vector to have norm 1, i.e., we solved

$$\left\langle \beta_0^{-1} \boldsymbol{r} \, \middle| \, \boldsymbol{A}^{-2} \cdot \beta_0^{-1} \boldsymbol{r} \right\rangle = \beta_0^{-2} \cdot \left\langle \boldsymbol{r} \, \middle| \, \boldsymbol{A}^{-2} \boldsymbol{r} \right\rangle = \beta_0^{-2} \cdot \|\boldsymbol{x}^* - \tilde{\boldsymbol{x}}\|_2^2,$$

with $\beta_0 = \|\boldsymbol{r}\|_2$. We now obtain lower and upper bounds as

$$\beta_0 \sqrt{\left\langle \boldsymbol{e}_1 \mid (\boldsymbol{J}^{(m)})^{-2} \, \boldsymbol{e}_1 \right\rangle} \quad \leq \quad \|\boldsymbol{x}^* - \tilde{\boldsymbol{x}}\|_2 \quad \leq \quad \beta_0 \sqrt{\left\langle \boldsymbol{e}_1 \mid (\hat{\boldsymbol{J}}^{(m+1)})^{-2} \, \boldsymbol{e}_1 \right\rangle}$$

It should be remarked again that these bounds are only valid for symmetric positive definite matrices $\boldsymbol{A}$. Of course, we can always transfer a linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ into an equivalent system $\hat{\boldsymbol{A}}\boldsymbol{x} = \hat{\boldsymbol{b}}$ with $\hat{\boldsymbol{A}} = \boldsymbol{A}^T \boldsymbol{A}$ and $\hat{\boldsymbol{b}} = \boldsymbol{A}^T \boldsymbol{b}$. Then $\hat{\boldsymbol{A}}$ is s.p.d. but $\text{cond}(\hat{\boldsymbol{A}}) = \text{cond}(\boldsymbol{A})^2$. This limits the range of matrices we can handle to $\text{cond}(\boldsymbol{A}) \leq \epsilon^{-1/2}$. However, this restriction is not as important as it seems to be because we have to use a more precise arithmetic anyway, as we will see in Section 4.3.3.

4.3.3 Verified Computation

Of course, the results of the two preceding sections 4.3.1 and 4.3.2 assume that all computations are *exact* or at least valid bounds of the exact values. Since we cannot guarantee this by using floating-point arithmetic, we have to bring the tools from Section 3 into action.

In all subsequent algorithms, the variable *accu* represents a long accumulator in the sense of section 3.1. In particular, expressions of the form $accu = accu \pm x \cdot y$ denote exact accumulation of $x \cdot y$ in *accu*.

- *Decomposition Error* $\|\boldsymbol{L}\boldsymbol{U}^T - \boldsymbol{A}\|_2$

Suppose $\boldsymbol{L}$, $\boldsymbol{U}$ to be a nonsingular lower triangular matrices. Then Algorithm 4.1 computes a rigorous upper bound for $\|\boldsymbol{L}\boldsymbol{U}^T - \boldsymbol{A}\|_2$.

- *Smallest Singular Value*

Due to ideas of Rump [109], we compute the smallest singular value of a matrix $\boldsymbol{A}$ in two steps. First we factorize $\boldsymbol{A}$ approximately in a product of two triangular

$$e_1^{\text{row}} := e_2^{\text{row}} := \ldots := e_n^{\text{row}} := 0$$
$$e_1^{\text{col}} := e_2^{\text{col}} := \ldots := e_n^{\text{col}} := 0$$
for $i = 1, \ldots, n$
    for $j = 1, \ldots, i - 1$
        $accu := -a_{i,j}$
        for $k = 1, \ldots, j$
            $accu = accu + l_{i,k} * u_{j,k}$
        $e_i^{\text{row}} = e_i^{\text{row}} \triangle\triangle(|accu|)$
        $e_j^{\text{col}} = e_j^{\text{col}} \triangle\triangle(|accu|)$
    $accu := -a_{i,i}$
    for $k = 1, \ldots, i$
        $accu = accu + l_{i,k} * u_{i,k}$
    $e_i^{\text{row}} = e_i^{\text{row}} \triangle\triangle(|accu|)$
    $e_i^{\text{col}} = e_i^{\text{col}} \triangle\triangle(|accu|)$
$$e_{\max}^{\text{row}} := \max\{e_1^{\text{row}}, e_2^{\text{row}}, \ldots, e_n^{\text{row}}\}$$
$$e_{\max}^{\text{col}} := \max\{e_1^{\text{col}}, e_2^{\text{col}}, \ldots, e_n^{\text{col}}\}$$
return $\sqrt[\triangle]{e_{\max}^{\text{row}} \triangle e_{\max}^{\text{col}}}$

Algorithm 4.1: Compute a verified upper bound for $\|\boldsymbol{LU}^T - \boldsymbol{A}\|_2$ via the inequality $\|\boldsymbol{B}\|_2 \le \sqrt{\|\boldsymbol{B}\|_1 \cdot \|\boldsymbol{B}\|_\infty}$.

matrices, say $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$ (compare Section 1.4.2) and then we compute a lower bound of $\sigma_{\min}(\boldsymbol{T}_1\boldsymbol{T}_2^T)$ via[1] $\sigma_{\min}(\boldsymbol{T}_1\boldsymbol{T}_2^T) \ge \sigma_{\min}(\boldsymbol{T}_1)\sigma_{\min}(\boldsymbol{T_2})$.

Now we have to compute the smallest singular values of these triangular matrices. The basic idea is first to compute an approximation $\tilde{\sigma} \approx \sigma_{\min}(\boldsymbol{T})$ and then proving that $\boldsymbol{T}\boldsymbol{T}^T - \kappa\tilde{\sigma}^2\boldsymbol{I}$ is positive semidefinite, where κ is slightly less than one. In case of success, $\sqrt{\kappa}\tilde{\sigma}$ is a lower bound of $\sigma_{\min}(\boldsymbol{T})$. To decide whether the shifted $\boldsymbol{T}\boldsymbol{T}^T$ remains positive semidefinite, we try to compute its Cholesky factorization $\boldsymbol{L}\boldsymbol{L}^T$. Since this decomposition is usually not exact, we have to apply the following theorem from Wilkinson to guarantee that $\boldsymbol{L}\boldsymbol{L}^T$, if it exists, is not too far from $\boldsymbol{T}\boldsymbol{T}^T - \kappa\tilde{\sigma}^2\boldsymbol{I}$ so that the positive definiteness of $\boldsymbol{L}\boldsymbol{L}^T$ is sufficient for the smallest eigenvalue of $\boldsymbol{T}\boldsymbol{T}^T - \kappa\tilde{\sigma}^2\boldsymbol{I}$ to be nonnegative.

Theorem 4.6 *Let* $\boldsymbol{B}$, $\tilde{\boldsymbol{B}} \in \mathbb{R}^{n \times n}$ *be symmetric and* $\lambda_i(\boldsymbol{B})_{i=1}^n$, *respectively* $\lambda_i(\tilde{\boldsymbol{B}})_{i=1}^n$ *be the eigenvalues ordered by magnitude.*

Then from $\|\boldsymbol{B} - \tilde{\boldsymbol{B}}\|_\infty \le d$ *it follows that* $|\lambda_i(\boldsymbol{B}) - \lambda_i(\tilde{\boldsymbol{B}})| \le d$.

See [142] for a proof.

That is, if $\|\boldsymbol{L}\boldsymbol{L}^T - (\boldsymbol{T}\boldsymbol{T}^T - \kappa\tilde{\sigma}^2\boldsymbol{I})\|_\infty \le d$ then $\sigma_{\min}(\boldsymbol{T})^2 = \lambda_{\min}(\boldsymbol{T}\boldsymbol{T}^T) \ge \kappa\tilde{\sigma}^2 - d$. Thus, if d is a verified upper bound for $\|\boldsymbol{B} - \tilde{\boldsymbol{B}}\|_\infty$ and $\kappa\tilde{\sigma}^2 \ge d$ we have $\sigma_{\min}(\boldsymbol{T}) \ge \sqrt{\kappa\tilde{\sigma}^2 - d}$.

---

[1]In this computation of the smallest singular value hides the $O(n^3)$ effort which seems to be necessary to compute error bounds [27].

- *Verifying Positive Definiteness of $\boldsymbol{T}\boldsymbol{T}^{T} - \sigma^{2}\boldsymbol{I}$*

Suppose $\boldsymbol{T}$ to be a nonsingular lower triangular matrix. Then Algorithm 4.2 computes a rigorous lower bound for its smallest singular value.

$\sigma = 0.9 * approx\_smallest\_singular\_value(\boldsymbol{T})$
start:
$e_1 := e_2 := \ldots := e_n := 0$
for $i = 1, \ldots, n$
  for $j = 1, \ldots, i - 1$
    $accu := 0$
    for $k = 1, \ldots, j$
      $accu = accu + t_{i,k} * t_{j,k}$
    for $k = 1, \ldots, j - 1$
      $accu = accu - l_{i,k} * l_{j,k}$
    $l_{i,j} = \square(accu)/l_{j,j}$
    $e_i = e_i \triangle\triangle(|accu - l_{i,j} * l_{j,j}|)$
    $e_j = e_j \triangle\triangle(|accu - l_{i,j} * l_{j,j}|)$
  $accu := -\sigma^2$
  for $k = 1, \ldots, i$
    $accu = accu + t_{i,k} * t_{i,k}$
  for $k = 1, \ldots, i$
    $accu = accu - l_{i,k} * l_{i,k}$
  if $accu < 0$
    if $iter < max\_iter$
      $\sigma = 0.9 * \sigma$
      $iter = iter + 1$
      goto start
    else
      return failed
  $l_{i,i} = \sqrt{\square(accu)}$
  $e_i = e_i \triangle\triangle(|accu - l_{i,i}^2|)$
$e_{\max} := \max\{e_1, e_2, \ldots, e_n\}$
if $\sigma \triangledown \sigma \geq e_{\max}$
  return $\sqrt[\triangledown]{\sigma \triangledown \sigma \triangledown e_{\max}}$
else
  return failed

Algorithm 4.2: Compute a verified lower bound $\sigma \leq \sigma_{\min}(\boldsymbol{T})$ and a lower triangular matrix $\boldsymbol{L}$ with $\boldsymbol{L}\boldsymbol{L}^{T} - (\boldsymbol{T}\boldsymbol{T}^{T} - \sigma^{2}\boldsymbol{I}) \leq e_{\max}$.

- *Recursion Coefficients of the Gauß Quadrature Rule*

Given an approximate solution $\boldsymbol{x}$ we have to compute an enclosure of the residual $\tilde{\boldsymbol{v}}_0$ as described in Section 3.4. We then start a straightforward interval formulation

of the Lanczos algorithm (see Algorithm 4.3) to get enclosures of the α's and β's and thus for $\boldsymbol{J}^{(m)}$.

In the next section we will see that it is also possible to get an enclosure for the solution of equation (4.10) on page 62, i.e., an enclosure for δ_m and therefore we are also able to compute $[\hat{\boldsymbol{J}}]^{(m+1)}$.

Given $[\tilde{\boldsymbol{v}}]_0$, e.g. $[\tilde{\boldsymbol{v}}]_0 = \diamond(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0)$
$[\beta]_0 = \|[\tilde{\boldsymbol{v}}]_0\|_2$
$[\boldsymbol{v}]_0 = [\beta]_0^{-1} \cdot [\tilde{\boldsymbol{v}}]_0$
for $m = 1, 2, \ldots$
   $[\alpha]_m = \langle [\boldsymbol{v}]_{m-1} \mid \boldsymbol{A}[\boldsymbol{v}]_{m-1} \rangle$
   if $m = 1$
      $[\tilde{\boldsymbol{v}}]_m = (\boldsymbol{A} - [\alpha]_m \boldsymbol{I})[\boldsymbol{v}]_{m-1}$
   else
      $[\tilde{\boldsymbol{v}}]_m = (\boldsymbol{A} - [\alpha]_m \boldsymbol{I})[\boldsymbol{v}]_{m-1} - [\beta]_{m-1}[\boldsymbol{v}]_{m-2}$
   $[\beta]_m = \|[\tilde{\boldsymbol{v}}]_m\|_2$
   $[\boldsymbol{v}]_m = [\beta]_m^{-1} \cdot [\tilde{\boldsymbol{v}}]_m$

Algorithm 4.3: Interval Lanczos-algorithm.

• *Solutions of Symmetric Tridiagonal Systems*

A well known technique for computing solutions for interval linear systems is the interval Gauß algorithm (IGA). The shape of the solution set of a interval linear system can be fairly complicated, but since we use interval arithmetic we are only able to compute a multidimensional box that contains the true solution set. For general matrices it cannot be guaranteed to get a solution box that is near to the smallest box containing the true solution but it can be shown (see [38]) that the IGA produces optimal results, i.e. smallest in diameter, for tridiagonal interval systems with system matrices $[\boldsymbol{J}]^{(m)}$, $[\boldsymbol{J}]^{(m)} - a\boldsymbol{I}$, or $[\hat{\boldsymbol{J}}]^{(m+1)}$ respectively. However, we only sketch the algorithm here (Algorithm 4.4).

$[c]_1 = [\alpha]_1$
$[e]_1 = [b]_1$
for $i = 2, \ldots, m$
   $[c]_i = [\alpha]_i - [\beta]_{i-1}[\beta]_{i-1}/[c]_{i-1}$
   $[e]_i = [b]_i - [\beta]_{i-1}[e]_{i-1}/[c]_{i-1}$
$[x]_m = [e]_m/[c]_m$
for $i = m-1, \ldots, 1$
   $[x]_i = ([e]_i - [\beta]_i[x]_{i+1})/[c]_i$

Algorithm 4.4: Interval Gauß algorithm for $\text{tridiag}([\beta]_i, [\alpha]_i, [\beta]_{i+1}) \cdot [\boldsymbol{x}] = [\boldsymbol{b}]$.

5

High Performance Object Oriented Numerical Linear Algebra

> ❝ *I always knew C++ templates were the work of the Devil, and now I'm sure.* ❞
>
> Cliff Click, 1994

> ❝ *My desire to inflict pain on the compilers is large. They've been tormenting me for the last 8 years. Now is my chance to strike back!* ❞
>
> Scott Haney, 1996

An often used prejudice against modern object oriented programming techniques, is that object orientation is almost equivalent to low performance. At a first glance, if one compares the speed of a simple routine once written in, e.g., Fortran and once naively written in C++, this proposition seems to be true. Object oriented programming is massively based on abstraction, encapsulation of data, access restriction, and polymorphism. Most of these features imply a big organizational overhead because many decisions have to be done at runtime like dereferencing of

polymorphic types or allocation and deallocation of temporary variables. Besides these runtime penalties, also the compiler is unable to optimize around virtual function calls which prevents instruction scheduling, data flow analysis, loop unrolling, etc.

Thus, the situation — at least at the beginning of object oriented programming — was that though we had really nice, well readable, and excellent maintainable code, we had to pay with relatively low performance. Therefore and because most of the existent programs where coded in old Fortran versions, the scientific computing community decided to stay in the *stone age* of software technique.

Meanwhile, started in the middle of the 1990's, there have been made several proposals to improve the performance of object oriented programs. Using these techniques, it is possible to write highly abstract, object oriented programs that are comparable in speed with Fortran or C and are sometimes even faster [79, 139]. These improvements can be roughly splitted into two categories.

- The first kind tries to use the object orientation itself to remove redundancies, reduce code size and separate conceptually non-coupled program units. This isolation of performance critical code sections actually enables writing portable high performance codes. The key to this kind of structured programming is called *genericity*. We describe some of the most important concepts in Section 5.1.

- The second category aims to reduce the organizational overhead by relocating performance critical parts from run-time execution to compile-time execution. This technique can be viewed as a code generation system that removes, e.g., virtual function calls which are essentially required by polymorphic types. This compile-time polymorphism is called *static polymorphism* and has much more favorable optimization properties. The key technique is called *compile-time programming* and we describe some aspects in Section 5.2.

5.1   Genericity

The traditional approach writing basic linear algebra routines is a combinatorial affair. There are typically four precision types that need to be handled (single and double precision real, single and double precision complex), several dense storage types, a multitude of sparse storage types (the Sparse BLAS Standard Proposal includes 13 different sparse storage types [125]), as well as row and column orientations for each matrix type. On top of that, if one wants to parallelize these codes, there are several data distributions to be supported. To provide a full implementation one might need to code literally hundreds of versions of the same routine! It is no wonder the NIST implementation of the Sparse BLAS contains over 10 000 routines and an automatic code generation system [113].

This combinatorial explosion arises because with most programming languages, algorithms and data structures are more tightly coupled than is conceptually necessary. That is, one cannot express an algorithm as a subroutine independently from the type of data that is being operated on. Thus, although abstractly one might have only a single algorithm to be expressed, it must be realized separately for every

data type that is to be supported. As a result, providing a comprehensive linear algebra library — much less one that also offers high-performance — would seem to be an impossible task.

Fortunately, certain modern programming languages, such as Ada and C++, provide support for generic programming, a technique whereby an algorithm can be expressed independently of the data structure to which it is being applied. One of the most celebrated examples of generic programming is the C++ Standard Template Library (STL). Especially for numerical linear algebra, there is a library called the Matrix Template Library (MTL) which extends this generic programming approach to cover the needs of scientific computing [118].

The principal idea behind genericity is that many algorithms can be abstracted away from the particular data structures on which they operate. Algorithms typically need the abstract functionality of traversing through a data structure and accessing its elements. If data structures provide a standard interface for traversal and access, generic algorithms can be freely mixed and matched with data structures [117].

5.1.1 Data Structures: Containers

In object oriented numerics (OON), data structures like `records` in Pascal, `TYPE`s in Fortran or `struct`s in C together with a set of functions operating on this data are called containers. In C++, the equivalent to a container is called `class`. Containers basically consist of two parts: an internal representation which is only directly accessible by a set of authorized functions and a public interface which provides functions and methods to access the encapsulated data.
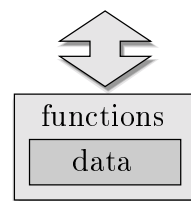
Let us consider the following example to demonstrate the basic ideas of containers. Suppose we want to design the concept of matrices. First we have to think about the storage types we wish to support. Assume we need

- banded matrices, i.e., we only store the diagonals between `lower_bandw` and `upper_bandw` and implicitly define the remaining elements to be zero,

- general sparse matrices, i.e., all nonzero elements are stored in a list with elements of type `(row, col, value)`, and

- dense matrices.

a) banded   b) sparse   c) dense
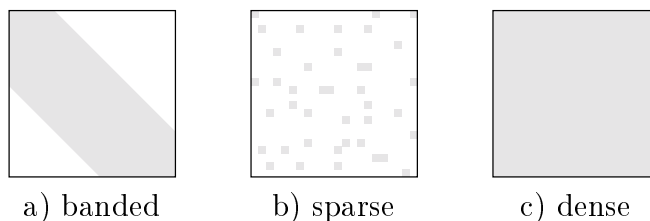
Figure 5.1: The storage types for our matrices

In this example, the internal representation might be

- a set of vectors storing the used diagonals (banded matrices),

- a list of triples `(row, col, value)` storing all nonzero elements (general sparse matrices), or

- a `num_rows` times `num_cols` sized memory block (dense matrices).

However, in the public interface we only need functions like `get_num_rows()` or `get_lower_bandwith()` to get informations about the shape of the matrices and most important we need access to the matrix entries. Traditionally only simple access functions like `get_entry_at( row, col )` were provided.

With this technique we have two possibilities to write algorithms which need access to the matrix entries: One is to traverse through all `num_rows` times `num_cols` elements (although most of them are zero) which is very slow. The other is to provide special algorithms with exact knowledge of the sparsity pattern for each matrix type which results in an enormous amount of code and therefore in an enormous amount of errors. Additionally, if we add a new storage type or modify an existing one, we have to add/modify a complete set of algorithms (compare Figure 5.2, left).
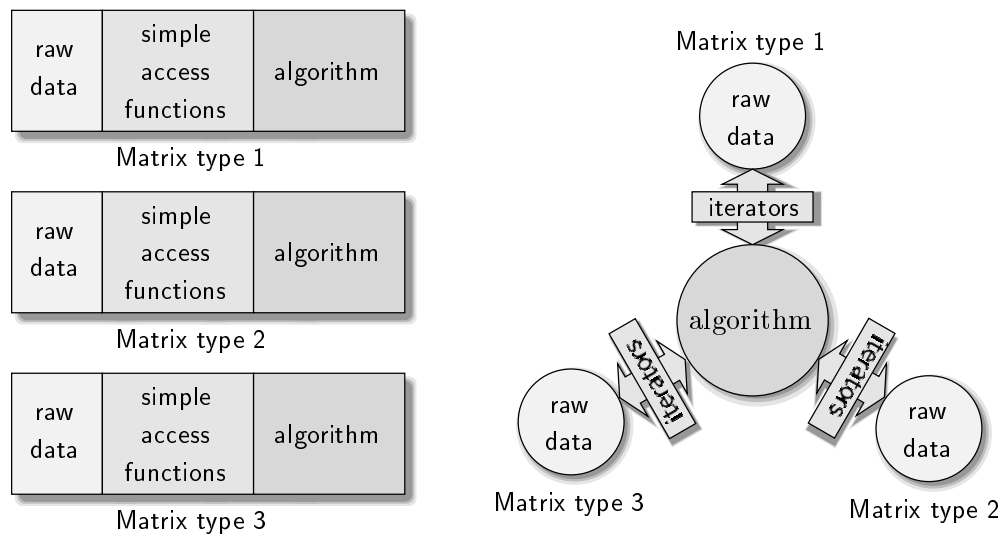
Figure 5.2: Separating the raw data structures from algorithms by the use of generic access and traversal functions — so called iterators.

Fortunately we can do much better by providing generic access via so called Iterators as will be described in the following section.

5.1.2 Traversing and Accessing Elements: Iterators

Instead of accessing the matrix data element-wise we conceptually design our matrices to be containers of containers (although they are typically not actual implemented in this way). For example we interpret a matrix to be a column vector with row vectors as element type (or vice versa). These vector containers, which haven't

allocated any storage but only refer to the memory of their parent matrix are called
iterators. Designing these Iterators to have a common public interface, enables us
to use only one routine for each algorithm (see Figure 5.2, right).

Thus, iterators can be used to traverse through a matrix in the following sense.
Each matrix container provides a datatype called, e.g., `row_2DIterator` (line 3)
and two functions returning special `row_2DIterator`'s, one is called `begin()` (line
4) and one is called `end()` (line 5) returning the first respectively last row vector of
the matrix.

```
1 class Matrix {
2 public:
3    typedef row_2DIterator my_row_iterator;
4    row_2DIterator begin();
5    row_2DIterator end();
6    ...
7 };
```

The `row_2DIterator` itself — which in this case is implemented as a `my_row_iterator`
— provides the datatype `1DIterator` (line 10) and again the two functions `begin()`
(line 11) and `end()` (line 12) here returning the first respectively last element of the
according row. Additionally, it provides the function `next()` (line 13) which returns
the successor of the row from which it is called.

```
 8 class my_row_iterator {
 9 public:
10    typedef 1DIterator my_element_iterator;
11    1DIterator begin();
12    1DIterator end();
13    my_row_iterator next();
14    int get_index();
15    ...
16 };
```

With this concept, we can ask the `Matrix` for the `begin()` `row_2DIterator` to
get its first row-vector. With the `next()` function we can traverse through all rows
until we end up with `end()`.

Beside these functions for traversing through the data structure we usually have
functions for accessing the stored information. In this case we have `get_index()`
which returns the row number (line 14).

Finally we have the `1DIterator` which doesn't refer to any subsequent itera-
tor. Here we only have one traversal function `next()` and two access functions
`get_index()` and `get_value()`.

```
17 class my_element_iterator {
18 public:
19    my_vector_iterator next();
20    int get_index();
21    double get_value();
22    ...
23 };
```
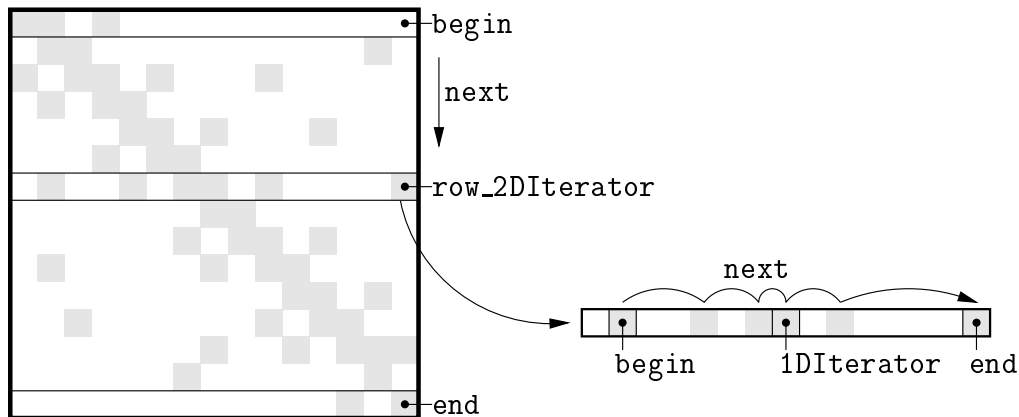
Figure 5.3: Traversing and accessing elements via iterators

```
 1 void print( Matrix A )
 2 {
 3   A::row_2DIterator       current_row;
 4   current_row::1DIterator current_entry;
 5
 6   current_row = A.begin();
 7   do {
 8     current_entry = current_row.begin()
 9     do {
10       cout << "[" << current_row.get_index()   << ","
11                   << current_entry.get_index() << "] = "
12                   << current_entry.get_value() << endl;
13       current_entry = current_entry.next();
14     } while( current_entry!=current_row.end() );
15     current_row = current_row.next();
16   } while( current_row!=A.end() );
17 }
```

Algorithm 5.1: A generic routine for printing arbitrary matrices.

In the same way we iterate through the rows of the matrix. With the `1DIterator` it is now possible to traverse through the elements of each `row_2DIterator` to access the row and column number and value of each nonzero entry, compare Figure 5.3.

In Algorithm 5.1 we illustrate this technique with the simple example for printing an arbitrary matrix. In line 3 we declare the variable `current_row` to be of type `A::row_2DIterator`, that is, `A`'s `row_2DIterator` type and in line 4 we declare `current_entry` to be a `1DIterator` of `current_row`. In line 6 we set `current_row` to be the first row of `A` and then iterate in the outer do-while-loop until `current_row` reaches the last row of `A` (line 16). For each `current_row` we set `current_entry` to be the first entry and iterate until `current_entry` reaches the last entry of `current_row`. Meanwhile we print `current_row.get_index()` (line 10) which is the row index, `current_entry.get_index()` (line 11): the column index and finally `current_entry.get_value()` (line 12): the value of the current entry.

This `print()` routine is capable to print all matrix types which provides these coupled 2D/1D iterator set. That is, each new matrix type only has to provide iterators with the same interface as `row_2DIterator` and `1DIterator` to be automatically printable via this function. For example, if we provide a matrix container with a `row_2DIterator` that actually represents the columns, we immediately can print the transposed of this matrix.

Similar to this print function, we can provide algorithms to multiply a matrix with a vector, to multiply two matrices, to add or copy matrices, and so on, only by writing one function for each algorithm.

5.1.3   A Point of View

Generalizing the above idea of printing a transposed matrix leads to so called *views* or *adaptors*. A view is a special container that actually has no own storage allocated but only refers to the memory of a *legal* container. The difference is, that a view provides a modified set of iterators. For example we could define a transposed view of a matrix via

```
1 class transposedMatrix : public Matrix {
2 public:
3   typedef row_2DIterator Matrix::col_2DIterator;
4   ...
5 };
```

From line 1 we see that a `transposedMatrix` is derived from a `Matrix` but exports its `row_2DIterator` as `Matrix`'s `col_2DIterator`. Consequently, if we access the rows of a `transposedMatrix` we actually get the columns of the original `Matrix`.

With this technique we can, e.g., also implement sub-matrix views by modifying the `begin()` and `end()` functions or diagonal views by modifying the `next()` function to return the next element on a given diagonal. All these operations are O(1), i.e., they need constant time and (nearly) no storage.
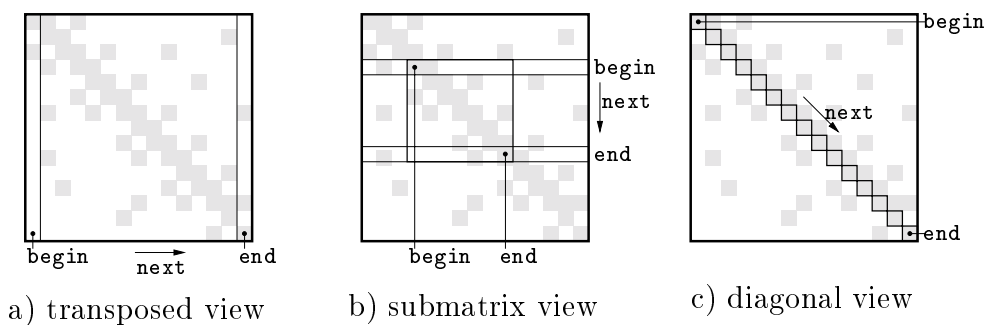

a) transposed view   b) submatrix view   c) diagonal view

Figure 5.4: Some views of a matrix

5.2   Two-Stage Programming

In this section we describe some advanced techniques to improve the performance of object oriented code [138]. Although they are hard to write and maintain and

increase compile times, they are great for library designers. Many of theses problems can be hidden from the end user. All ideas presented here, are based on a two-stage programming process. That is, beside the usual run-time execution, some parts of the program are evaluated at compile-time and thereby, e.g., generate specialized problem oriented code without polymorphism and thus without virtual function calls, or expressions without temporary variables, or even explicitly unrolled loops, etc. In C++ this technique was enabled by introducing so called *templates*, which makes it possible to write programs in a subset of C++ which are interpreted at compile-time. In this section we first introduce some basic concepts of compile time programming and then present two very powerful techniques to improve the performance of object oriented programs: namely expression templates and automatic self optimization.

5.2.1   Compile Time Programming

To demonstrate the power of template meta programming, i.e., routines that are evaluated completely by the compiler, we start with an example taken from [136].

```
template<int N_factorial>
class Value {};

template<int N>
class Factorial {
public:
  enum { value = N * Factorial<N-1>::value };
};

class Factorial<1> {
public:
  enum { value = 1 };
};

void foo()
{
  Value<Factorial<6>::value> dummy = Factorial<6>();
}
```

Using this `Factorial`-class, the value N! (factorial of N) is available at compile-time as `Factorial<N>::value`. How does it work? When `Factorial<N>` is instantiated, the compiler needs `Factorial<N-1>` in order to assign the enum `value`. So it instantiates `Factorial<N-1>`, which in turn requires `Factorial<N-2>`, requiring `Factorial<N-3>`, and so on until `Factorial<1>` is reached, where template specialization is used to end the recursion. The compiler effectively performs a for loop to evaluate N! at compile-time.

Thus compiling the above C++ program with `gcc factorial.cc` one gets the compiler error:

```
factorial.cc:17: conversion from 'Factorial<6>' to
                 non-scalar type 'Value<720>' requested
```

Although this technique might seem like just a cute C++ trick, it becomes powerful when combined with normal C++ code. In this hybrid approach, source code contains two programs: the normal C++ run-time program, and a template metaprogram which runs at compile-time. Template metaprograms can generate useful code when interpreted by the compiler, for example massively in-lined algorithm — such as an implementation of an algorithm which works for a specific input size, and has its loops unrolled. This results in large speed increases for many applications.

There are template-meta-program equivalents for most C++ flow control structures like 'if/else if/else', 'for', 'do/while', 'switch', or subroutine calls. See Tables 5.1 and 5.2 for some examples [136]. Of course these compile-time versions aren't very handy but we don't intend to write entire programs in this way. However, this kind of programming is worth the work in highly critical parts of an algorithm, e.g., in inner loops. Here we can partially unroll loops or reverse the order of computations to exploit memory caches or pipeline facilities (see Section 5.2.2).

C++ version	Template metaprogram version
| ```
if (condition) {
  statement1;
} else {
  statement2;
}
``` | ```
// Class declarations:
template<bool C>
class _if {};

class _if<true> {
public:
  static inline void _then() {
    statement1;
  }
};

class _if<false> {
public:
  static inline void _then() {
    statement2;
  }
};

// Replacement for 'if/else' statement:
_if<condition>::_then();
``` |

Table 5.1: A C++ if/else structure and its template metaprogram equivalent.

Another valuable field for using templates is to avoid run-time polymorphism. Let us first briefly describe this concept by using again our matrix example from the beginning of Section 5.1.1. Suppose we want to implement the simple access function get_entry_at(row, col).

C++ version	Template metaprogram version
| ```
int i = N;

do {
  statement(i);
  i--;
} while (i>0);
``` | ```
// Class declarations:
template<int I>
class _do_from {
public:
  static inline void _downto_1() {
    statement(I);
    _do_from<I-1>::_downto_1();
  }
};

class _do_from<0> {
public:
  static inline void _downto_1() {}
};

// Replacement for 'do/while' statement:
_do_from<N>::_downto_1();
``` |

Table 5.2: A C++ `do/while` structure and its template metaprogram equivalent.

```
class Matrix {
public:
  virtual double get_entry_at( int row, int col ) = 0;
};

class BandedMatrix : public Matrix {
public:
  virtual double get_entry_at( int row, int col );
};

class SparseMatrix : public Matrix {
public:
  virtual double get_entry_at( int row, int col );
};

class DenseMatrix : public Matrix {
public:
  virtual double get_entry_at( int row, int col );
};
```

Here we defined a polymorphic type `Matrix` because a `Matrix` can represent several specialized matrix types. Writing, e.g., a maximum norm function

```
double max_norm( const Matrix* A ) {
  double maxnorm = 0.0,
         rowsum;
  for( int row=0; row<A->num_rows(); ++row ) {
    rowsum = 0.0;
    for( int col=0; col<A->num_cols(); ++col )
      rowsum += abs( A->get_entry_at( row, col ) );
    maxnorm = max( maxnorm, rowsum );
  }
  return maxnorm;
}


Matrix* B = new SparseMatrix;
max_norm( B );
```

causes the run-time system to decide which particular `get_entry_at( row, col )` is to be called every time when a `Matrix::get_entry_at( row, col )` is requested. That is `num_rows`×`num_cols` times it has to be figured out to which particular matrix type `A` points to. This will ruin the performance of any matrix algorithm!

One way to replace this run-time polymorphism by a static polymorphism is to use structure parameters which encapsulate particular storage information:

```
class BandedMatrix {
  // Storage information for banded matrices
};


...


template<class T_structure>
class Matrix {
private:
  T_structure _data;
};


template<class T_structure>
double max_norm( Matrix<T_structure>& A ) { ... }

Matrix<BandedMatrix> B;
max_norm( B );
```

Here, the `Matrix` type is quasi polymorphic but only up to compilation. After interpreting the template metaprogram part, `Matrix<BandedMatrix>` is a static type and every call to one of its functions (e.g., `max_norm`) is non-virtual. The disadvantages of this solution are

- `Matrix` has to constantly delegate operations to the structure objects.

- The interface between the `Matrix` and the `T_structure` object must be identical for all structures.

- Interfaces must expand to accommodate every supported matrix structure. For example if we need banded matrices, every matrix type must provide a `get_lower_bandwith()` function.

Another approach to avoid virtual function calls are so called curiously recursive template patterns. Here we have a base class (`Matrix`) with a derived class (e.g., `SparseMatrix`) as template parameter. Inside the base class we have a function that explicitly converts the base class itself to be of the derived class type (lines 4-6). Each function simply delegates its execution to the corresponding leaf class by changing its type to `T_leafType` and calling an appropriate function of this leaf class (compare lines 7-9). Since, e.g., the code for a function `Matrix<T_leafType>::get_lower_bandw()` is only generated at compile-time if it is actually needed in the program, there is no need to write meaningless functions such as `DenseMatrix::get_lower_bandwith()` which would be called at runtime by `Matrix<DenseMatrix>::get_lower_bandwith()`.

```
 1 template<class T_leafType>
 2 class Matrix {
 3 public:
 4   T_leafType& asLeaf() {
 5     return static_cast<T_leafType&>( *this );
 6   }
 7   double get_entry_at( int row, int col ) {
 8     return asLeaf().get_entry_at( row, col );
 9   }
10 };
11
12 class SparseMatrix : public Matrix<SparseMatrix> {
13   double get_entry_at( int row, int col );
14 };
15
16 ...
17
18 template<class T_leafType>
19 double max_norm( Matrix<T_leafType>& A ) { ... }
20
21 Matrix<SparseMatrix> B;
22 max_norm( B );
```

5.2.2   Self Optimization

The bane of portable high performance numerical linear algebra is the need to tailor key routines to specific execution environments. For example, to obtain high performance on a modern microprocessor, an algorithm must properly exploit the associated memory hierarchy and pipeline architecture (typically through careful loop blocking and structuring). Ideally, one would like to be able to express high performance algorithms in a portable fashion, but there is not enough expressiveness in languages such as C or Fortran to do so. Recent efforts [29] have resorted to going outside the language, i.e., to code generation systems in order to gain this kind of flexibility. Another approach is the Basic Linear Algebra Instruction Set (BLAIS) [116], a library specification that takes advantage of certain features of the C++ language to express high-performance loop structures that can be easily reconfigured for a particular architecture.

To demonstrate the ability of C++ in generating environment dependent code, we give an example of a scalar product routine with partially unrolled loops. These blocks of unrolled loops could for example be distributed to different processors in a multiprocessor environment. In order not to complicate our example more than necessary, we assume that `BlockSize` divides `Size` (the size of the vectors to be multiplied). Furthermore we do not do anything special with the unrolled blocks than simply adding the results.

```
 1 template<int N>
 2 class cnt {};
 3
 4 template<int Size, int BlockSize, class InIter1, class InIter2,
 5          class Out>
 6 inline Out scalp( InIter1 x, InIter2 y, Out,
 7                   cnt<Size>, cnt<BlockSize> ) {
 8   return scalp( x, y, Out(), cnt<BlockSize>(), cnt<1>() )
 9        + scalp( x+BlockSize, y+BlockSize, Out(),
10                 cnt<Size-BlockSize>, cnt<BlockSize> );
11 }
12
13 template<class InIter1, class InIter2, class Out>
14 inline Out scalp( InIter1 x, InIter2 y, Out, cnt<1>, cnt<1> ) {
15   return *x * *y;
16 }
17
18 template<int BlockSize, class InIter1, class InIter2, class Out>
19 inline Out scalp( InIter1 x, InIter2 y, Out,
20                   cnt<0>, cnt<BlockSize> ) {
21   return 0.0;
22 }
```
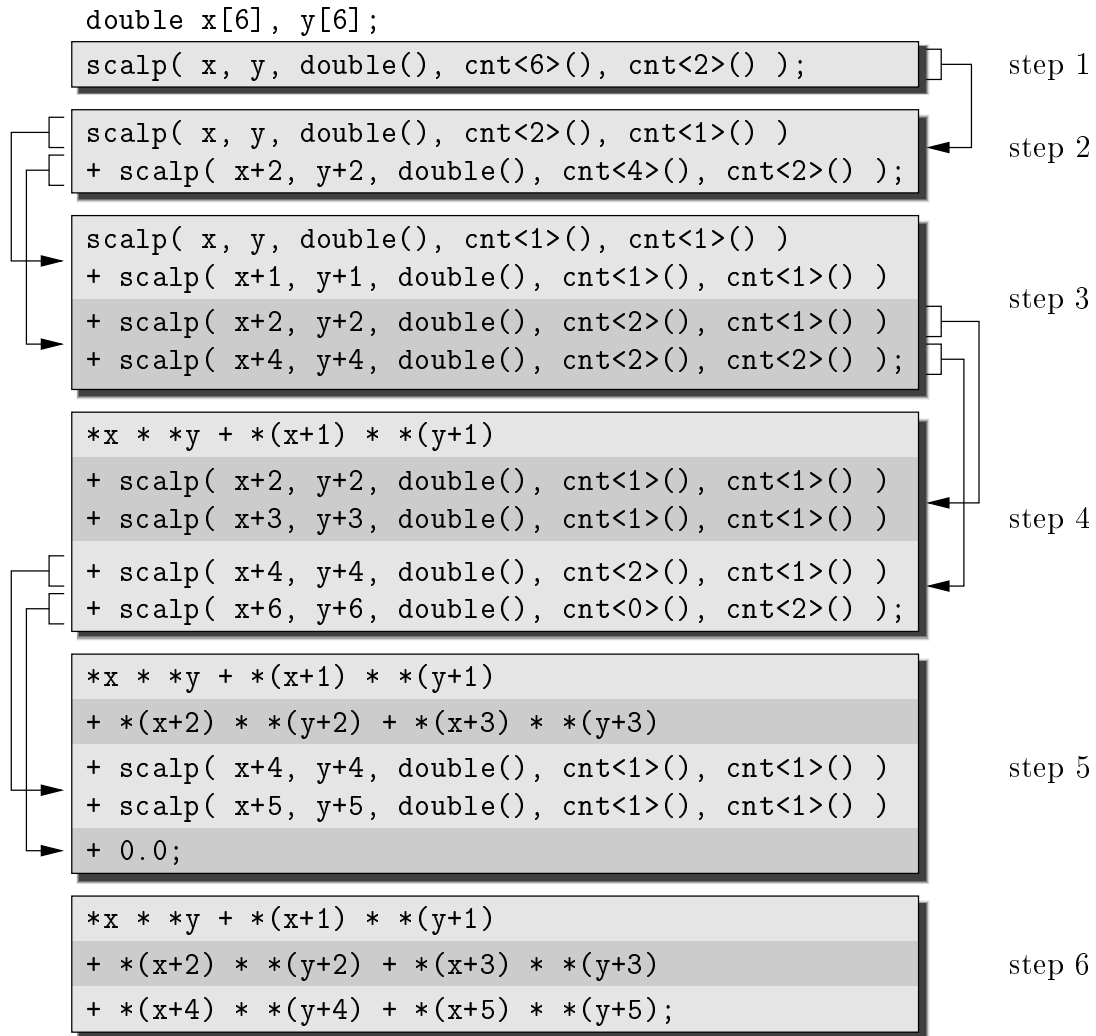
The counter class `cnt` (line 1,2) is used to encapsulate integers. In line 4 the template parameters `Size`, `BlockSize`, `InIter1/2` (used to iterate through the input vectors), and `Out` (type of the result) are introduced. Then we define the function `scalp` with result type `Out`. This function recursively computes the scalar product of the first block and that of the remaining blocks (lines 8 and 9). Two template specializations are used to terminate the recursion. The first actually computes scalar products with `BlockSize=1` (lines 13-16) and the second returns 0.0 if we are outside the vector range (lines 18-22). All functions are in-lined, i.e., the entire scalar product finally appears as one block of code without function calls.

Figure 5.5 illustrates what happens at compile-time. Each step represents a recursion depth when all possible recursion calls are executed. In step 6 we see the final code, which is generated by the template metaprogram. Note: `*(z+i)` denotes the dereferenciation of the iterator `z` at position `i`, i.e., the ith entry of the vector `z`.

5.2.3 Expression Templates

Expression templates are a C++ technique for evaluating vector and matrix expressions in a single pass without temporaries. This technique can also be used

```
double x[6], y[6];
scalp( x, y, double(), cnt<6>(), cnt<2>() );                      step 1

scalp( x, y, double(), cnt<2>(), cnt<1>() )
+ scalp( x+2, y+2, double(), cnt<4>(), cnt<2>() );               step 2

scalp( x, y, double(), cnt<1>(), cnt<1>() )
+ scalp( x+1, y+1, double(), cnt<1>(), cnt<1>() )
+ scalp( x+2, y+2, double(), cnt<2>(), cnt<1>() )                step 3
+ scalp( x+4, y+4, double(), cnt<2>(), cnt<2>() );

*x * *y + *(x+1) * *(y+1)
+ scalp( x+2, y+2, double(), cnt<1>(), cnt<1>() )
+ scalp( x+3, y+3, double(), cnt<1>(), cnt<1>() )
+ scalp( x+4, y+4, double(), cnt<2>(), cnt<1>() )                step 4
+ scalp( x+6, y+6, double(), cnt<0>(), cnt<2>() );

*x * *y + *(x+1) * *(y+1)
+ *(x+2) * *(y+2) + *(x+3) * *(y+3)
+ scalp( x+4, y+4, double(), cnt<1>(), cnt<1>() )               step 5
+ scalp( x+5, y+5, double(), cnt<1>(), cnt<1>() )
+ 0.0;

*x * *y + *(x+1) * *(y+1)
+ *(x+2) * *(y+2) + *(x+3) * *(y+3)                             step 6
+ *(x+4) * *(y+4) + *(x+5) * *(y+5);
```

Figure 5.5: In-lining a recursively defined blocked scalar product.

for passing expressions as function arguments.. The expression can be inlined into the function body, which results in faster and more convenient code than C-style callback functions. In benchmark results, one compiler evaluates vector expressions at 95-99% efficiency of hand-coded C using this technique (for long vectors). The speed is 2-15 times that of a conventional C++ vector class, see [137].

Expression templates solve the pairwise evaluation problem associated with operator-overloaded array expressions in C++. A naive implementation of

```
Vector<double> a, b, c, d;
a = b + c + d;
```

results in:

```
double* _t1 = new double[N];
for (int i=0; i<N; ++i )
  _t1[i] = b[i] + c[i];
double* _t2 = new double[N];
for (int i=0; i<N; ++i )
```

```
     _t2[i] = _t1[i] + d[i];
   for (int i=0; i<N; ++i )
     a[i] = _t2[i];
   delete [] _t2;
   delete [] _t1;
```

For small arrays, the overhead of `new` and `delete` results in very poor performance: about 1/10th that of C. For medium (in-cache) arrays, the overhead of extra loops and cache memory accesses hurts (by about 30-50% for small expressions). The extra data required by temporaries cause the problem to go out-of-cache sooner.

For large arrays, the cost is in the temporaries: all that extra data has to be shipped between main memory and cache. Typically scientific codes are limited by memory bandwidth (rather than flops), so this really hurts. For N distinct array operands and M operators, the performance is about

$$\frac{N+1}{3M}$$

that of C/Fortran. This is particularly bad for stencils, which have $N = 1$ (or otherwise very small) and M very large. It is not unusual to get 1/9 (5-point stencil — first order discretization on a 2D mesh), or even 1/24 (second order discretization on a 2D mesh) the performance of C/Fortran for big stencils.
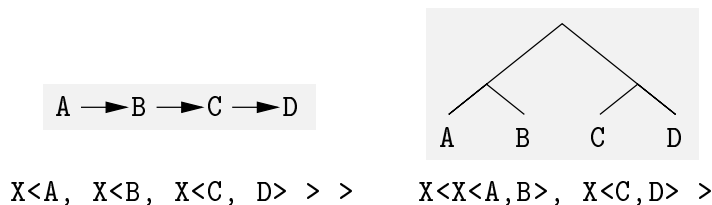
To avoid these temporary variables we have do delay the evaluation up to the point where the entire expression is parsed. That is, we have to parse the expression ourself and afterwards evaluate this parse tree in one pass. Fortunately this can be done inside C++. Let us shortly introduce the needed tools.

A class can take itself as a template parameter. This makes it possible to build linear lists or trees in the following sense:

```
template<class T1, class T2>
class X {};
```


X<A, X<B, X<C, D> > >   X<X<A,B>, X<C,D> >

Figure 5.6: Representing lists and trees with recursive template patterns.

The basic idea behind expression templates is to use operator overloading to build parse trees. For example:
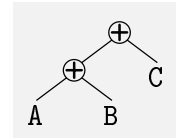
```
Array A, B, C, D;
D = A + B + C;
```

The expression `A+B+C` could be represented by a type such as

```
Op<plus, Op<plus, Array, Array>, Array>
```

Building such types is not hard:

```
template<class T>
Op<plus, T, Array>  operator+(T, Array)
{
  return Op<plus, T, Array>();
}
```

Then:

```
D = A + B + C;
  = Op<plus, Array, Array>() + C;
  = Op<plus, Op<plus, Array, Array> ,Array>();
```

Of course to be useful we need to store data in the parse tree (e.g. pointers to the arrays). Here is a minimal expression templates implementation for 1-D arrays. First, the `plus` function object:

```
class plus {
public:
  static double apply( double a, double b ) {
    return a+b;
  }
};
```

This class only provides the function `apply` which will be called to evaluate an object of type `Op<plus, Array, Array>()`.

The parse tree node:

```
template<class T_op, class T1, class T2 >
class Op {
public:
  Op( T1 a, T2 b ) {
    leftNode_  = a;
    rightNode_ = b;
  }

  double operator[]( int i ) {
    return T_op::apply( leftNode_[i], rightNode_[i] );
  }

private:
  T1 leftNode_;
  T2 rightNode_;
};
```

Here we have a constructor which simply stores its arguments in the private variables `leftNode_` and `rightNode_`. Additionally, the class `Op` provides an index operator which applies the operator `T_op` to the node-data and returns its result.

Now a simple array class:

```
class Array {
public:
  Array( double* data, int size ) {
    data_ = data;
    size_ = size;
  }

  template<class T_op, class T1, class T2 >
  operator=( Op<T_op, T1, T2> expr ) {
    for( int i=0; i<size_; ++i )
      data_[i] = expr[i];
  }

  double operator[]( int i ) {
    return data_[i];
  }

private:
  double* data_;
  int size_;
};
```

The `Array` constructor stores a pointer to the data and the number of elements. The `operator=` assigns the values of an expression (represented by its parse tree) to `data_`. Note that calling the index operator of `expr` actually causes the evaluation of the entire parse tree in one pass. Additionally `Array` itself provides an index operator which behaves traditionally, i.e., it simply returns the array entries.

And finally the `operator+` which actually does not add anything but constructs the parse tree:

```
template<class T>
Op<plus, T, Array> operator+( T a, Array b ) {
  return Op<plus, T, Array>(a,b);
}
```

Now see it in action:

```
int main() {
  double a.data[] = { 2, 3, 5, 9 },
         b.data[] = { 1, 0, 0, 1 },
         c.data[] = { 3, 0, 2, 5 },
         d.data[4];

  Array A(a.data,4),
        B(b.data,4),
        C(c.data,4),
        D(d.data,4);

  D = A + B + C;

  for (int i=0; i < 4; ++i)
    cout << D[i] << " ";
  cout << endl;
```

```
      return 0;
   }
```

Output: 6 3 7 15

See how `operator+` builds up the parse tree step by step:

```
D = A + B + C;
  = Op<plus, Array, Array>( A, B ) + C;
  = Op<plus, Op<plus, Array, Array>, Array>
        ( Op<plus, Array, Array>( A, B ), C );
```

Then it matches to template `Array::operator=`:

```
D.operator=( Op<plus, Op<plus, Array, Array>, Array>
                          ( Op<plus, Array, Array>( A, B ), C ) expr )
{
  for (int i=0; i < N.; ++i)
    data_[i] = expr[i];
}
```

See how `expr[i]` is evaluated successively by `Op<T_Op, Array, Array>::operator[]` calling `T_op::apply()`:

```
data_[i] = plus::apply( Op<plus, Array, Array>( A, B )[i], C[i] );
         = plus::apply( A[i], B[i] ) + C[i];
         = A[i] + B[i] + C[i];
```

...more or less. It's all clear now, right?

● *Excursion: Exact Scalar Product*

The exact scalar product, as introduced in Section 3.1, is often useful in critical computations. Computing residual vectors, for example, highly suffers from cancellation errors because many large numbers are added to a (hopefully) small sum. This error source can be completely avoided using exact scalar products. However, simulated in software, this routine is relatively slow compared to the execution time of an ordinary scalar product. Thus it would be advantageous if we could easily switch between the slower but exact computation and the fast approximative one (and maybe some precisions in between).

Usually in C++ libraries that provide operator overloading for matrix/vector expressions, we either have all scalar products evaluated exactly or none. Using expression templates enables us to introduce a Pascal-XSC [68] like notation for switching between exact and naive scalar products.

Pascal-XSC version:

```
r #*( b - A*x );
```

C++ version (compare [78]):

```
ExprMode::beginAccurate( RoundToNearest );
  r = b - A*x;
ExprMode::endAccurate( RoundToNearest );
```

6

vk — A Variable Precision Krylov Solver

> *"The first American Venus probe was lost due to a program fault caused by the inadvertent substitution of a statement of the form* `DO 3 I = 1.3` *for one of the form* `DO 3 I = 1,3`*"*
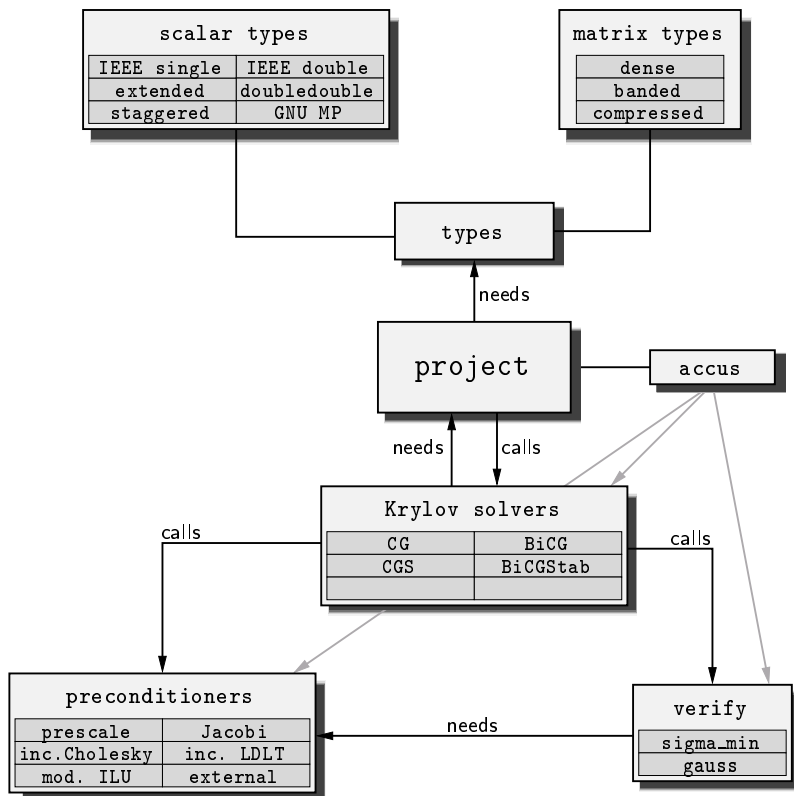>
> Jim Horning, 1979

Almost all ideas and concepts presented in this thesis are implemented in the **v**ariable precision **k**rylov solver **vk**. The program is written in the C++ programming language and makes extensive use of generic programming paradigms and compile-time programming. The main emphasis on writing **vk** was to produce an easily maintainable and extendable code (at least outside the kernel) while simultaneously providing an acceptable performance.

Several ideas on algorithms and data structures are taken from [28, 36, 39, 84]

6.1   Functional Description of vk

Figure 6.1 gives a quick overview over the main units of **vk**. The central object of **vk** is a `project`. A `project` stores all information about the problem to solve

and manages all necessary steps from reading the data, creating the preconditioner, solving the linear system, logging some interesting information, verifying the iterated solution, up to finally writing the result. Additionally it provides the data types for each arithmetic class (see Section 6.1.1) and hosts the multiplicators and accumulators for computing various scalar products in.

Figure 6.1: Functional structure of **vk**.

6.1.1   Variable Precision

In **vk**, nearly all computations can be performed in almost arbitrary precision ranging from IEEE single up to thousands of mantissa digits. For this purpose all computations were subdivided into several *arithmetic type-classes*. All variables of such classes are of equal data type and all scalar products (or matrix-vector products) inside one class are performed in the same precision (component products and accumulation). Table 6.1 lists the possible data types (compare Section 3.2), while Table 6.2 shows the arithmetic type-classes. To realize, e.g., an exact scalar product for IEEE double vectors in *internal* computations, one would have to set

$$\text{INTERNAL\_PREC} = -1, \ \text{INTERNAL\_PROD} = 2, \ \text{INTERNAL\_ACCU} = 67.$$

In fact, `INTERNAL_PROD = -3` would suffice, but is significantly slower due to additional data conversions.

Type	Name	type_id	Mantissa	Exp.
IEEE single precision	`single`	0	23 bit	8 bit
IEEE double precision	`double`	-1	53 bit	11 bit
Intel extended precision[a]	`extended`	-2	64 bit	15 bit
staggered with length 2	`doubledouble`	-3	106 bit	11 bit
staggered[b] with length $l = -id - 3$	`staggered<l>`	$-4, -5, \ldots$	$l \times 53$ bit	11 bit
GNU MP floating-point number with length $l = id$	`multiple<l>`	$1, 2, \ldots$	$l \times 64$ bit	31 bit

Table 6.1: Available data types.

[a]if supported by the used platform
[b]not supported by the standard version of **vk**

Symbol	Description
`PRECOND_FACT_PREC`	Datatype to store the preconditioner in (e.g. Cholesky factors).
`PRECOND_FACT_PROD`	Precision to compute products of type `PRECOND_FACT_PREC`×`PRECOND_FACT_PREC` with.
`PRECOND_FACT_ACCU`	Datatype to accumulate numbers of type `PRECOND_FACT_PREC` or `PRECOND_FACT_PROD` in.
`PRECOND_APPL_PREC`	Datatype to store the preconditioned search directions in.
`PRECOND_APPL_PROD`	Precision to compute products of type `PRECOND_APPL_PREC`×`PRECOND_APPL_PREC` or `PRECOND_APPL_PREC`×`INTERNAL_PREC` with.
`PRECOND_APPL_ACCU`	Datatype to accumulate numbers of type `PRECOND_APPL_PREC` or `PRECOND_APPL_PROD` in.
`INTERNAL_PREC`	Datatype to store internal used quantities in (α's, β's, residuals, auxiliary vectors, ...).
`INTERNAL_PROD`	Precision to compute products of type `INTERNAL_PREC`×`INTERNAL_PREC` with.
`INTERNAL_ACCU`	Datatype to accumulate numbers of type `INTERNAL_PREC` or `INTERNAL_PROD` in.
`SOLUTION_PREC`	Datatype to store the iterated solution in.
`SOLUTION_CALC`	Precision to compute the saxpy-operations with, used to update the solution.
`VERIFICATION_PREC`	Precision to compute the floating-point part of the verification step in.

Table 6.2: Adjustable arithmetic type-classes. The _PROD and _ACCU types can be used to realize the exact scalar product (see Section 3.1).
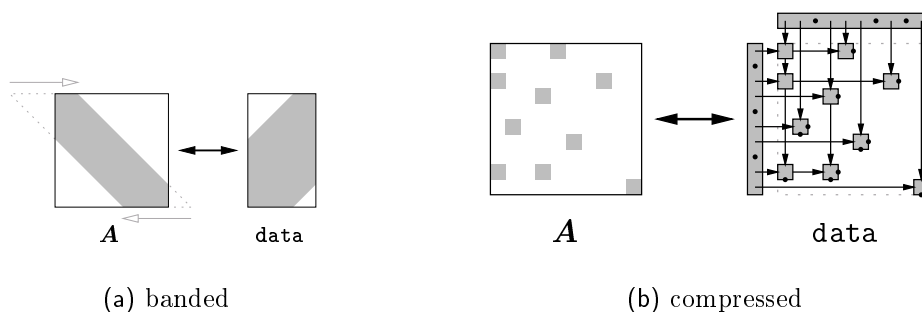
6.1.2 Matrix Types

Currently, **vk** supports three matrix types: dense, banded, and compressed.

dense A dense matrix is stored in a `dim` by `dim` memory block. This matrix type is mainly implemented for testing purposes since **vk** aims to solve sparse systems of equations.

banded A banded matrix actually is a mixture between dense and sparse matrices. The elements are stored in a dense `dim` by `bandwidth` memory block `data` and there is a simple mapping from matrix coordinates to memory coordinates and vice versa (inside the band):

$$
\begin{aligned}
\boldsymbol{A}(i,j) &\mapsto \texttt{data}[i, lower\_bw + j - i] \\
\texttt{data}[i,j] &\mapsto \boldsymbol{A}(i, i + j - lower\_bw)
\end{aligned}
$$

compressed In **vk**, compressed matrices are stored in a data structure which mainly consists of a doubly linked list and two vectors to find the first element in each row, respectively column. The elements of the list are of type `compressed_element` (compare Figure 6.2). With this matrix type, we can also perform transpose matrix-vector products. Without this feature we could save up to fifty percent of storage and the internal routines of this matrix container would be much simpler.

(a) banded   (b) compressed

Figure 6.2: Storage schemes for banded and compressed matrices.

```
template<class T>
class compressed_element {
  int col;
  int row;
  T value;
  T* next_in_row;
  T* next_in_col;
};
```

The actually used matrix type is automatically selected by construction to get maximum performance and minimum storage overhead.

There are 3 built-in linear systems for testing purposes and the possibility to read matrices stored in the MatrixMarket [85] format. The build in types are

Gregory/Karney 4.16 This matrix is out of a collection of matrices, edited by Gregory and Karney [52]. It is defined[1] via

$$
\mathsf{GK4.16} = \begin{pmatrix}
5 & -4 & 1 & & & & & \\
-4 & 6 & -4 & 1 & & & & \\
1 & -4 & 6 & -4 & 1 & & & \\
& \ddots & \ddots & \ddots & \ddots & \ddots & & \\
& & 1 & -4 & 6 & -4 & 1 & \\
& & & 1 & -4 & 6 & -4 \\
& & & & 1 & -4 & 5
\end{pmatrix}
\tag{6.1}
$$

This matrix is s.p.d. and its condition number is approximately n^4.

Hilbert Hilbert matrices are good examples for extremely high condition numbers at small dimensions. Usually these matrices are defined as $\boldsymbol{A}_{i,j} := 1/(i+j-1)$, with $i, j = 1, \ldots, n$. Scaling with the least common multiple of the numbers $1, \ldots, 2n - 1$ yields an integer matrix which can be stored exactly in IEEE double precision up to $n = 21$. Hilbert matrices are also s.p.d.

Gregory/Karney 4.20 This matrix again is out of Gregory/Karney's matrix collection. There, we have a parameter a which we set to 1. The matrix is symmetric but indefinite and is defined as

$$
\mathsf{GK4.20} = \begin{pmatrix}
-1 & 2 & 1 & & & & & \\
2 & 0 & 2 & 1 & & & & \\
1 & 2 & 0 & 2 & 1 & & & \\
& \ddots & \ddots & \ddots & \ddots & \ddots & & \\
& & 1 & 2 & 0 & 2 & 1 & \\
& & & 1 & 2 & 0 & 2 \\
& & & & 1 & 2 & -1
\end{pmatrix}.
\tag{6.2}
$$

6.1.3 Preconditioners

In **vk** we have several built-in preconditioners. Additionally, there is the possibility to import externally computed preconditioners from a file. To gain maximal performance, the preconditioner type must be selected at compile-time in order to enable the compiler to produce highly specialized code for preconditioned Krylov solvers. In particular we have the following types:

prescale is a Jacobi (see below) preconditioner but applied in advance to the linear system instead of applying it in each iteration.

Jacobi This preconditioner is realized by doing one step of the Jacobi iteration (compare Section 1.4.1).

Cholesky decomposition. This preconditioner is only applicable to s.p.d. systems (for this and the next two compare Section 1.4.2).

---

[1]The matrix results from a centered difference discretization of a fourth order differential equation, describing the bending line of weighted beam fixed at both ends [19].

$\boldsymbol{LDL}^T$ A generalized Cholesky preconditioner that works for arbitrary symmetric matrices.

$\boldsymbol{LDM}^T$ A modified (in the sense of Section 1.4.2) LU preconditioner.

external This kind of 'preconditioner' allows us to read a Cholesky factor ($\boldsymbol{L}$) or lower and upper triangular matrices ($\boldsymbol{L}$ and $\boldsymbol{U}$) and a permutation matrix ($\boldsymbol{P}$). These matrices are supposed to fulfill $\boldsymbol{LL}^T \approx \boldsymbol{A}$ or $\boldsymbol{LU}^T \approx \boldsymbol{PA}$, respectively. That is any kind of approximate triangular factorization (with pivoting) can be used.

Actually, only the prescale, Jacobi and Cholesky preconditioners can be computed in variable precision, i.e., with `PRECOND_CALC_*` $\neq$ `double`.

6.1.4 Krylov Solvers

The implemented Krylov solvers are CG, BiCG, CGS, and BiCGStab. All solvers support preconditioning, variable precision, and verification of their iterated solution. Due to generic programming, the latter two features are completely separated from the solver. Variable precision is a feature of the underlying matrix-vector arithmetic and the verification step can be considered as a separate post processing of the solution. Thus it is easy to transform any given Krylov solver to a verified Krylov solver with variable precision.

6.1.5 Verification

As mentioned in the section above, verification of iterated solutions with **vk** is completely separated from the solver itself. Both implemented verification algorithms only need the system matrix $\boldsymbol{A}$, the right hand side vector $\boldsymbol{b}$, an approximate triangular decomposition and obviously the approximate solution to be verified.

The supported verification algorithms are

sigma_min This is a verification method via *basic error bounds*, see Section 4.3.1.

gauss This verification method uses *improved error bounds* described in Section 4.3.2.

6.1.6 Output

Usually, there are two ASCII output files written by **vk**. One contains the iterated solution vector and the other is used for logging all important information about the system itself and the solving and verification process. The log-files are designed to be directly usable as `gnuplot` [40] input data files.

```
# vk-log-file
# filename                 : gk416_0000128_cg_Cholesky-none_[...].vk
# date                     : Wed May 17 10:55:15 2000
# precond calc precision   =    53 Bit (double)
# precond calc accu length =    53 Bit (double)
# precond calc prod length =    53 Bit (double)
# precond apply precision  =    53 Bit (double)
# precond apply accu length = 4288 Bit (multiple<67>)
# precond apply prod length =  128 Bit (multiple< 2>)
# internal precision       =    53 Bit (double)
```

```
# internal accu length    =  4288 Bit (multiple<67>)
# internal prod length    =   128 Bit (multiple< 2>)
# solution precision      =    53 Bit (double)
# solution calc prec      =   128 Bit (multiple< 2>)
# maxCount                = 128
# max. residual norm      = 2.46204e-12
# max. error norm         = 1e-05
# algorithm               : cg
# preconditioning         : Cholesky (none) (0  0:00:00.00 sec)
# matrix                  : gk416_0000128
# exact solution          : none
# dimension               = 128
# nnz                     = 634 (3.87%)
# lower bandwidth         = 2
# upper bandwidth         = 2
# min. singular value     = 2.46204e-07  (0  0:00:00.00 sec)
# decomposition error     = ---
# ||A*x-b|| (x=exact)     = ---
#------------------------------------------------------------------
# iter        time      res.norm      abs.error  ub of res.nrm.
#------------------------------------------------------------------
   1            0     1.1456e-08          ---         1.5628e-08
   2         0.01     2.2175e-19          ---         1.6908e-09
# verification failed ( upper error bound \in [0.0068674,0.0068674])
   3         0.02     5.6131e-31          ---         3.3959e-09
# verification failed ( upper error bound \in [0.013793,0.013793])
# STAGNATION after 3 steps.
# needed  3 iterations to reach ||r||: 5.61e-31 (updated)
# verified upper error bound      : 0.0069
# EOF (0  0:00:00.02 sec)
```

6.2 Using vk

Since **vk** makes extensive use of the two stage programming paradigm (see Section 5.2), several quantities have already to be known at compile-time. This enables the compiler to produce highly specialized code, e.g., for the particular data types we want to use. Therefore, using **vk** consists of two steps. First we have to compile an appropriate executable (see Section 6.2.1) and secondly we need to run the executable with proper command line options (Section 6.2.2).

6.2.1 Compiling vk

Before compiling **vk**, one have to choose the precision for each arithmetic class (compare Section 6.1.1) as well as the preconditioner. A preconditioner is selected by setting PRECOND_TYPE to an appropriate *prec_id*, i.e., by adding the definition -D PRECOND_TYPE=*prec_id* to the compile command line. Table 6.3 shows all supported preconditioners with according *prec_id*s. Similarly, the data type (Table 6.1) has to

Name	*prec_id*	Name	*prec_id*
none	0	LDL^T	4
prescale	1	LDM^T	5
Jacobi	2	external	6
Cholesky	3		

Table 6.3: All preconditioners supported by **vk**.

be selected for each arithmetic type-class, by adding -D *class_name = type_id* to the compiler options. The default precision for any class is IEEE double (*type_id*=-1).

If one arithmetic class is adjusted to have the data type `doubledouble` you need to link the doubledouble library [16] and add the options:

```
-L path/to/doubledouble/library/
-l doubledouble
-I path/to/doubledouble/includes/
-D DD_INLINE -D x86 -m 486 -D VK_USE_DOUBLEDOUBLE
```

You always need to link the `Profil` and `Bias` libraries[2] [70]:

```
-L path/to/profil_bias/library/
-l Profil -l Bias
-I path/to/profil_bias/includes/
```

Finally you need

```
-L path/to/gmp/library/
-l gmp
-I path/to/gmp/includes/
-I path/to/gmp++/source/
```

to tell the compiler where to find the `gmp++` library[3] which provides the `multiple<N>` data type.

All together the command line for compiling **vk** should look as follows:

```
g++ -o my_vk vk.cc -I ..
      options for doubledouble
      options for profil/bias
      options for gmp/gmp++
      define for the preconditioner
      defines for the data types
```

Assume we need a Krylov solver with Cholesky preconditioner. The Cholesky factorization shall be computed in a higher precision, say in `doubledouble`, but stored in the IEEE double format. Additionally we need all internal scalar products to be exact ones. Then we may compile **vk** via

```
g++ -o my_vk vk.cc -I ..
      -L ~/lib/doubledouble -l doubledouble
      -I ~/include/doubledouble
      -D DD_INLINE -D x86 -m 486 -D VK_USE_DOUBLEDOUBLE
      -L ~/lib/profil_bias -l Profil -l Bias
      -I ~/include/profil_bias
      -L /usr/lib -l gmp
      -I /usr/include/gmp
      -I ~/source/gmp++
```

---

[2]Note that the original versions produces oodles of warnings, but should compile anyhow.
[3]This library is obtainable from the author of this thesis.

```
-D PRECOND_TYPE      =  3
-D PRECOND_FACT_PROD = -3
-D PRECOND_FACT_ACCU = -3
-D INTERNAL_PROD     =  2
-D INTERNAL_ACCU     = 67
```

6.2.2 Command Line Options

Table 6.4 shows all currently supported command line options

Switch	Arg. type	Description
`-h, --help`	*none*	prints a short help message (similar to this table).
`-V, --version`	*none*	prints a description of the executable.
`-m, --matrix`	*string*	filename of the matrix or 'gk416_n, 'gk420_n, or 'hilbert_n.
`-r, --rhs`	*string*	filename of the right hand side vector or 'set' to set $b = Ax$, where x is the comparative solution.
`-c, --compsol`	*string*	file-name of the comparative solution.
`-a, --algorithm`	*string*	'CG', 'BiCG', 'CGS', or 'BiCGStab'.
`-p, --preconditioner`	*string*	option for the preconditioner (e.g., the file-name for 'external').
`-n, --maxcount`	*int*	maximum number of iterations.
`-e, --eps`	*float*	maximum residual norm (or maximum error norm, if verification is enabled).
`-v, --verify`	*none*	enable verification
`-w, --write`	*string*	name of the file to store the iterated solution in.
`-l, --logfile`	*string*	name of the file to store the logging messages in (may be 'auto' for automatically creating a file name).
`-q, --query`	*string*	Used for communication between **vk** and **xvk**

Table 6.4: Command line options for **vk**

6.3   xvk — A Graphical User Interface

Because there are a lot of compile-time and run-time options, include paths, libraries, and so on, it might be a little difficult to get **vk** running. To assist you with that, there is a graphical user interface (GUI) for **vk** called **xvk**. It is written in C and uses the X-toolkit GTK [53].

The GUI of **xvk** is organized as a notebook with 4 pages: 'Compile-Time Constants', 'Compiler Options', 'Run-Time Parameters', and 'Browse Logfiles' (the latter actually is in progress).

The page 'Compile-Time Constants' (see Figure 6.3) allows you to set all compile-time constants to appropriate values. The arithmetic type-classes (compare Table 6.2) are subdivided into 5 categories. For each type-class, there is a little menu that lets you choose a numerical data type (compare Table 6.1). If you select `multiple<N>`, the 'Precision' entry becomes sensitive. You may adjust `N` here.

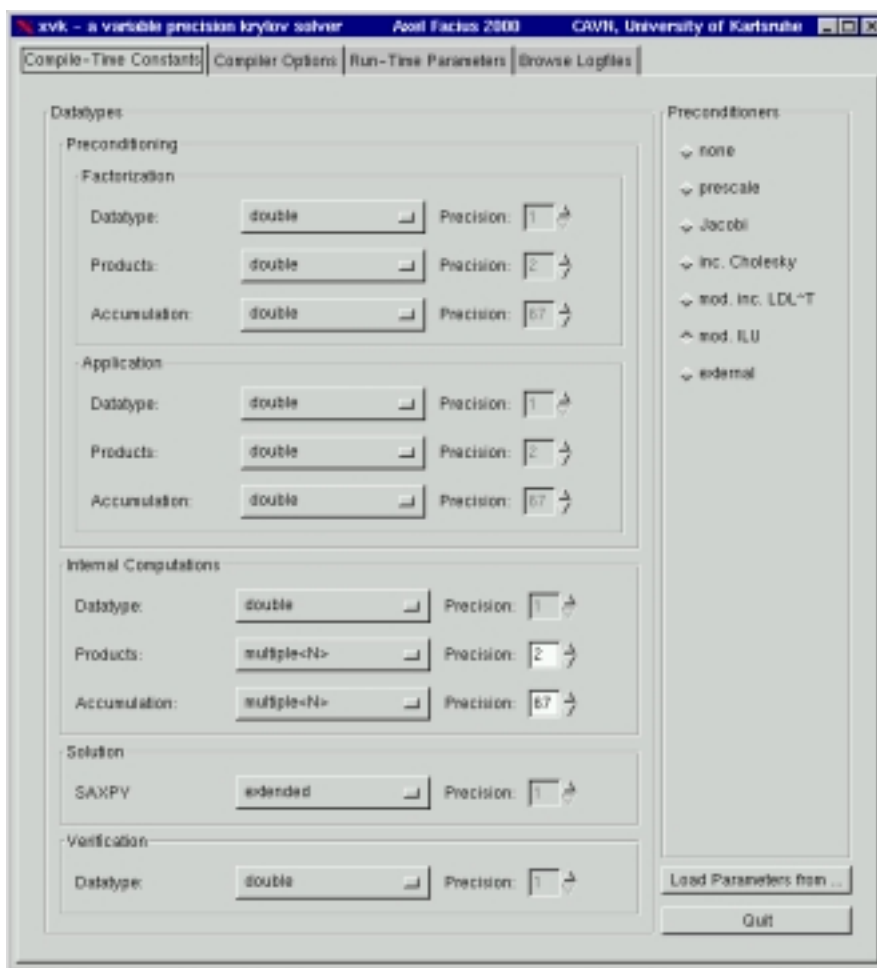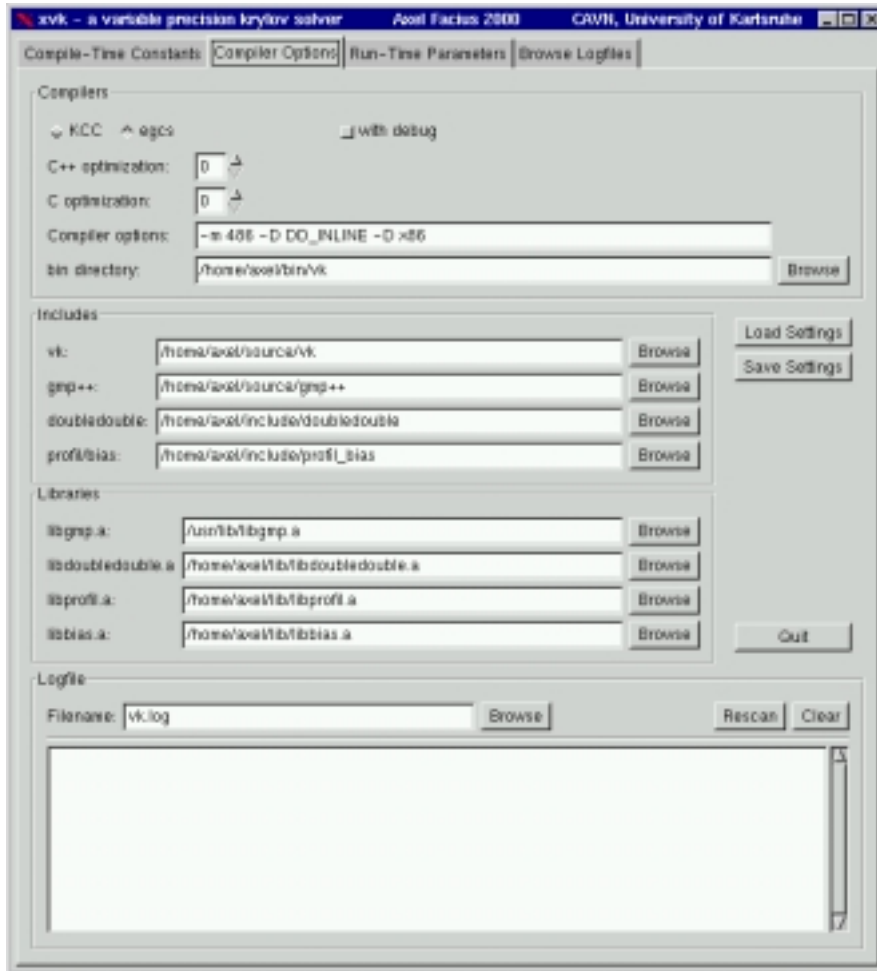Additionally, you can select the preconditioner type, by simply clicking on its radio button.

Figure 6.3: **xvk**: Compile-time constants.

On the page 'Compiler Options' (see Figure 6.4) all remaining options of the compiler command line can be adjusted. Since it shouldn't be necessary to change

these settings after installation, it is possible to save them on disk in order to get them preloaded every time you start **xvk**.

Additionally, at the bottom of this page, there is a window displaying the output of all compiler calls initiated by **xvk**. This output is also written into a logfile.
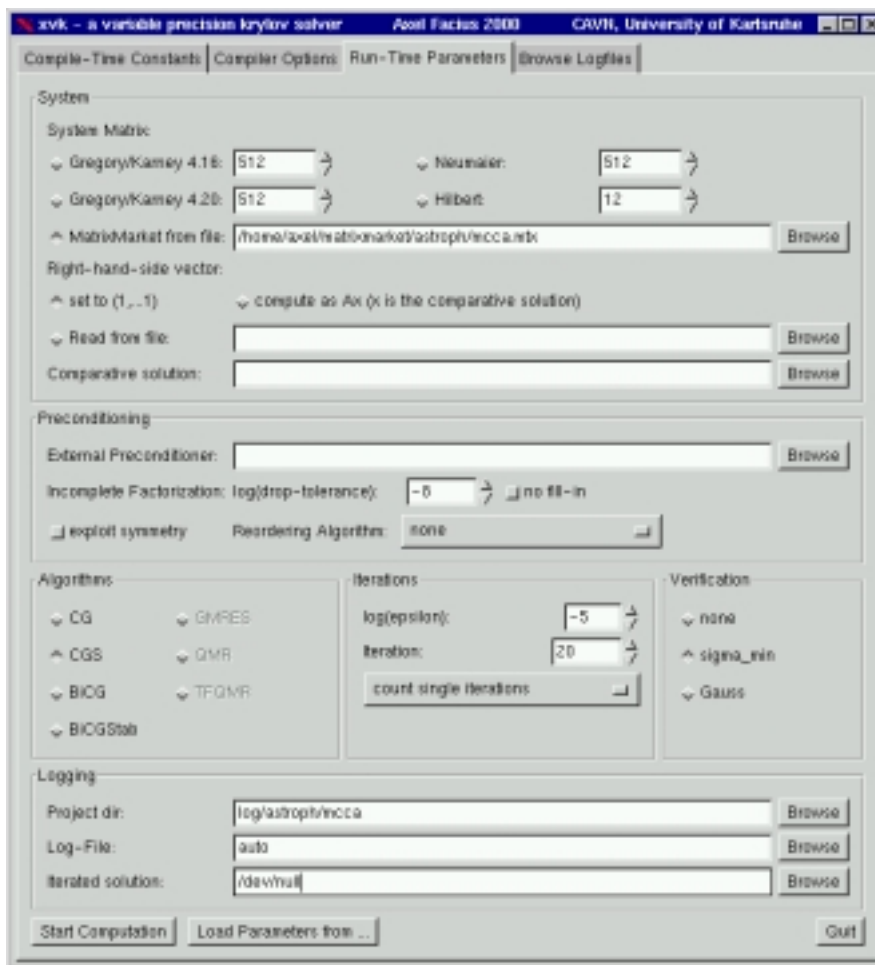

Figure 6.4: **xvk**: Compiler options.

The notebook page 'Run-Time Parameters' (see Figure 6.5) allows you to adjust all options understood by **vk** and beyond it you can choose some additional options only provided by **xvk**. The latter are computed by calling MATLAB preliminary to **vk**.

In particular you may

- select the system matrix either as one of the built-in types or by specifying a file in the MatrixMarket format (see [85]) or

- select the right-hand-side vector either by setting all its components to one, by computing it according to the comparative solution (must be given in this case), or by specifying a file in the MatrixMarket format and

- select a comparative solution,

- pass some options to the compiled-in preconditioner,

- select a reordering algorithm,

- choose a Krylov algorithm,

- specify the stopping criteria(s),

- set the verification mode, and finally

- adjust some project properties

Figure 6.5: **xvk**: Run-time parameters.
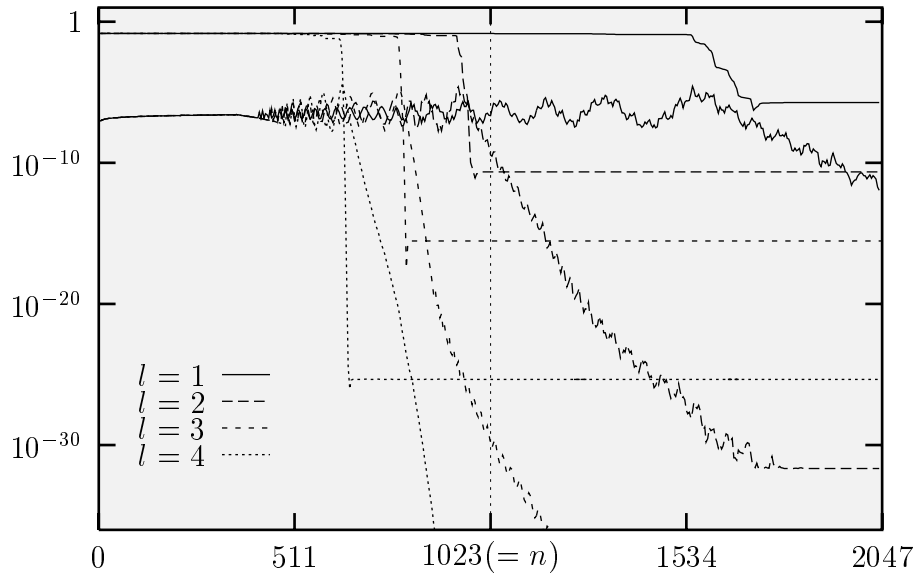
7

Computational Results

" Numerical subroutines should deliver results that satisfy simple, useful mathematical laws whenever possible. "
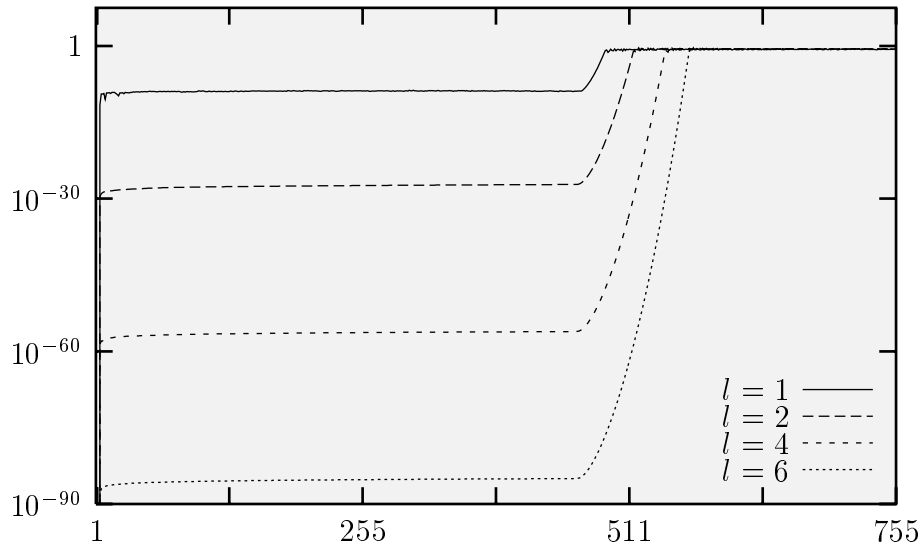
Donald E. Knuth, 1981

7.1 Level of Orthogonality

As a first example of the effectiveness of higher precision arithmetic, we solved again the system GK4.16(1023) (compare Section 2.3.2, Figure 2.4). The residual norms and error norms, achieved by using a staggered precision arithmetic, are plotted in Figure 7.1. The letter l denotes the *staggered length*, i.e., the number of floating-point numbers defining a staggered number (see Section 3.2.1). The case $l = 1$ corresponds with Figure 2.4. Since the staggered arithmetic is simulated in software, it cannot compete with the built-in double arithmetic ($l = 1$) in computing time. However, despite getting more accurate solutions, (which might be unnecessary for practical problems) we observe a significant saving in the number of iterations.

In Figure 7.2 we show the level of orthogonality of the new residual-vector $\boldsymbol{r}_{m+1}$ to the previous ones: $\max_{k=1}^{m}\{\langle \boldsymbol{r}_k \mid \boldsymbol{r}_{m+1}\rangle/(||\boldsymbol{r}_k||_2||\boldsymbol{r}_{m+1}||_2)\}$. Beside the expectedly *better* orthogonality at the beginning, the loss of orthogonality is not delayed very much. However, this little improvement is sufficient to give a significantly better convergence.

Figure 7.1: The Euclidean norms of the residuals (oscillating) and errors (more or less piecewise constant) during solving the GK(1023) system with staggered length l from 1 to 4.

Figure 7.2: The level of orthogonality during solving the GK(1023) system with staggered length $l \in \{1, 2, 4, 6\}$.
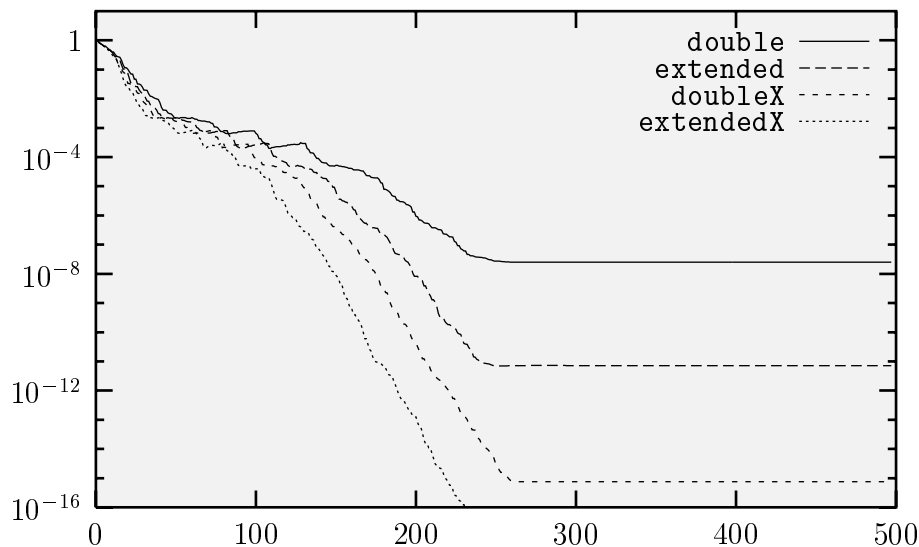
7.2 High Precision and Exact Scalar Products

In this example, we demonstrate the effect of high precision and exact scalar products. The used linear system is called `fidap009` and is described in the appendix as Matrix A.5. We solved this linear system with a preconditioned Conjugate Gradient solver. For the preconditioner we used an incomplete Cholesky factorization with drop-tolerance 10^{-10}. For several larger droptolerances, i.e. more sparse preconditioners, we got no convergence, neither with `double`, nor with `extended` arithmetic. The used arithmetics were:

`double`	IEEE double precision,
`doubleX`	IEEE double precision with exact scalar products,
`extended`	Intel's extended precision format, and
`extendedX`	Intel's extended precision format with exact scalar products[a]

---

[a]This accumulator needs approximately 32 kbyte. Since **vk** provides a central accu management, only one accumulator of this size is allocated.
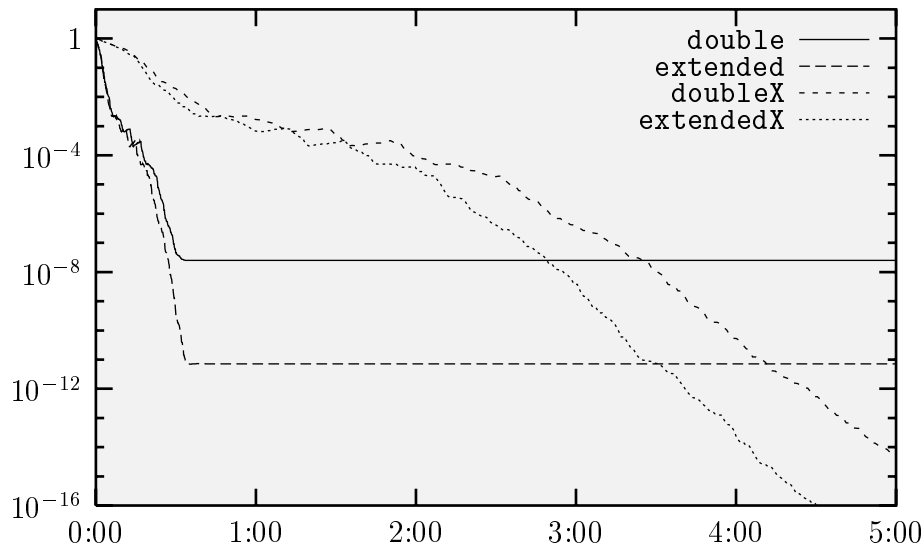
Figure 7.3 shows the relative error norms vs. number of iterations. As we can see, the higher precision used for accumulation, results in faster convergence and increased accuracy.

Figure 7.3: Solving a `fidap009` system (Matrix A.5) with an incomplete Cholesky preconditioned `CG` solver. The curves represent the relative error norms vs. number of iterations achieved by computing with differently precise arithmetics (IEEE double resp. Intel's extended with standard scalar products (`double`/`extended`) and with exact scalar products (`doubleX`/`extendedX`.)
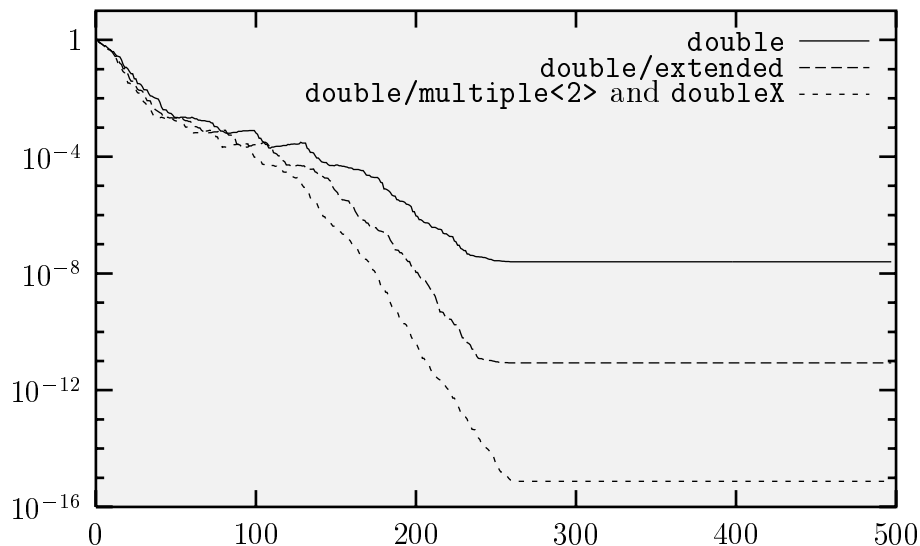
Taking into consideration, that exact scalar products, sufficiently supported in hardware, need not to be slower than ordinary scalar products, there it is simply no reason not to utilize this technique. However, presently, this operation is simulated in software only and therefore is relatively slow compared to the built-in

arithmetic. Figure 7.4 shows the same experiments as Figure 7.3 but now plotted against computing time (on a Pentium II, 400 Mhz).

Figure 7.4: This figure shows the same experiments as Figure 7.3 but now plotted against computing time (in sec) instead of iteration counts.

Of course, for practical problems, the exact scalar product often provides much more precision than is actually needed. To get an idea on how many precision would suffice, we solved the `fidap009` system with basic data type `double` and different accumulator precisions.

Figure 7.5: This figure shows the same experiments as Figure 7.3 but now we used different scalar products while leaving the basic data type fixed at `double`.

Namely, we used accumulation in `double`, `extended`, `multiple<2>` (128 bit mantissa length, see Section 3.2.2), and exact accumulation. The results are shown in Figure 7.5. The curves that corresponds with exact accumulation and `multiple<2>`-accumulation differ so little that they appear as a single line. That is, in this particular case, 128 bit mantissa length was sufficient for almost error free accumulation.

7.3 Beyond Ordinary Floating-Point Arithmetic

While in the latter section we only saved iterations by using a more precise arithmetic, we now show that there are examples that are not solvable in standard floating-point arithmetics at all or at least speed up by using higher precision.

For this purpose, we solved a linear system with the Hilbert matrix of dimension 13. To get a simple stopping criterion, we first computed a verified solution that is guaranteed to have at least 16 correct decimal digits and then stopped each of the following iterations, when the approximated solutions coincide with the verified solution within the first five digits.

Each experiment was carried out twice, once with a (complete) Cholesky preconditioner and once without preconditioning. The results for various arithmetics are displayed in Table 7.1. If there are two entries in the 'Arithmetic' column (separated by a slash), then the first denotes the data-type and the second is the accumulation precision. If there is only one arithmetic data type given, then all computations are performed with this type.

Arithmetic	No Precond.		Cholesky	
	iter	*time*	*iter*	*time*
`double`	>130	(—)	>130	(—)
`double/extended`	>130	(—)	>130	(—)
`extended`	>130	(—)	>130	(—)
`double/multiple<2>`	89	(0.13)	3	(<0.01)
`double/exact`	89	(0.13)	3	(<0.01)
`extended/exact`	37	(0.04)	3	(<0.01)
`multiple<2>`	23	(0.04)	3	(0.01)
`multiple<4>`	16	(0.02)	3	(0.01)
`multiple<5>`	13	(0.02)	3	(0.01)

Table 7.1: This table shows the number of iterations and the computing time, needed to obtain at least 5 correct decimal digits in the iterated solution. We stopped the process at a maximum of 130 steps (displayed in gray letters).

As we can see, the *smallest* arithmetic enabling convergence, is IEEE double with 128 bit accumulation (`double/mutiple<2>`). Using `extended` precision (with `mutiple<2>` accumulation), significantly speeds up the computation in the non-preconditioned case and further increasing the precision saves up to 85% computing time. With 320 bit mantissa length `mutiple<5>`, we match the *exact precision* property of convergence after at most n steps.

Though we used a (complete) Cholesky preconditioner in the right column, i.e. a direct solver in each iteration, we also got convergence only with at least 128 bit accumulation. Further increasing the precision does not save more iterations and consequently does not save computing time. However, even with this direct solver in each step, we need 3 iterations to get 5 correct digits.

7.4   Does Higher Precision Increase the Computational Effort?

Inspecting, for example, the MATLAB implementations of Krylov subspace solvers, we can see that the residual norm which is used for evaluating the stopping criterion, is always computed as `normr = norm(b - A * x);` instead of using the norm of the updated residual. This effort is often necessary because in finite precision these two theoretically equal values tend to differ significantly after sometimes only a few iterations (see Figure 7.6).

Figure 7.6: This figure compares the norm of the iteratively updated residual (thin lines) and the exact residual $\|\boldsymbol{b} - \boldsymbol{Ax}\|_2$ (thick lines) computed with differently precise arithmetics (`extended`: solid, `multiple<2>`: dashed).

However, there is one extra matrix-vector multiplication at each step and this information is solely used to decide whether the iteration should be stopped or not. Fortunately, we can do much better. Assume we have the internal precision adjusted to `doubledouble` (staggered with length 2). For generating the Krylov spaces, we need

$$\boldsymbol{Ar} = \boldsymbol{A}(\boldsymbol{r}^{(1)} + \boldsymbol{r}^{(2)}) = \boldsymbol{Ar}^{(1)} + \boldsymbol{Ar}^{(2)}.$$

That means, *increasing the precision by one* is comparable with doing one extra matrix-vector product. However, with this approach, we do not only improve the stopping criterion, but also significantly improve the accuracy of the iterated so-

lution, see Figure 7.6. This effect is demonstrated with the `mcca` matrix which is described in Matrix A.7.

7.5   Solving Ill-Conditioned Test-Matrices

Here we solved some GK4.16(n) (see (6.1) on page 89) systems with a preconditioned CG algorithm with Cholesky preconditioner (see Table 7.2).

Matrix		double	doubleX	extended	multiple<2>
$n = 100$	*error*	$1.3 \cdot 10^{-9}$	$1.6 \cdot 10^{-15}$	$2.0 \cdot 10^{-12}$	$1.4 \cdot 10^{-23}$
$\sigma_{\min} = 8.4 \cdot 10^{-7}$	*bound*	$7.5 \cdot 10^{-3}$	$2.0 \cdot 10^{-3}$	$4.2 \cdot 10^{-6}$	$4.9 \cdot 10^{-15}$
vtime < 0.01sec	*iter*	3^*	3^*	2	2
	time	< 0.01	< 0.01	< 0.01	0.01
$n = 1000$	*error*	$9.6 \cdot 10^{-5}$	$5.3 \cdot 10^{-15}$	$9.0 \cdot 10^{-8}$	$6.9 \cdot 10^{-25}$
$\sigma_{\min} = 9.7 \cdot 10^{-11}$	*bound*	> 1	> 1	> 1	$1.0 \cdot 10^{-11}$
vtime $= 0.04$sec	*iter*	3^*	3^*	3^*	3
	time	< 0.01	0.13^a	0.02	0.15
$n = 10\,000$	*error*	> 1	$3.2 \cdot 10^{-14}$	$2.7 \cdot 10^{-3}$	$4.8 \cdot 10^{-25}$
$\sigma_{\min} = 9.7 \cdot 10^{-15}$	*bound*	> 1	> 1	> 1	$2.5 \cdot 10^{-6}$
vtime $= 0.47$sec	*iter*	3^*	5^*	3^*	5
	time	0.08	2.33^a	0.10	3.06
$n = 50\,000$	*error*	> 1	$2.5 \cdot 10^{-13}$	> 1	$1.9 \cdot 10^{-14}$
$\sigma_{\min} = 2.2 \cdot 10^{-17}$	*bound*	> 1	> 1	> 1	$3.3 \cdot 10^{-7}$
vtime $= 3.01$sec	*iter*	3^*	12^*	13^*	9
	time	0.45	28.66^a	2.83	28.73
$n = 100\,000$	*error*	> 1	$2.8 \cdot 10^{-13}$	> 1	$4.7 \cdot 10^{-14}$
$\sigma_{\min} = 4.1 \cdot 10^{-18}$	*bound*	> 1	> 1	> 1	$7.0 \cdot 10^{-7}$
vtime $= 4.83$sec	*iter*	3^*	11^*	7^*	12
	time	0.93	52.55^a	2.96	77.31

Table 7.2: Solving some GK4.16(n) systems with an incomplete Cholesky preconditioned CG solver. The iteration was stopped after 5 correct digits were guaranteed (by the verification procedure) or after stagnation (gray). The cases where we have 5 digits accuracy, compared to the previously computed highly precise verified solution (but not verified), are displayed in dark gray.

[a]Note that this loss of convergence speed is caused by the slow software simulation of the exact scalar product. Sufficiently supported in hardware, `doubleX` should need the same time as `double`.

In this and all following examples, we stopped the iteration as soon as five correct digits of the solution could be guaranteed or after stagnation of the residual norm (marked by an *). With *error* we denote the actual relative error of the approximate solution. Usually, we have no exact solution available and therefore we cannot compute the *error*. However, for these examples we use a very tight enclosure of the exact solution, computed with a high precision arithmetic. With *bound* we denote the computed upper bound of the error and *iter* and *time* are the

number of iterations and the time needed for these iterations (in sec, measured on a
PentiumII/400). The quantity *vtime* denotes the time needed to compute a rigorous
bound for the smallest singular value of $\boldsymbol{A}$.

As we can see, verification with the techniques described in Section 4, is only
possible in conjunction with higher precision arithmetic. Even if we are only inter-
ested in a non-verified solution, we cannot trust in standard floating-point (`double`)
arithmetic. However, simply replacing all floating-point scalar products by exact
scalar products suffices to deliver always enough correct digits in the approximate
solution, although not verified. Using a 128 bit arithmetic we always achieved fast
convergence and highly accurate verified solutions.

In the next example we solve some Hilbert systems, again with a Cholesky pre-
conditioned `CG` solver, see Table 7.3. With a standard `double` or `extended` arith-
metic for the solver, we can only handle very small dimensions, while a mantissa
length of 128 bit always is sufficient to get fast convergence and good approxima-
tions to the solution. At dimension 14, the Cholesky decomposition (computed in
`double`) fails.

dim	$\sigma_{\min}$	double		extended		multiple<2>		Chol. prec.
		iter	time	iter	time	iter	time	
8	$3.60 \cdot 10^{-05}$	3	< 0.01	1	< 0.01	1	< 0.01	double
10	$2.29 \cdot 10^{-05}$	> 10	—	2	< 0.01	2	< 0.01	
12	$5.11 \cdot 10^{-07}$	> 12	—	> 12	—	3	< 0.01	
13	$3.05 \cdot 10^{-08}$	> 13	—	> 13	—	4	< 0.01	
15	$4.92 \cdot 10^{-09}$	> 15	—	> 15	—	1	0.01	multiple<2>
17	$2.83 \cdot 10^{-10}$	> 17	—	> 17	—	1	0.02	
19	$9.62 \cdot 10^{-12}$	> 19	—	> 19	—	1	0.02	
21	$3.60 \cdot 10^{-13}$	> 21	—	> 21	—	1	0.02	

Table 7.3: Here we aimed to verify 5 correct digits in the solution of various
Hilbert systems. For dimensions up to 13, the Cholesky precondi-
tioner was computed in `double` while the higher dimensional Hilbert
matrices were factorized with a `multiple<2>` arithmetic.

Increasing the precision used to compute the Cholesky decomposition enables us
to handle larger Hilbert matrices. In Table 7.3, rows 15-21 we used the data type
`multiple<2>` for computing the preconditioner. Now the Cholesky decomposition
is sufficient to solve the linear system in one step.

We stop at dimension 21, because it is not possible to store higher dimensional
Hilbert matrices in IEEE double exactly (compare Section 6.1.2).

This high precision preconditioning also allows us to handle larger dimensions for
the `GK4.16` matrices. Table 7.4 shows the results of the `GK4.16(2 000 000)` system.

Just to see how far we can go with Hilbert matrices, I extended **vk** to allow
multiple precision system matrices (in the standard version, system matrices are
always stored in `double`). With this extension we can even solve Hilbert matrices
of dimension 42 [1] and higher[1].

---

[1] In fact, **vk** solved the Hilbert 42 system in less than 8 seconds with 113 guaranteed decimals
(using a data type with 2560 bits mantissa length, i.e. approximately 770 decimal digits).

Matrix		doubleX	extended	multiple<2>	multiple<3>
$n = 2\,000\,000$	error	—	—	—	—
$\sigma_{\min} = 4.2 \cdot 10^{-24}$	bound	> 1	> 1	$1.8 \cdot 10^{-2}$	$6.7 \cdot 10^{-7}$
$vtime = 123.27\text{sec}$	iter	8^*	2^*	2^*	2
	time	903.34^a	18.15	261.42	306.54

Table 7.4: This table shows the same experiment as Table 7.2 but now with dimension $2\,000\,000$ (this was the largest possible dimension solvable on my PC due to storage limitations). See Table 7.2 for explanation of footnote[a].

Our final 'test' example is matrix GK4.20 (see (6.2) on page 89). This matrix is symmetric and indefinite but it nevertheless turns out that CG works fine. Here we used a modified $\boldsymbol{LDL}^T$ preconditioner, Table 7.5.

Matrix		double	doubleX	extended	multiple<2>
$n = 100000$	error	$2.7 \cdot 10^{-7}$	$1.5 \cdot 10^{-8}$	$1.3 \cdot 10^{-10}$	$3.5 \cdot 10^{-14}$
$\sigma_{\min} = 9.8 \cdot 10^{-11}$	bound	$5.6 \cdot 10^{-4}$	$3.1 \cdot 10^{-4}$	$2.9 \cdot 10^{-7}$	$1.7 \cdot 10^{-13}$
$vtime = 15.44\text{sec}$	iter	3^*	3^*	2	2
	time	2.43	13.87^a	0.74	10.61
$n = 500000$	error	$1.5 \cdot 10^{-6}$	$1.4 \cdot 10^{-7}$	$7.3 \cdot 10^{-10}$	$8.6 \cdot 10^{-14}$
$\sigma_{\min} = 2.9 \cdot 10^{-12}$	bound	$4.6 \cdot 10^{-2}$	$2.2 \cdot 10^{-2}$	$2.9 \cdot 10^{-5}$	$2.2 \cdot 10^{-9}$
$vtime = 78.11\text{sec}$	iter	3^*	3^*	3^*	2
	time	12.08	68.7^a	16.78	53.67
$n = 1000000$	error	$7.3 \cdot 10^{-6}$	$1.6 \cdot 10^{-6}$	$3.6 \cdot 10^{-9}$	$1.1 \cdot 10^{-13}$
$\sigma_{\min} = 8.1 \cdot 10^{-13}$	bound	0.39	> 1	$2.4 \cdot 10^{-4}$	$8.2 \cdot 10^{-9}$
$vtime = 157.03\text{sec}$	iter	3^*	3^*	3^*	2
	time	24.58	141.4^a	33.55	$1{:}48.3$

Table 7.5: This table shows the results of our experiments with the GK4.20 matrices. See Table 7.2 for explanation of the used notations. See Table 7.2 for explanation of footnote[a].

7.6 Verified Solutions for 'Real-Life' Problems

In this section we investigate some example systems taken from various application areas such as fluid dynamics, structural engineering, computer component design, and chemical engineering (see Appendix A).

Symmetric positive definite systems we always solved with a Cholesky preconditioned Conjugate Gradient solver. In the nonsymmetric case we show only the results of the fastest ILU preconditioned Krylov solver.

Particularly we utilized various solvers (BiCG, CGS, and BiCGStab) and incomplete preconditioners. In this context of high precision arithmetics, the GMRES algorithm couldn't compete with the *short recurrence solvers*. Since GMRES needs

most arithmetic operations and storage anyway, this lack is even reinforced by the increased requirements in memory and computing time for the high precision arithmetic operations.

In Table 7.6 we compare the results achieved by using several arithmetics (see Table 7.2 for explanation). Table 7.7 gives a quick overview over some systems we solved with **vk**.

Matrix		double	doubleX	extended	multiple<2>
fidap009	*error*	$1.9 \cdot 10^{-4}$	$1.1 \cdot 10^{-13}$	$1.6 \cdot 10^{-6}$	$6.2 \cdot 10^{-11}$
$n = 4683$	*bound*	0.33	$6.6 \cdot 10^{-2}$	$2.0 \cdot 10^{-4}$	$1.1 \cdot 10^{-8}$
$\sigma_{\min} = 2.9 \cdot 10^{-4}$	*iter*	3^*	4^*	3^*	2
$vtime = 22.55$sec	*time*	0.53	6.72^a	0.68	2.82
s3rmt3m1	*error*	$3.4 \cdot 10^{-5}$	$8.6 \cdot 10^{-14}$	$2.5 \cdot 10^{-8}$	$4.3 \cdot 10^{-11}$
$n = 5489$	*bound*	> 1	> 1	$9.7 \cdot 10^{-2}$	$5.2 \cdot 10^{-7}$
$\sigma_{\min} = 3.5 \cdot 10^{-7}$	*iter*	3^*	3^*	3^*	2
$vtime = 507.6$sec	*time*	1.14	18.40^a	1.37	12.42
e30r5000	*error*	$1.1 \cdot 10^{-11}$	$1.5 \cdot 10^{-14}$	$5.4 \cdot 10^{-12}$	$1.2 \cdot 10^{-38}$
$n = 9661$	*bound*	> 1	> 1	0.7	$1.4 \cdot 10^{-32}$
$\sigma_{\min} = 3.7 \cdot 10^{-12}$	*iter*	3^*	3^*	3^*	2
$vtime = 2$h09.47min	*time*	3.86	69.4^a	5.45	70.41
e40r5000	*error*	$2.8 \cdot 10^{-11}$	$1.9 \cdot 10^{-14}$	$2.3 \cdot 10^{-14}$	$1.6 \cdot 10^{-32}$
$n = 17281$	*bound*	> 1	> 1	> 1	$1.3 \cdot 10^{-29}$
$\sigma_{\min} = 1.5 \cdot 10^{-13}$	*iter*	3^*	3^*	3^*	2
$vtime = 6$h48.09min	*time*	9.04	232.3^a	13.00	848.0

Table 7.6: Solving some 'real-life' problems with different arithmetics. See Table 7.2 for explanation of footnote[a] and the used notations.

Again, we often have enough correct digits in the approximate solution but usually we are not aware of this fact. Particularly, we are only able to prove this by using a higher precision arithmetic.

7.7   Verification via Normal Equations

In the nonsymmetric case, we have to meet the assumption

$$\sigma_{\min}(\boldsymbol{LU}) > \|\boldsymbol{LU} - \boldsymbol{A}\|_2 \tag{7.1}$$

which is sometimes a problem if either $\sigma_{\min}(\boldsymbol{LU})$ is very small or $\boldsymbol{A}$ is ill-conditioned and its elements are large. In such cases it is often advantageous to switch to the normal equations. We stress that we actually do not have to compute $\boldsymbol{A}^T\boldsymbol{A}$ and $\boldsymbol{A}^T\boldsymbol{b}$ [98]. Although we have squared the smallest singular value (which possibly makes it more difficult to find a verified lower bound if $\sigma_{\min}(\boldsymbol{LU}) < 1$), we now do not have to fulfill (7.1) anymore.

Using this technique, we solved, e.g., the `mcca` system (see Matrix A.7). For this matrix, **vk** computes a lower bound for the smallest singular value as $2.4 \cdot 10^{-2}$ (this

Name	dim	nnz	cond	bound	time
fs 680 1	680	2646	$2.1 \cdot 10^4$	$3.56 \cdot 10^{-38}$	21.75 sec
west2021	2021	7353	$7.5 \cdot 10^{12}$	$9.43 \cdot 10^{-25}$	705.93 sec
mvmtls4000	4000	8784	$2.7 \cdot 10^7$	$3.62 \cdot 10^{-30}$	1h03.55 min
pores2	1224	9613	$3.31 \cdot 10^8$	$4.87 \cdot 10^{-17}$	292.68 sec
bcsstk08	1074	12960	$4.7 \cdot 10^7$	$9.49 \cdot 10^{-28}$	82.72 sec
pde2961	2961	14585	$9.49 \cdot 10^2$	$1.77 \cdot 10^{-14}$	7.22 sec
add32	4960	23884	$2.14 \cdot 10^2$	$6.89 \cdot 10^{-15}$	1h08.28 min
fidap009	4683	95053	$1.04 \cdot 10^7$	$1.1 \cdot 10^{-8}$	25.37 sec
s3rmt3m1	5489	112505	$1.33 \cdot 10^{10}$	$5.2 \cdot 10^{-7}$	520.02 sec
e30r5000	9661	306356	$1.27 \cdot 10^{11}$	$1.2 \cdot 10^{-38}$	2h10.57 min
e40r5000	17281	553956	$1.4 \cdot 10^{16}$	$1.3 \cdot 10^{-29}$	7h02.09 min

Table 7.7: A quick overview over some systems we solved with **vk**. The objective was to get 5 correct digits in the iterated solution. Since convergence sometimes was very fast, we overshot at times.We only display the results of the smalles arithmetic that delivers these 5 digits (almost always `multiple<2>`).Note that 'time' denotes the overall time for solving and verification.

seems to be roughly underestimated due to the very high condition number) and an upper bound for $\|LU - A\|_2$ as $2.31 \cdot 10^9$. That is we cannot apply Theorem 4.4 directly. Switching to the normal equations, **vk** finds $7.67 \cdot 10^3$ as a lower bound for $\sigma_{\min}(A^T A)$. Applying a Cholesky preconditioned CG algorithm, we are able to verify five decimal digits in less than three seconds.

7.8 Performance Tuning

The verification time depends strongly on the number of nonzero elements and the bandwidth of L and U. Therefore, it is advantageous to reduce these quantities. We discuss two possibilities to achieve this reduction: column/row reordering algorithms and incomplete factorizations (see Section 1.4.2).
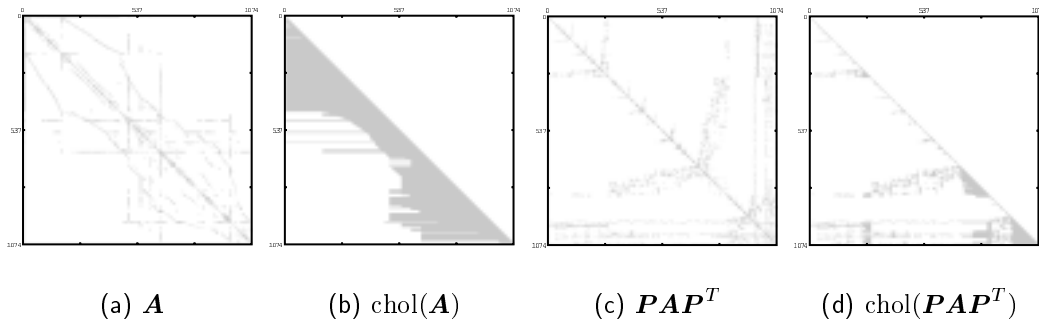
Drop-Tol.	nnz	$\sigma_{\min}(LU)$	$\|LU - A\|_2$	total time	bound
complete	54058 (6.67%)	$4.2284 \cdot 10^{-2}$	$4.50 \cdot 10^{-15}$	9.95 sec	$1.23 \cdot 10^{-26}$
$1 \cdot 10^{-4}$	32083 (3.96%)	$4.2456 \cdot 10^{-2}$	$2.43 \cdot 10^{-3}$	7.05 sec	$1.60 \cdot 10^{-8}$
$1 \cdot 10^{-3}$	26843 (3.31%)	$4.4330 \cdot 10^{-2}$	$7.68 \cdot 10^{-2}$	failed	—

Table 7.8: Here we demonstrate the possible speed up by using incomplete LU factorizations at the example of Matrix `pde900` (A.8).

The problem with incomplete factorizations is that we have to bear inequality (7.1) in mind. Particularly for ill-conditioned systems $\|LU - A\|_2$ grows heavily with increasing sparsity in L and U. However, if the smallest singular value of A is not too small, we can achieve a significant improvement, as shown in Table 7.8 at

the example of Matrix `pde900` (A.8). Switching from a complete LU preconditioner to incomplete LU with drop-tolerance 10^{-4} saves nearly 30% computing time

Applying reordering algorithms can sometimes dramatically speed up the verification process. We illustrate the effect of reordering with the `bcsstk08` and `tols4000` matrix (see Matrices A.2 and A.12). Because `bcsstk08` is symmetric, we applied the symmetric minimum degree reordering algorithm. This algorithm computes a permutation matrix $\boldsymbol{P}$ to a given matrix $\boldsymbol{A}$, such that the Cholesky factors of $\boldsymbol{PAP}^T$ have less nonzero elements, see Figure 7.7.

| (a) $\boldsymbol{A}$ | (b) chol($\boldsymbol{A}$) | (c) $\boldsymbol{PAP}^T$ | (d) chol($\boldsymbol{PAP}^T$) |

Figure 7.7: spy

Matrix `tols4000` is nonsymmetric and therefore we applied a reverse Cuthill-McKee algorithm, especially designed to deliver smaller bandwidths in the LU factors.

Table 7.9 shows some experiments with and without reordering.

Reordering Algorithm	Precond. (droptol)	Preconditioner		total time	bound
		nnz	lo/up bandw.		
none	Chol(10^{-5})	115645	591/1	158.62 sec	$9.01{\cdot}10^{-7}$
SymmMinDeg	Chol(10^{-5})	23688	1054/1	59.35 sec	$5.35{\cdot}10^{-8}$
none	ILU(10^{-6})	13584	2401/3218	63.55 min	$3.62{\cdot}10^{-30}$
Cuthill-McKee	ILU(10^{-6})	14820	89/90	34.84 sec	$6.10{\cdot}10^{-28}$

Table 7.9: Reordering algorithms can significantly speed up convergence. This table shows a symmetric and a nonsymmetric example. The first aims to reduce the number of nonzero elements while the second tries to reduce the bandwidth.

Conclusion

As the main result of our investigations, we conclude that traditionally used arithmetics (mostly IEEE double precision) are often *not the best choice* for solving linear systems of equations.

Both, theoretically and by examples, we showed that iterative solvers — particularly Krylov subspace methods — heavily suffer from rounding errors. Usually, computationally expensive reorthogonalization strategies (or even full orthogonalizing methods) are utilized to work against arithmetic insufficiencies.

In this thesis, we showed that using *improved* arithmetics can lead to much better results, compared to those obtained from *ordinary* floating-point arithmetic. Particularly, exchanging the classically used floating-point scalar product by the exact scalar product often suffices to obtain significantly more accuracy in the computed solutions, at least for not too ill-conditioned matrices. For symmetric systems we mostly obtained nearly maximum accuracy (13 to 15 correct decimal digits), although not verified.

Sufficiently supported in hardware, the exact scalar product can be computed as fast as an ordinary scalar product. Thus, we urgently postulate this technique to be implemented in future processors in hardware.

However, if we need to guarantee the computed solutions, this arithmetical improvement does often not benefit to obtain small error bounds. For this purpose,

---

[2] "A work of this kind actually never finishes. You have to declare it finished when you did all in your power, dependent on time and circumstances."

or when the condition number is too large, we have to switch to higher precision numbers. This enables us to solve practically arbitrary ill-conditioned systems with almost any accuracy and guaranteed error bounds (if needed).

Considering the results of this work, it should be discussed whether hardware manufacturers should be asked to develop hardware support for multiple precision arithmetics, maybe solely based on integer arithmetic. The number type might consist of a few bytes for the sign, exponent, and some status information, and a, propably variable, number of bytes for the mantissa. Alternatively, we could utilize staggered precision numbers, which would greatly suffice from a hardware supported exact scalar product. With this approach, we only had to extend modern computer architectures by one operation to get a high performance multiple precision arithmetic.

Combining these techniques, it should easily be possible, not only to save iterations (as we always did in our tests) but also to save real computing time, while simultaneously getting more accurate solutions. Additionally, the verification process will speed up significantly due to its extensive usage of exact scalar products.

Used Matrices

Here we list several test matrices used throughout this thesis. The matrices are taken from the Matrix-Market [85] and are alphabetically ordered.

Matrix A.1:   `add32`

Computer component design, 32-bit adder

S. Hamm, Motorola Inc. Semicond. Systems Design Technology

Size	Type	Properties
$dim = 4960$	real,	$\|\cdot\|_F = 1.6$
$nnz = 23884$	unsymmetric	$cond = 2.14 \cdot 10^2$
$bandw = 4030/4030$		$\sigma_{\min} = 2.99 \cdot 10^{-4}$

Matrix A.2:   `bcsstk08`

Structural engineering

John Lewis, Boeing Computer Services

Size	Type	Properties
$dim = 1074$	real,	$\|\cdot\|_F = 1.0 \cdot 10^{11}$
$nnz = 7017$	symmetric	$cond = 4.7 \cdot 10^7$
$bandw = 591/591$		$\sigma_{\min} = 2.1 \cdot 10^3$

Matrix A.3:   `e30r5000`

Driven cavity , 30x30 elements, Re=5000

Andrew Chapman, University of Minnesota

Size	Type	Properties
$dim = 9661$	real,	$\|\cdot\|_F = 2.20 \cdot 10^3$
$nnz = 306356$	unsymmetric	$cond = 1.27 \cdot 10^{11}$
$bandw = 342/342$		$\sigma_{\min} = 3.7 \cdot 10^{-12}$

Matrix A.4:   `e40r5000`

Driven cavity , 40x40 elements, Re=5000

Andrew Chapman, University of Minnesota

Size	Type	Properties
$dim = 17281$	real,	$\|\cdot\|_F = 2.10 \cdot 10^3$
$nnz = 553956$	unsymmetric	$cond = 7.68 \cdot 10^{10}$
$bandw = 452/452$		$\sigma_{\min} = 1.5 \cdot 10^{-13}$

Matrix A.5:   `fidap009`

Finite element modeling of fluid dynamics

Isaac Hasbani, Fluid Dynamics International

Size	Type	Properties
$dim = 3363$	real,	$\|\cdot\|_F = 3.00 \cdot 10^{10}$
$nnz = 99397$	symmetric	$cond = 4.05 \cdot 10^{13}$
$bandw = 86/86$		$\sigma_{\min} = 2.27 \cdot 10^{-4}$

Matrix A.6:   `fs680 1`

Chemical kinetics problems

Alan Curtis, Computer Science and Systems Division

Size	Type	Properties
$dim = 680$	real,	$\|\cdot\|_F = 1.2 \cdot 10^{14}$
$nnz = 2646$	unsymmetric	$cond = 2.1 \cdot 10^4$
$bandw = 561/281$		$\sigma_{\min} = 7.44 \cdot 10^8$

Matrix A.7:   `mcca`

Nonlinear radiative transfer and statistical equilibrum in astro-physics

Mats Carlson, Institute of Theoretical Astrophysics

Size	Type	Properties
$dim = 180$	real,	$\|\cdot\|_F = 2.3 \cdot 10^{19}$
$nnz = 2659$	unsymmetric	$cond = 3.6 \cdot 10^{17}$
$bandw = 43/66$		$\sigma_{\min} =$

Matrix A.8: `pde900`

Elliptic partial differential equation

H. Elman, University of Maryland

Size	Type	Properties
$dim = 900$	real,	$\lVert \cdot \rVert_F = 2.2 \cdot 10^2$
$nnz = 4380$	unsymmetric	$cond = 8.73 \cdot 10^2$
$bandw = 31/31$		$\sigma_{\min} = 4.43 \cdot 10^{-2}$

Matrix A.9: `pde2961`

Elliptic partial differential equation

H. Elman, University of Maryland

Size	Type	Properties
$dim = 2961$	real,	$\lVert \cdot \rVert_F = 2.2 \cdot 10^2$
$nnz = 14585$	unsymmetric	$cond = 9.49 \cdot 10^2$
$bandw = 48/48$		$\sigma_{\min} = 4.23 \cdot 10^{-2}$

Matrix A.10: `pores2`

Reservoir modeling

John Appleyard, Harwell Laboratory

Size	Type	Properties
$dim = 1224$	real,	$\lVert \cdot \rVert_F = 1.5 \cdot 10^8$
$nnz = 9613$	unsymmetric	$cond = 3.31 \cdot 10^8$
$bandw = 472/471$		$\sigma_{\min} = 2.63 \cdot 10^{-2}$

Matrix A.11: `s3rmt3m1`

Finite element analysis of cylindrical shells

Reijo Kouhia, Helsinki University of Technology

Size	Type	Properties
$dim = 5489$	real,	$\lVert \cdot \rVert_F = 1.7 \cdot 10^5$
$nnz = 112505$	symmetric	$cond = 1.33 \cdot 10^{10}$
$bandw = 192/192$		$\sigma_{\min} = 3.50 \cdot 10^{-7}$

Matrix A.12: `tols4000`

Aeroelasticity, stability analysis of an airplane in flight

S. Godet-Thobie, CERFACS and C. Bès, Aerospatiale

Size	Type	Properties
$dim = 4000$	real,	$\lVert \cdot \rVert_F = 3 \cdot 10^8$
$nnz = 8784$	unsymmetric	$cond = 2.7 \cdot 10^7$
$bandw = 2401/2418$		$\sigma_{\min} = 3.03 \cdot 10^{-12}$

Matrix A.13: `west2021`

Chemical engineering plant models

Art Westerberg, University of Pittsburgh

Size	Type	Properties
$dim = 2021$	real,	$\| \cdot \|_F = 1.8 \cdot 10^6$
$nnz = 7353$	unsymmetric	$cond = 7.50 \cdot 10^{12}$
$bandw = 1888/1309$		$\sigma_{\min} = 2.90 \cdot 10^{-8}$

Free and Open Source Software

" *When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.* "

GNU General Public License (Version 2), 1991

At this place, I wish to thank the hundreds of programmers that spend their time, energy and knowledge in producing free and open source software. This thesis would basically be impossible in this form without these programs. In the following, I enumerate the most important of them, used for writing this thesis and coding the programs.

First of all, I want to mention the operating system itself: LINUX. All text editing was done with XEmacs combined with auctex-mode. For formatting the thesis I used TeX/LaTeX together with a couple of packages and my own document style 'dissbook'. The graphics where created using XFig, The Gimp and gnuplot. Besides the dozens of really helpful little (and large) utilities, my programming environment consisted of the GNU C compiler, egcs, xxgdb and again XEmacs with cc-mode. Additionally I used several libraries, like gmp (GNU multiple precision), doubledouble, profil, BIAS and gtk (GNU toolkit).

Curriculum Vitae

Name		Axel Facius
Address		Jenaer Strasse 8, 76139 Karlsruhe
Date of Birth/ Birthplace		June 21, 1969 in Schwäbisch Gmünd
Nationality		German
School Education	1976 - 1980	Grundschule (primary school)
	1980 - 1989	Gymnasium (high school)
University	1991 - 1997	study of Mathematics, Computer Science and Electrical Engineering at the University of Karlsruhe
	1995	project on "Simulation of Dynamical Oscillators in Neural Networks on Parallel Computers"
	1996-1997	Diploma-Thesis at the Institute of Logic, Complexity, and Deduction Systems, Prof. Menzel, subject: "Reconstruction and Analysis of Independent Components with Neuronal Networks"
Diploma	11/1997	graduation to 'Diplom Technomathematiker'
Thesis	since 1998	work on Ph.D. at the Institute of Applied Mathematics, advisors Prof. Kulisch and Dr. Lohner

Bibliography

[1] D. Adams. *Life, the Universe and Everything.* Ballantine Books, 1995.

[2] L. Adams and H. Jordan. Is SOR color-blind? *SIAM J. Sci. Statist. Comput.*, 7:490–506, 1986.

[3] G. Alefeld and J. Herzberger. *Introduction to Interval Computations.* Academic Press, 1983.

[4] American National Standards Institute / Institute of Electrical and Electronic Engineers, New York. *A Standard for Binary Floating-Point Arithmetic*, 1985. ANSI/IEEE Std. 754-1985.

[5] M. Arioli and C. Fassino. Roundoff error analysis of algorithms based on Krylov subspace methods. *BIT*, 36:189–205, 1996.

[6] W. Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quart. Appl. Math.*, 9:17–29, 1951.

[7] O. Axelson. Bounds of eigenvalues of preconditioned matrices. *SIAM J. Matrix Anal. Appl.*, 13:847–862, 1992.

[8] O. Axelson and G. Lindskog. On the eigenvalue distribution of a class of precondition matrices. *Numer. Math.*, 48:479–498, 1986.

[9] R. Barret, M. Berry, T. 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, 1993.

[10] C. Baumhof. *Ein Vektorarithmetik-Koprozessor in VLSI-Technik zur Unterstützung des Wissenschaftlichen Rechnens.* PhD thesis, Universität Karlsruhe, Karlsruhe, Germany, 1996.

[11] R. Beauwens. Modified incomplete factorization strategies. In O. Axelson and L. Kolotilina, editors, *Preconditioned Conjugate Gradient Methods*, Lecture Notes in Mathematics 1457, pages 1–16. Springer Verlag, Berlin, New York, 1990.

[12] J. A. Bollen. Round-off error analysis of descent methods for solving linear equations. Master's thesis, Technische Hogeschool Eindhofen, 1980.

[13] R. P. Brent. A Fortran multiple-precision arithmetic package. *ACM Trans. Math. Softw.*, 4:57–70, 1978.

[14] C. Brezinsky, M. Zagila, and H. Sadok. Avoiding breakdown in the CGS algorithm. *Numer. Alg.*, 1:261–284, 1991.

[15] C. Brezinsky, M. Zagila, and H. Sadok. A breakdown free Lanczos type algorithm for solving linear systems. *Numer. Math.*, 63:29–38, 1992.

[16] K. Briggs. The doubledouble homepage.
`www-epidem.plantsci.cam.ac.uk/~kbriggs/doubledouble.html`

[17] J. R. Bunch and D. J. Rose. *Sparse Matrix Computations*. Academic Press Inc., New York, San Francisco, London, 1976.

[18] T. Chan, E. Gallopoulos, V. Smoncini, T. Szeto, and C. Tong. A quasi minimal residual variant of the Bi-CGSTAB algorithm for non-symmetric systems. *SIAM J. Sci. Comp.*, 15:338–347, 1994.

[19] L. Collatz. *Differentialgleichungen*. 6. Aufl. Teubner, Stuttgart, 1981.

[20] J. K. Cullum and A. Greenbaum. Relations between Galerkin and norm-minimizing iterative methods for solving linear systems. *SIAM J. Matrix Anal. Appl.*, 17:223–247, 1996.

[21] J. K. Cullum and R. A. Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*, volume I Theory. Birkhäuser, Boston, Basel, Stuttgart, 1985.

[22] J. K. Cullum and R. A. Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*, volume II Programs. Birkhäuser, Boston, Basel, Stuttgart, 1985.

[23] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *ACM Proc. 24th Nat. Conf.*, 1969.

[24] G. Dahlquist, S. C. Eisenstat, and G. H. Golub. Bounds for the error of linear systems of equations using the theory of moments. *J. Math. Anal. Appl.*, 37:151–166, 1972.

[25] T. J. Dekker. A floating-point technique for extending the available precision. *Numer. Math.*, 18:224–242, 1971.

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

[27] J. Demmel, B. Diament, and G. Malajovich. On the complexity of computing error bounds. `www.cs.berkeley.edu/~demmel/`, 1999.

[28] J. Dongarra, I. Duff, D. Sorensen, and H. van der Vorst. *Numerical Linear Algebra on High-Performance Computers*. SIAM, Philadelphia, PA, 1998.

[29] J. Dongarra and R. C. Whaley. Automatically tuned linear algebra software (ATLAS). Technical report, University of Tennessee and Oak Ridge National Laboratory, 1997.

[30] J. J. Du Croz and N. J. Higham. Stability of methods for matrix inversion. *IMA J. Numer. Anal.*, 12:1–19, 1992.

[31] I. Duff and G. Meurant. The effect of ordering on preconditioned conjugate gradients. *BIT*, 29:635–657, 1989.

[32] V. Faber, W. Joubert, M. Knill, and T. Manteuffel. Minimal residual method stronger than polynomial preconditioning. *SIAM J. Matrix Anal. Appl.*, 17:707–729, 1996.

[33] V. Faber and T. Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM J. Numer. Anal. Appl.*, 21:315–339, 1984.

[34] A. Facius. Influences of rounding errors in solving large sparse linear systems. In T. Csendes, editor, *Developments in Reliable Computing*, pages 17–30. Kluwer Academics, 1999.

[35] R. Fletcher. Conjugate gradient methods for indefinite systems. In G. Watson, editor, *Numerical Analysis Dundee 1975*. Springer, Berlin, New York, 1976.

[36] G. E. Forsythe, M. A. Malcolm, and C. B. Moler. *Computer Methods for Mathematical Computations*. Prentice–Hall, Englewood Cliffs, 1977.

[37] R. W. Freund. A transpose-free quasi-minimum residual algorithm for non-Hermitian linear systems. *SIAM J. Sci. Comput.*, 14:470–482, 1993.

[38] A. Frommer and A. Weinberg. Verified error bounds for linear systems through the Lanczos process. *Reliable Computing*, 5(3):255–267, 1999.

[39] K. Gallivan, M. Heath, E. Ng, J. Orthega, B. Peyton, R. Plemmons, C. Romine, A. Sameh, and R. Voigt. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, PA, 1990.

[40] gnuplot. `www.gnuplot.org`, 1999. Version 3.7.1.

[41] G. Golub and C. van Loan. *Matrix Computations*. Johns Hopkins, third edition, 1996.

[42] G. H. Golub and G. Meurant. Matrices, moments and quadrature. *BIT*, 1994.

[43] G. H. Golub and G. Meurant. Matrices, moments and quadratures II or how to compute the norm of the error in iterative methods. *BIT*, 1997.

[44] G. H. Golub and D. O'Leary. Some history of the conjugate gradient and Lanczos methods. *SIAM Rev.*, 31:50–102, 1989.

[45] G. H. Golub and Z. Strakoš. Estimates in quadratic formulas. *Numerical Algorithms*, 8:241–268, 1994.

[46] T. Granlund. The GNU Multiple Precision Arithmetic Library.
     `www.csd.uu.se/documentation/~programming/gmp/`

[47] A. Greenbaum. Estimating the attainable accuracy of recursively computed
     residual methods. *SIAM J. Matrix Anal. Appl.*, 18:535–551, 1997.

[48] A. Greenbaum. *Iterative Methods for Solving Linear Systems*. SIAM, Philadel-
     phia, 1997.

[49] A. Greenbaum, V. Pták, and Z. Strakoš. Any nonincreasing convergence curve
     is possible for GMRES. *SIAM J. Matrix Anal. Appl.*, 17:465–469, 1996.

[50] A. Greenbaum, M. Rozložník, and Z. Strakoš. Numerical behavior of the
     modified Gram-Schmidt GMRES implementation. *BIT*, 37:706–719, 1997.

[51] A. Greenbaum and Z. Strakoš. Predicting the behavior of finite precision
     Lanczos and conjugate gradient computations. *SIAM J. Matrix Anal. Appl.*,
     13:121–137, 1992.

[52] R. Gregory and D. Karney. *A Collection of Matrices for Testing Computational
     Algorithms*. Wiley Interscience, 1969.

[53] GTK — A X-Window GUI toolkit. `www.gtk.org`, 1999. Version 3.7.1.

[54] J. Gustafson. Computational verifyability and feasibility of the ASCI pro-
     gramm. *IEEE Comput. Sci. Eng.*, 5:36–45, 1998.

[55] W. Hackbusch. *Iterative Lösung großer schwachbesetzter Gleichungssysteme*.
     B. G. Teubner, Stuttgart, 1993.

[56] W. Hager. Condition estimators. *SIAM J. Sci. Statist. Comput.*, 5:311–316,
     1984.

[57] R. Hammer, M. Hocks, U. Kulisch, and D. Ratz. *Toolbox for Verified Com-
     puting*. Springer, Berlin Heidelberg, 1993.

[58] R. Hammer, M. Hocks, U. Kulisch, and D. Ratz. *C++ Toolbox for Verified
     Computing*. Springer, Berlin Heidelberg, 1995.

[59] M. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear
     systems. *Journal of Research of the National Bureau of Standards*, 49:409–436,
     1952.

[60] M. R. Hestenes. Conjugacy and gradients. In *A History of Scientific Comput-
     ing*, pages 167–179. Addison-Wesley, Reading, MA, 1990.

[61] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM,
     Philadelphia, 1996.

[62] N. J. Higham and P. A. Knight. Finite precision behavior of stationary itera-
     tion for solving linear singular systems. *Lin. Alg. Appl.*, 192:165–186, 1993.

[63] M. Hochbruck. *Lanczos- und Krylov-Verfahren für nicht-hermitesche lineare Systeme*. PhD thesis, Universität Karlsruhe, Fakultät für Mathematik, 1992.

[64] M. Hochbruck and C. Lubich. Error analysis of Krylov methods in a nutshell. *SIAM J. Sci. Comput.*, 19:695–701, 1998.

[65] W. Kahan and B. N. Parlett. How far should you go with the Lanczos process. In *Sparse Matrix Computations*, 1974.

[66] S. Kaniel. Estimates for some computational techniques in linear algebra. *Math. Comput.*, 20:369–378, 1966.

[67] R. Kirchner and U. Kulisch. Accurate arithmetic for vector processors. *J. Parallel Distrib. Comput.*, 5:250–270, 1988.

[68] R. Klatte, U. Kulisch, M. Neaga, D. Ratz, and C. Ullrich. *PASCAL-XSC*. Springer Verlag, 1991.

[69] R. Klatte, U. Kulisch, A. Wiethoff, C. Lawo, and M. Rauch. *C-XSC*. Springer Verlag, 1992.

[70] O. Knüppel. PROFIL (Programmer's Runtime Optimized Fast Interval Library) and BIAS (Basic Interval Arithmetic Subroutines).
`www.ti3.tu-harburg.de/Software/PROFIL.html`

[71] O. Knüppel. BIAS — basic interval arithmetic subroutines. Technical report, Universität Hamburg-Harburg, Hamburg, Germany, 1993.

[72] O. Knüppel. PROFIL — Programmer's Runtime Optimized Fast Interval Library. Technical report, Universität Hamburg-Harburg, Hamburg, Germany, 1993.

[73] U. Kulisch, editor. *Wissenschaftliches Rechnen mit Ergebnisverifikation – Eine Einführung*. Akademie Verlag, Ost-Berlin, Vieweg, Wiesbaden, 1989.

[74] U. Kulisch. Advanced arithmetic for digital computer design of arithmetic units. *Electronic Notes in Theoretical Computer Science*, 24, 1999.

[75] U. Kulisch and W. L. Miranker. *Computer Arithmetic in Theory and Practice*. Academic Press, 1981.

[76] U. Kulisch and W. L. Miranker, editors. *A New Approach to Scientific Computation*. Academic Press, New York, 1983.

[77] C. Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards*, 45:255–282, 1950.

[78] M. Lerch. Expression templates for dot product expressions. personal communication, 1998.

[79] M. Lerch. Expression concepts in scientific computing. In T. Csendes, editor, *Developments in Reliable Computing*, pages 119–130. Kluwer Academics, 1999.

[80] S. Linnainmaa. Software for doubled precision floating-point computations. *ACM Trans. Math. Software*, 7:272–283, 1981.

[81] R. Lohner. A verified solver for linear systems with band structure. to be included in: Toolbox for Verified Computing II.

[82] R. Lohner. Interval arithmetic in staggered correction format. *Scientific Computing with Automatic Result Verification*, 8:301–321, 1993.

[83] G. Luenberger. *Introduction to Linear and Nonlinear Programming*. Addison Wesley, Reading, New York, 1973.

[84] M. Mascagni and W. L. Miranker. Arithmetically improved algorithmic performance. *Computing*, 35:153–175, 1985.

[85] The matrix market: a visual web database for numerical matrix data. `math.nist.gov/MatrixMarket/`

[86] G. Mayer. Enclosing the solutions of systems of linear equations by interval iterative processes. In U. Kulisch and H. Stetter, editors, *Scientific Computation with Automatic Result Verification*, number 6 in Computing Supplementum, pages 47–58. Springer Verlag, 1988.

[87] G. Mayer and J. Rohn. On the applicability of the interval Gaussian algorithm. *Reliable Computing*, 4:205–222, 1998.

[88] D. R. McCoy and E. W. Larsen. Unconditionally stable diffusion-synthetic acceleration methods for the slab geometry discrete ordinates equations. *Nuclear Sci. Engrg.*, 82:47–70, 1982. Parts I and II.

[89] A. Meister. Numerische Lineare Algebra. Skriptum, Institut für Angewandte Mathematik, Universität Hamburg, WS 1997/98.

[90] S. G. Mikhlin. *Error Analysis in Numerical Processes*. John Wiley & Sons, Chichester, New York, Brisbane, Toronto, Singapore, 1991.

[91] N. Nachtigal. *A look-ahead variant of the Lanczos algorithm and its application to the quasi minimal residual method for non-Hermitian linear systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1991.

[92] A. Neumaier. The wrapping effect, ellipsoid arithmetic, stability and confidence regions. In R. Albrecht, G. Alefeld, and H. J. Stetter, editors, *Validation Numerics*, Comput. Suppl., pages 175–190. Springer, 1993.

[93] C. Paige. *The Computation of Eigenvalues and Eigenvectors of Very Large Sparse Matrices*. PhD thesis, University of London, 1971.

[94] C. Paige. Error analysis of the Lanczos algorithm for tridiagonalizing a symmetric matrix. *J. Inst. Maths. Appl.*, 18:341–349, 1976.

[95] C. Paige. Accuracy and effectiveness of the Lanczos algorithm for the symmetric eigenproblem. *Lin. Alg. Appl.*, 34:235–258, 1980.

[96] C. Paige, B. Parlett, and H. van der Vorst. Approximate solutions and eigen-value bounds from Krylov subspaces. *Numer. Lin. Alg. Appl.*, 29:115–134, 1995.

[97] C. Paige and M. Saunders. Solutions of sparse indefinite systems of linear equations. *SIAM J. Numer. Anal.*, 12:617–629, 1975.

[98] C. Paige and M. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Trans. Math. Soft.*, 8:43–71, 1982.

[99] B. N. Parlett. A new look at the Lanczos algorithm for solving symmetric systems of linear equations. *Lin. Alg. Appl.*, 29:323–346, 1980.

[100] B. N. Parlett. *The Symmetric Eigenvalue Problem*. Prentice-Hall, Englewood Cliffs, NJ., 1980.

[101] B. N. Parlett and D. S. Scott. The Lanczos algorithm with selective orthogo-nalization. *Math. Comput.*, 33:217–238, 1979.

[102] B. N. Parlett, D. R. Taylor, and Z. A. Liu. A look-ahead Lanczos algorithm for unsymmetric matrices. *Math. Comput.*, 44:105–124, 1985.

[103] D. A. Pope and M. L. Stein. Multiple-precision arithmetic. *Comm. ACM*, 13:809–813, 1970.

[104] J. K. Reid. A note on the stability of Gaussian elimination. *J. Inst. Math. Applic.*, 8:374–375, 1971.

[105] M. Rozložník. *Numerical Stability of the GMRES Method*. PhD thesis, Akademie věd České Republiky, 1997.

[106] S. M. Rump. *Kleine Fehlerschranken bei Matrixproblemen*. PhD thesis, Uni-versität Karlsruhe, Karlsruhe, Germany, 1980.

[107] S. M. Rump. Solving non-linear systems with least significant bit accuracy. *Computing*, 29:183–200, 1982.

[108] S. M. Rump. Solving algebraic problems with high accuracy. In U. Kulisch and W. Miranker, editors, *A New Approach to Scientific Computation*, pages 53–120. Academic Press, 1983.

[109] S. M. Rump. Validated solution of large linear systems. In R. Albrecht, G. Ale-feld, and H. J. Stetter, editors, *Validation Numerics*, number 9 in Computing Supplementum, pages 191–212. Springer Verlag, 1993.

[110] S. M. Rump. Verification methods for dense and sparse systems of equations. Technical report, Universität Hamburg-Harburg, 1993.

[111] S. M. Rump. INTLAB — interval laboratory. Technical report, Universität Hamburg-Harburg, 1998.

[112] Y. Saad. The Lanczos biorthogonalization algorithm and other oblique projection methods for solving large unsymmetric systems. *SIAM J. Numer. Anal.*, 19:485–506, 1982.

[113] Y. Saad. SPARSEKIT: A basic tool kit for sparse matrix computations. Technical report, NASA Ames Research Center, 1990.

[114] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, Boston, MA, 1996.

[115] Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7:856–869, 1986.

[116] J. Siek. The basic linear algebra instruction set: Building blocks for portable high performance. In *SciTools*, 1998.

[117] J. G. Siek. Generic programming for high performance numerical linear algebra. In *SIAM Workshop on Inter-operable Object-Oriented Scientific Computing*, 1998.

[118] J. G. Siek. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.

[119] H. D. Simon. *The Lanczos Algorithm for Solving Symmetric Linear Systems*. PhD thesis, University of California, Berkeley, 1982.

[120] H. D. Simon. Analysis of the Lanczos algorithm with reorthogonalization methods. *Lin. Alg. Appl.*, 61:101–131, 1984.

[121] H. D. Simon. The Lanczos algorithm with partial reorthogonalization. *Math. Comput.*, 42:115–142, 1984.

[122] G. L. G. Sleijpen, H. A. van der Vorst, and J. Modersitzki. *The Main Effects of Rounding Errors in Krylov Solvers for Symmetric Linear Systems*. Universiteit Utrecht, The Netherlands, 1997. Preprint 1006.

[123] D. M. Smith. A Fortran package for floating-point multiple-precision arithmetic. *ACM Trans. Math. Softw.*, 17:273–283, 1991.

[124] P. Sonnefeld. CGS, a fast Lanczos-type solver for unsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 10:36–52, 1989.

[125] BLAS standard draft chapter 3: Sparse BLAS, 1997. Technical report Basic Linear Algebra Subprograms Technical Forum.

[126] J. Stoer. *Numerische Mathematik*. Springer Verlag, New York, Heidelberg, Berlin, 1993.

[127] J. Stoer and R. Bulirsch. *Numerische Mathematik*. Springer Verlag, New York, Heidelberg, Berlin, 1990.

[128] Z. Strakoš. On the real convergence rate of the conjugate gradient method. *Linear Algebra Appl.*, 154/156:535–549, 1991.

[129] Z. Strakoš. Convergence and numerical behavior of the Krylov space methods. In *NATO ASI*. Kluwer Academics, 1998.

[130] Z. Strakoš. Rounding errors in the symmetric and nonsymmetric Krylov space methods: Do they behave differently? to appear in: Iterative Methods in Scientific Computations II, 2000.

[131] Sun FORTE tools. `www.sun.com/forte/developer/`.

[132] L. N. Trefethen. *Numerical Linear Algebra*. SIAM, Philadelphia, 1997.

[133] H. A. van de Vorst. The convergence behavior of preconditioned CG and CG-S in the presence of rounding errors. In O. Axelson and L. Kolotilina, editors, *Preconditioned Conjugate Gradient Methods*, Lecture Notes in Mathematics 1457, pages 47–69. Springer Verlag, Berlin, New York, 1990.

[134] H. A. van de Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 13:631–644, 1992.

[135] R. Varga. *Matrix Iterative Analysis*. Prentice-Hall Inc., Englewood Cliffs, NJ, 1962.

[136] T. Veldhuizen. Algorithm specialization in C++. `extreme.indiana.edu/~tveldhui/papers/`, 1995.

[137] T. Veldhuizen. Expression templates. *C++ Report*, 7:26–31, 1995.

[138] T. Veldhuizen. Techniques for scientific C++. `extreme.indiana.edu/~tveldhui/papers/techniques/`, 1999.

[139] T. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran? In *Proc. ISCOPE*. Springer, 1997.

[140] H. F. Walker. Implementation of the GMRES method using Householder transformations. *SIAM J. Sci. Statist. Comput.*, 9:152–163, 1988.

[141] R. Weiss. *Convergence Behavior of Generalized Conjugate Gradient Methods*. PhD thesis, Universität Karlsruhe, Karlsruhe, Germany, 1990.

[142] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Clarendon Press, Oxford, 1988.

[143] H. Wozniakowski. Roundoff error analysis of a new class of conjugate gradient algorithms. *Lin. Alg. Appl.*, 29:507–529, 1980.